

Projects in Automatic Control Spring 2025

Charlotta Johnsson
Editor



LUND
UNIVERSITY

Department of Automatic Control

Technical Report TFRT-7673
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Printed in Sweden by Tryckeriet i E-huset.
Lund 2025

Preface

The Department of Automatic Control at Lund University gives a project course in Systems, Control and Learning (FRTN70) twice per year. The course is given at the advanced level (7.5 ECTS credits), and the students work in small teams to achieve a common goal. The projects typically involve a real-world estimation or control problem with relevance to industrial or other applications. In this course, the students get an opportunity to explore implementational aspects of concepts they have learned in previous control systems courses. With a faculty member or doctoral student as an advisor, the groups independently formulate an objective and an associate time plan. Subsequent activities typically involve modelling, controller design, implementation, documentation, and verification. The students present their work through two feedback seminars, an oral presentation, a demonstration session, and a written report.

The reports of the spring 2025 edition of the course are presented in this booklet. This year, the cohort consisted of 25 students, working in teams of 2-5 persons. The course included 6 projects; Ball-E * 2, Making Crazyflie Crazy, Jenga Tower, Robot Jenga, and Explainable AI for Critical Care Prediction. Every group managed to perform successful and satisfactory real-time experiments to generate the final experimental results ready in time for the demonstration in May 2025.

Doctoral students Martin Gemborn-Nilsson, Zheng Jia, Sebastiano Fregnan, and Post-Doc Henry Pigot have served the course as project advisors, research engineers Alexander Pisarevsky, Anders Blomdell and Anders Nilsson have supported the groups throughout their projects, and Mika Nishimura has help with student registration and related matters.

To find out more about the course, please visit <http://www.control.lth.se/course/FRTN70>.

Lund, June 2025

Charlotta Johnsson, Course responsible FRTN70

Contents

Ball-E	7
<i>Thorselius A, Gruber K, Trane Lindberg N, Kulle V</i>	
BALL-E	15
<i>Löving A, Neckmar D, Stenroos J, Ruetschi M, Carnehult R</i>	
Making Crazyflie Crazy	23
<i>Tianci W, Yaliang C, Haobo Z, Faraji V, Changsheng H</i>	
Jenga Tower	29
<i>Lagerstedt A, Sweger A, Sjöstedt P, Smådal S, Chen Z</i>	
Robot Jenga	35
<i>Hansson A, Liu S</i>	
Explainable AI for Critical Care Prediction	43
<i>Knezevic D, Wik H, Bergkvist T, Panai Ersbo R</i>	

Ball-E

Axel Thorselius¹ Konrad Gruber² Niclas Trane Lindberg³ Viktor Kulle⁴

¹axel.thorselius@gmail.com ²konrad.gruber.2001@gmail.com ³trindberg@gmail.com
⁴vi8354ku-s@student.lu.se

Abstract: The objective of this project is to design and implement a control system for Ball-E, a robot balancing on top a basketball. In real time, the system combines control theory, sensor data processing and embedded systems to achieve stability. Ball-E uses an Arduni-based controller, and inertial measurement unit (IMU) to measure tilt and angular velocity and three omni-wheels driven by stepper motors situated on the surface on the ball. The system was modeled as an inverted pendulum approximation and assumes a rigid body with no slip. A linearized state-space representation was created and was utilized in combination with a Linear-Quadratic Regulator (LQR) to stabilize the system in discrete time. The input data from the IMU was filtered using a complementary filter to mitigate noise and drift, while wheel accelerations are computed to maintain balance via force distribution calculations. The initial LQR gains derived from the model were proved insufficient in practice. The gains were therefore manually tuned. The final implementation successfully managed to balance the robot for up to two minutes. Despite limitations and model approximations, the project demonstrated the viability of balancing a robot on a ball using LQR control.

1. Introduction

The goal of this project is to balance a robot, Ball-E, on top of a basketball. Ball-E is a simple system consisting of a main body that houses a battery, an Arduino microcontroller, sensors (including a gyroscope and accelerometer), stepper motors that drive three omni-wheels that moves the basketball to maintain balance of the system. In addition to ensuring that Ball-E achieves equilibrium on top of the basketball, another goal of the project is to gain a better understanding of automatic control and developing skills whilst working in a group project.

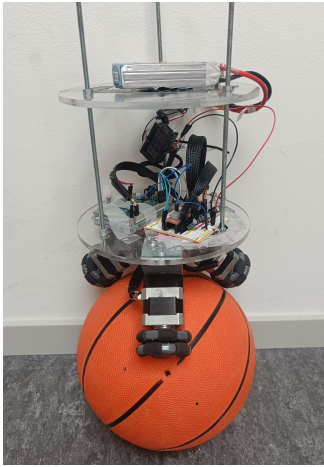


Figure 1. Ball-E robot balancing on top of the basketball

2. Modeling

Modeling the system is an important but complex process and its difficulty is largely dependent on the tolerance of necessary accuracy. Creating a fully accurate model, even on a seemingly simple system like Ball-E, would be very difficult and time consuming due to the given structure of the robot as well as chosen hardware and software systems, all contributing to inconsistencies. However, upon analysis, the system closely resembles an inverted pendulum. Therefore, an inverted pendulum will serve as the primary model for understanding and designing the control system and encapsulates the following assumptions and simplifications.

2.1 Assumptions & Simplifications

To simplify the system it was assumed that all of the components in the system including the basketball, the robot and its wheels and the floor have been considered as rigid to disregard possible deformations. Further assumptions made are that there is no slip between the robot's wheels and the basketball as well as between the basketball and the floor. In addition, the friction forces have been ignored. These assumptions make it significantly easier to model the system in a manageable way.

Table 1. System Parameters Notations.

Parameter	Notation
Body mass	m_{bo}
Ball mass	m_{ba}
Length, ball center to COM	l
Ball radius	r_{ba}
Omni-wheel mass	m_{wh}
Wheel radius	r_{wh}

2.2 System Modeling

A large part of the modeling was based off of a few articles and projects which closely resembles the Ball-E system with the same purpose of balancing the system. These all assume the inverted pendulum model [2] to different degrees, see as an example [1].

An approach of modeling the system in 2D was taken, as depicted in Figure 2, with it's system coordinates in the xz -plane. The Ball-E robot/body in this model is simplified as a wheel, which is represented by the smaller circle in the figure and is on top of the basketball, which is represented by the bigger circle. This model is then duplicated identically in the yz -plane and the two models will be superpositioned in order for the system to work in any direction. The general coordinates chosen for this system are s as the distance the basketball has moved horizontally and the angle θ as the rotation of the robot around the basketball, denoted as

$$q = (s, \theta)^T. \quad (1)$$

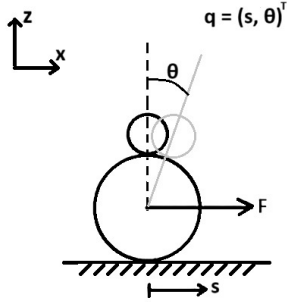


Figure 2. Model of the system in 2D.

To find equations of motion for the system, the Euler-Lagrange equations were used, which utilizes the kinetic and potential energies of the system expressed through the generalized coordinates q and it's time-derivative $\dot{q}$. Let f represent the terms of the Euler-Lagrange equations

$$f = \frac{d}{dt} \left(\frac{\partial T}{\partial \dot{q}} \right) - \frac{\partial T}{\partial q} + \frac{\partial V}{\partial q} - Q_{nc} - F, \quad (2)$$

where T is the kinetic energy, V is the potential energy and F is generalized external forces. As we have neglected any generalized non-conservative forces (like friction), $Q_{nc} = 0$.

To express the kinetic energy, the position p and then the velocity v of the robot from Figure 2 are needed, stated as vectors with x - and z -components. More specifically, p is the position of the Ball-E body's center of mass, as it rotates around the basketball. Similarly, v represents the velocity of the Ball-E body as it rotates around the basketball. With the origin in the center of the basketball, the position and velocity vectors become as follows:

$$p = \begin{bmatrix} l \sin \theta + s \\ l \cos \theta \end{bmatrix}$$

$$v = \dot{p} = \begin{bmatrix} l \dot{\theta} \cos \theta + \dot{s} \\ -l \dot{\theta} \sin \theta \end{bmatrix}$$

Now the velocity of the robot can be inserted into the equation for kinetic energy (3), which together with the equation for potential energy (4) where $h = p_z = l \cos \theta$, will be utilized in the Euler-Lagrange formula to find the equation of motion for the robot (5).

$$T = \frac{1}{2} m v^2. \quad (3)$$

$$V = m g h. \quad (4)$$

$$f = \begin{bmatrix} (\frac{7m_{ba}}{5} + m_{bo})\ddot{s} + m_{bo}l\ddot{\theta} \cos \theta - m_{bo}l\dot{\theta}^2 \sin \theta \\ m_{bo}l^2\ddot{\theta} + m_{bo}\ddot{s}l \cos \theta - m_{bo}gl \sin \theta \end{bmatrix} - \begin{bmatrix} u \\ 0 \end{bmatrix} = 0. \quad (5)$$

The equation of motion f is then linearized around an equilibrium point, which refers to a state in which the system can remain indefinitely if there are no perturbations. By definition this means that $\ddot{q} = 0$ and $\dot{q} = 0$. This system will have an equilibrium configuration denoted $\bar{q}$, when the applied balancing force $\bar{u} = 0$, $\theta = 0$ and at an arbitrary s , but is for convenience sake set to $s = 0$, representing the basketball and Ball-E in its initial conditions and desired state. The linearization of the equation of motion results in the mass matrix M , damping matrix D and stiffness matrix G :

$$M = \frac{\partial f}{\partial \ddot{q}} \big|_{\bar{q}, \bar{\dot{q}}, \bar{\ddot{q}}} = \begin{bmatrix} 7m_{ba}/5 + m_{bo} & m_{bo}l \\ m_{bo}l & m_{bo}l^2 \end{bmatrix}, \quad (6)$$

$$D = \frac{\partial f}{\partial \dot{q}} \big|_{\bar{q}, \bar{\dot{q}}, \bar{\ddot{q}}} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}, \quad (7)$$

$$G = \frac{\partial f}{\partial q} \big|_{\bar{q}, \bar{\dot{q}}, \bar{\ddot{q}}} = \begin{bmatrix} 0 & 0 \\ 0 & -m_{bo}gl \end{bmatrix}. \quad (8)$$

Now the equations of motions are expressible in the linearized matrix form. As the stiffness matrix G has a negative eigenvalue, the equilibrium configuration is unstable and a generalized external force F is needed to keep it in this state:

$$M\ddot{q} + Gq = F. \quad (9)$$

To make this compatible with our chosen controller, the equations of motion need to be rewritten in a state-space form. The state vector for the 2D model in the xz -plane is defined as

$$x = \begin{bmatrix} q \\ \dot{q} \end{bmatrix} = \begin{bmatrix} s \\ \theta \\ \dot{s} \\ \dot{\theta} \end{bmatrix}, \quad (10)$$

Which implies that:

$$\dot{x}_1 = x_2 = \dot{q}. \quad (11)$$

Using (9)

$$\ddot{x}_2 = \ddot{q} = -M^{-1}Gx_1 + M^{-1}F, \quad (12)$$

which results in the following state space form and output equation:

$$\dot{x} = Ax + Bu, \quad (13)$$

$$y = Cx. \quad (14)$$

Where matrix A, B and C are represented as follows:

$$A = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & (-5m_{bo}g)/(7m_{ba}) & 0 & 0 \\ 0 & (7m_{ba} + 5m_{bo})g/(7m_{ba}l) & 0 & 0 \end{bmatrix},$$

$$B = \begin{bmatrix} 0 \\ 0 \\ 5/(7m_{ba}) \\ -5/(7m_{ba}l) \end{bmatrix}, \quad C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

2.3 Balancing Force to Acceleration of the Omni-Wheel

To relate acceleration of the omni-wheel to the force required for balancing, a simplified physical model is used. Wheel acceleration is mapped to a horizontal force acting on a cart-like system, similar to the approach described in the inverted pendulum model from [2]. The wheel refers to the virtual representation of the Ball-E robot and thereby also as the combined rotational force utilized to rotate the basketball. In actuality, the three omni-wheels enforce the rotational force on the basketball but, by dividing the system first into a single wheel responsible for the complete rotation and then to the individual acceleration by the omni-wheel, the calculations are much simplified. These calculations are based on the total energy of the basketball, T_{tot} , which becomes a combination of the translation energy, T_{trans} , and the rotational energy, T_{rot} .

$$T_{tot} = T_{trans} + T_{rot} = \frac{1}{2}m_{ba}v_{ba}^2 + \frac{1}{2}I_{ba}\omega_{ba}^2 \quad (15)$$

Assuming no slipping between the wheel and the ball the following assumption can be made,

$$\omega_{ba} = \omega_{wh} \frac{r_{wh}}{r_{ba}} \quad (16)$$

Inserting 16 into 15,

$$T_{tot} = \frac{1}{2}(m_{ba}v_{ba}^2 + I_{ba})\omega_{wh}^2 \left(\frac{r_{wh}}{r_{ba}}\right)^2 \quad (17)$$

Finding the derivative of T_{tot} gives

$$\frac{dT_{tot}}{dt} = \frac{1}{2}(m_{ba}v_{ba}^2 + I_{ba})\dot{\omega}_{wh}^2 \left(\frac{r_{wh}}{r_{ba}}\right)^2 \quad (18)$$

Using the relationship between the energy and force along with 16 gives

Table 2. Kinetic Energy Component Notations

Notation	Description
v_{ba}	Horizontal speed of the ball
ω_{ba}	Rotational speed of the ball
I_{ba}	Inertia of the ball

$$\frac{dT_{tot}}{dt} = Fv_{ba} = Fr_{wh}\omega_{wh} \quad (19)$$

Finally, combining 19 and 18, and solving for $\dot{\omega}_{wh}$ gives

$$\dot{\omega}_{x,y} = \frac{r_{ba}^2}{(m_{ba}r_{ba}^2 + I_{ba})r_{wh}} F_{x,y} \quad (20)$$

This is the conversion from the force to the rotational wheel acceleration necessary for the body to achieve equilibrium. This conversion is made in the vertical 2D plane, assuming that one virtual omni-wheel is responsible for the entire necessary acceleration. Therefore, another conversion is necessary from the wheel acceleration to the individual omni-wheels.

2.4 Wheel Acceleration to Individual Omni-wheel Accelerations

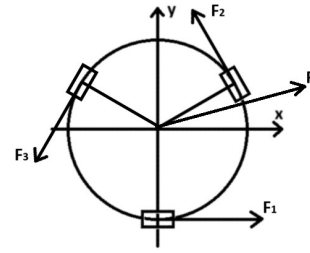


Figure 3. Forces acting on the ball in the xy -plane.

To proportionally divide the load of each omni-wheel F_1 , F_2 , and F_3 when an arbitrary force F (where $F = [F_x, F_y]$) is present on the basketball, a force diagram the basketball was made, see Figure 3. The total force is divided into the local force vectors, $F_{x,\ell}$ and $F_{y,\ell}$, not to be confused with the total force vectors F_x and F_y as these are scaled by Equation 3.

$$(x) : F_1 - F_2 \sin(30^\circ) - F_3 \sin(30^\circ) = F_{x,\ell} \quad (21)$$

$$(y) : F_2 \cos(30^\circ) - F_3 \cos(30^\circ) = F_{y,\ell} \quad (22)$$

$$(\theta) : r_{ball}(F_2 + F_3 + F_1) = 0 \quad (23)$$

Equation 23 is necessary to have a solvable equation system but also enforces that the basketball doesn't rotate horizontally. Solve for F_1 , F_2 , and F_3 :

$$F_1 = \frac{2}{3}F_{x,\ell} \quad (24)$$

$$F_2 = -\frac{1}{3}F_{x,\ell} + \frac{1}{\sqrt{3}}F_{y,\ell} \quad (25)$$

$$F_3 = -\frac{1}{3}F_{x,\ell} - \frac{1}{\sqrt{3}}F_{y,\ell} \quad (26)$$

Using the relationship between force and rotational acceleration along with 20. The final set of equations becomes,

$$\dot{\omega}_{wh1} = 2\dot{\omega}_x \quad (27)$$

$$\dot{\omega}_{wh2} = -\dot{\omega}_x + \sqrt{3}\dot{\omega}_y \quad (28)$$

$$\dot{\omega}_{wh3} = -\dot{\omega}_x - \sqrt{3}\dot{\omega}_y \quad (29)$$

3. Controller Design

Since the system is controlled by a microcontroller (Arduino Due), which operates in discrete time, the continuous-time model must be converted into a discrete-time state-space representation. This involves converting the model from:

$$\dot{x}(t) = Ax(t) + Bu(t) \quad (30)$$

to:

$$x[k+1] = A_d x[k] + B_d u[k] \quad (31)$$

This transformation was performed using the `c2d` function in MatLab, which applies a zero-order hold method to discretize the system.

A Linear-Quadratic Regulator (LQR) was then selected as the controller. It computes the control input u based on the linear state feedback law [3]:

$$u = -Kx[k] \quad (32)$$

Here, K is a feedback gain matrix computed to stabilize the system by driving the state x towards a desired reference. In practice, this corresponds to applying a corrective force to keep the system upright. The gain matrix K is computed offline and implemented in the Arduino Due code.

3.1 Cost Function and Riccati Equation

The feedback law in (32) is derived by minimizing a discrete-time quadratic cost function that penalizes both state deviation and control effort:

$$J = \sum_{k=0}^{\infty} (x_k^T Q x_k + u_k^T R u_k) \quad (33)$$

The matrices Q and R are weighting matrices that reflect the relative importance of minimizing state errors versus control input magnitude. The solution to this optimization problem involves solving the discrete-time algebraic Riccati equation:

$$P = A^T P A - A^T P B (R + B^T P B)^{-1} B^T P A + Q \quad (34)$$

Once the matrix P is found, the optimal gain matrix K is given by:

$$K = (R + B^T P B)^{-1} B^T P A \quad (35)$$

In this project, Matlabs's `dlqr`-function was used to compute the gain matrix K directly from the discretized system matrices A_d , B_d , and the chosen weights Q and R .

4. Implementation

4.1 Equipment

The main equipment used for Ball-E are an Arduino Due/Mega, 3 58mm Omni Directional Wheels, 3 Stepper online Nema 17 + A4988 drivers LSM9DS1 MEMS IMU, and a LiPo battery.

4.2 Sensors and Filtering

To balance the Ball-E on top of a basketball we need accurate measurements of its tilt angle, angular velocity, velocity and position. For this we use an Inertial Measurement Unit (IMU), specifically the LSM9DS1 which provides both accelerometer and gyroscope data. The Accelerometer measures the acceleration along the x,y and z axes while the gyroscope measures angular velocity around these axes.

Tilt Estimation To estimate the robot's tilt angle we can calculate it directly from the accelerometer readings using trigonometry relationships (36,37). Relying only on the accelerometer data produces a noisy signal making it unsuitable for precise control. Alternatively we could estimate the angle by integrating the gyroscope data over time. While this method initially provides a smooth signal it suffers from drifting. The small errors accumulate over time leading to increasingly inaccurate angle estimates. To overcome these limitations we combine accelerometer and gyroscope measurements using a complementary filter. The complementary filter merges the both sensors. Applying a low pass filter to the accelerometer data to retain long term stability and a high pass filter to the gyroscope data to capture rapid changes while minimizing the drift. The resulting estimates is much more reliable.

The roll angle on accelerometer is calculated as

$$Roll = \arctan\left(\frac{a_y}{\sqrt{a_x^2 + a_z^2}}\right), \quad (36)$$

and the pitch angle as

$$Pitch = \arctan\left(\frac{-a_x}{\sqrt{a_y^2 + a_z^2}}\right). \quad (37)$$

Where, a_x , a_y , and a_z are the accelerometer readings along the x-, y-, and z-axes, respectively.

The complementary filter is implemented using the following

$$\theta(t) = \alpha(\theta(t - \Delta t) + \omega_{gyro}(t) \cdot \Delta t) + (1 - \alpha)\theta_{accel}(t). \quad (38)$$

Where, $\theta(t)$ is the estimated tilt angle at time t , $\omega_{gyro}(t)$ is the angular velocity from the gyroscope, Δt is the sampling interval and $\theta_{accel}(t)$ is the tilt angle computed from the accelerometer data. The constant α determines how much influence each sensor has on the final estimate.

Velocity & Position Estimation In addition to angle and angular velocity the LQR controller also requires estimates of the robot's velocity and position on the ball. Since there is no direct measurement of Ball-E's position, we must estimate it. This is done by calculating the number of steps sent to the motors and combining this with the angle. In reality the wheels do not always move in sync with the ball. Slippage can occur

meaning that the movement of the wheels do not make change in the position. To account for this we introduce a slip factor. The slip factor helps correct the position estimate by slightly reducing the accumulated position from previous steps. This reduction takes into account the slippage that may occur. The formula used for calculating the position below:

$$s(t) = \alpha \cdot s(t) + \Delta s_w \quad (39)$$

$$s_o(t) = s(t) + r \cdot \theta(t) \quad (40)$$

Where, $s(t)$ represents the position before applying the offset due to the tilt angle, while $s_o(t)$ denotes the position after including this angular offset. The term r refers to the radius of the ball, and Δs_w is the change in distance caused by the wheel's motion. The variable $\theta(t)$ corresponds to the tilt angle.

To calculate the velocity of Ball-E, we use the same approach as for the position estimation. Instead of accumulating the distance over time, we simply calculate the change in distance divided by the time. This gives us the velocity of Ball-E as

$$v(t) = \frac{\Delta s_w}{\Delta t} + r \cdot \dot{\theta}, \quad (41)$$

where, $v(t)$ denotes the velocity, and r is the radius of the ball. The term Δs_w represents the change in distance from the wheel, while $\dot{\theta}(t)$ is the angular velocity. Lastly, Δt refers to the sampling time.

IMU Specifications Our IMU operates at a maximum sampling rate of 119 Hz. To maintain system stability and synchronization with the controller we have chosen a sampling time of 20 ms (equivalent to a frequency of 50 Hz) which is well within the IMU's capabilities. This choice provides a good balance between response time and computational load.

4.3 Motor Control

As the motors used are stepper motors that rotate in discrete steps instead of continuously rotating. One step is achieved by sending a high-low pulse to the step pin connection on the motor. This implies that the rotational velocity of the motor is controlled by the delay between pulses. The motors have 200 steps/revolution and the drivers are configured to divide into quarter-steps resulting in 800 steps/revolution.

4.4 Code Implementation

To achieve a control loop with fixed and reliable timings without freezing or blocking, an interrupt-based approach was chosen for the motor acceleration control. The control structure is divided into two substructures, one main loop running continuously and one function that executes upon each hardware interrupt.

Main Control Loop The main control loop reads and filters the sensor data and computes the pitch and roll angles (38) and estimates the position (40) and velocity using step counts from each wheel (41). The gyroscopic readings is used directly as the angular velocity. These are used in the LQR to calculate the desired x - and y acceleration (10), (32). Finally, the x - and y accelerations are translated into the necessary rotational acceleration for each of the three wheels according to (27),

(28), (29) with a separate WheelDivision-function. To avoid race conditions, interrupts are disabled during this final step. This loop is run every 20 ms with non-blocking timing with an elapsed time check. This means that the loop runs continuously, without any coded delay or sleep, and only executes the above-mentioned calculations if 20 ms has elapsed.

```
1 // Main-loop pseudo-code. Runs every 20 ms
2 void loop() {
3   if((now - prev) > 20 ms) {
4     read & filter sensor values;
5     estimate position & velocity;
6     aX, aY = -Kx;
7     global a1, a2, a3 = WheelDivision(aX, aY);
8   }
9 }
```

Interrupt Function The interrupt function computes the momentary velocity from the desired acceleration and elapsed time since the previous interrupt, translates that into a pulse delay and steps the motor if that delay has elapsed and increments or decrements the step count depending on the direction. This is carried out for each of the three motors.

```
1 // Interrupt pseudo-code. Runs every 0.5 ms.
2   Repeated for each omni-wheel.
3 interrupt() {
4   v1 += a1*dt;
5   step_interval = 1 / v1;
6   if((now - prev) > step_interval){
7     stepPin(HIGH);
8     stepPin(LOW);
9     if(v1 > 0){
10      step_counter++;
11    } else {
12      step_counter--;
13    }
14  }
```

5. Testing

Initially when the testing period began there was a desire to gather data from the different tests. However, in order to gather data the Arduino had to be plugged into a computer which led to the motors spinning uncontrollably. This made it impossible to conduct proper tests, especially during the balancing optimization, which is the reason behind the lack of hard data to back up the tests.

5.1 System Confirmation

The main part of the testing process consisted of optimizing the values of the feedback gain vector, K , as this is the main

Table 3. System Parameter's Values

Notation	Value
m_{bo}	2.139 kg
m_{ba}	0.536 kg
l	0.2848 m
r_{ba}	0.1218 m
m_{wh}	0.077 kg
r_{wh}	0.029 m

contributor to balancing the system. However, before this, each subpart had to be tested to check for system breaking faults or errors which could influence the performance of the system.

Sensor As the input values from the sensors and their filtered value is crucial to accurately control the system it was first to be tested. This was done by attaching an iPhone to the structure and using the inbuilt angle measuring tool. The body was tilted and the continuously recorded sensors values were compared to the reading values of the iPhone measuring tool. The test showed that there was $< 1^\circ$ difference in the recorded sensors, thereby, it was declared accurate enough to not pose any problems when optimizing the system.

Filter To test the function of the complementary filter and determine a good value for α , which determines the weighting between the accelerometer and gyroscopic sensor data according to Equation 38, a simple manual back and forth tilt test was conducted. Four different tests with $\alpha = 1, 0.99, 0.95$ and 0.90 were performed and the accelerometer, gyroscope, and filtered angle data were plotted for the x direction. No noticeable difference in accuracy of the filtered angle was observed for the different α -values. Thus we chose to continue with $\alpha = 0.98$ which was chosen early in the project as a compromise between widely accepted reasonable values.

Motor Direction Control The next test was to ensure that the motors could move the ball in the desired direction. More specifically, Equations 27, 28, and 29 as these calculate the driving direction by combining the calculated forces F_x and F_y . This was done by inverting the body and placing the basketball on top of the motors. Several directions were tested by inputting values for forces F_x and F_y in the Arduino code and observing if the balls movement matched the desired direction. The sequential tests proved that Equations 27, 28, and 29 were correctly calculated as balls movement match the inputted direction.

Thereafter it was necessary to test the complete, non-optimized, system. This was done to ensure that the complete chain from the input values of the sensors to the final direction output of the motors was functioning as modeled. The Ball-E body was placed on top of the basketball and, whilst being held, was tilted to check that the motors would roll the ball back underneath the body. The gain feedback vector, K , was purposely set to have a slow correction to make it easier to observe how the system reacted to the tilted body. Throughout all the tested tilt angles the basketball would roll slowly back to balance underneath the Ball-E body.

This concluded the necessary system confirmation tests which meant that the next step was to optimize the gain feedback vector, K , starting from the initial values taken from the original model.

5.2 LQR Optimization

When using the following Q and R start values:

$$Q = \begin{bmatrix} 100 & 0 & 0 & 0 \\ 0 & 100 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$R = 0.01.$$

The resulting K vector from the model calculated in Mat-Lab ended up as:

$$K = [-53.1 \quad -138.9 \quad -35 \quad -18.8].$$

When testing these K values, Ball-E couldn't balance itself and a manual tuning approach of the K values was adopted, based on observations of the systems reaction to changes in the values. This was a more direct and easy method rather than changing the Q and R values as it would be more difficult knowing which parameters to balance and use, even though it would eventually give the same K values.

It became evident that the indexes correlated with the position s and angle θ impacted the overall force generated by the omni-wheels for Ball-E to correct itself from a tilted angle to equilibrium. Furthermore, the indexes correlated with the velocity $\dot{s}$ and angular velocity $\dot{\theta}$ impacted the acceleration of the omni-wheels, thus the response-rate of the robot.

6. Results

The final implementation of our LQR controller was able to maintain balance for several seconds during multiple test runs. Initially we computed the LQR gain matrix K based on theoretical model of the system. However when tested on Ball-E, the calculated gain resulted in unstable behavior. The system responded too aggressively which led to loss of balance. To overcome this we manually tuned the K values through trial and error. After several iterations we found a set of gains that allowed the robot to consistently stabilize itself and remain upright for durations ranging from a few seconds to two minutes. These results indicate that, although the theoretical gain were not directly applicable, the LQR controller was capable of maintaining balance when tuned to the real system.

During testing we observed occasional instability. In several cases the Ball-E began to oscillate back and forth after some time, eventually leading to a loss of balance and falling over. A common cause of this behavior appeared to be slipping or bumps between the ball and the Ball-E, which introduced errors in the position estimation. When the ball lost traction the system could no longer accurately track its position resulting in more and more movement and eventually instability.

7. Discussion

In conclusion, the project was a success, as the main goal was achieved: Ball-E was able to balance on top of the basketball for several seconds and, in some test runs, for up to two minute. Additionally, the group members gained valuable experience in automatic control and collaborative development. However, the design process did not always align with initial expectations, and several challenges arose that led to deviations from the original plan.

This section reflects on the lack of data, the gap between modeling assumptions and real-world behavior, the effectiveness of the chosen control strategy, potential improvements, and the group's collaboration and development process.

7.1 Lack of Data

As mentioned previously, data from the testing period could not be collected. As the technical issue only caused problems when running the motors, the sensors and filtering test could still be conducted which ensured that the input values were not incorrect. Unfortunately the motor direction control tests were slightly limited as the exact directions couldn't be measured from the data, however, extensive testing was convincing enough to prove that the omni-wheels mostly ran in the accurate direction. Surprisingly enough, the lack of data had very little impact on the LQR optimization process as almost all the changes were made based on visual queues. However, the major issue came about when reviewing the entire project process. Having data on each part of the process would allow for a far greater analysis and evaluation on each subsystem. This means that most of the discussions and conclusions are based on observation and the collective belief of the project group. Unfortunately this reduces the trustworthiness of the conclusions and it is definitely something to be further looked into for future improvements.

7.2 Modeling Assumptions vs. Reality

The modeling assumptions proved to be a major source of discrepancy between theoretical predictions and real-world results. This became evident during the tuning of the feedback gain vector K , where the initial values, derived from the model, were significantly incorrect. The inverted pendulum model turned out to be an oversimplification and did not sufficiently represent the dynamics of the actual system. Another likely contributors include the exclusion of body inertia, assumptions of rigid-body behavior, and the neglect of slippage between the wheels and the basketball. In practice, Ball-E's body deformed under acceleration, with visible twisting and wobbling, and the omni-wheels frequently slipped on the basketball surface.

Due to such hardware uncertainties, achieving a highly accurate model would be difficult even without the initial simplifications. Deciding on how to allocate time is a key factor during a project and in this case it was clear that optimizing the regulator was more efficient. Far more hours were spent on modeling and designing the system which ultimately did not yield useful information and it is therefore believed that attempting to optimize the model would not be worth allocating the time for.

7.3 Control Strategy Effectiveness

Shifting the focus from model-based gain values to empirical tuning immediately improved system performance. The LQR controller proved effective, particularly due to its ability to regulate the state vector $[s, \theta, \dot{s}, \dot{\theta}]$, which allowed it to respond well to disturbances.

For instance, incorporating control over the total displacement s prevented the basketball from drifting excessively, improving stability and reducing the impact of external disturbances—such as contact with uneven or slippery surfaces on the ball. This highlighted the strength of LQR in handling multi-variable regulation, even when applied to a simplified model.

7.4 Potential System Improvements

To improve consistency and robustness, several hardware changes could be made. One major issue was inconsistent traction between the omni-wheels and the basketball. The textured surface of the ball, particularly the rubber seams, often caused slipping. Replacing the ball with a smoother, more uniform surface would likely yield more predictable behavior and better controller performance.

Additionally, increasing the rigidity of the mechanical structure would reduce wobbling and vibration during motion, improving sensor accuracy and making the system easier to control. Improving mechanical reliability, such as having a smoother ball, would also reduce the effect of model mismatch.

7.5 Team Collaboration & Development Process

Overall, the team collaborated effectively and followed the initial project plan. Responsibilities were well divided, and problems were typically addressed quickly and collaboratively. However, during some phases—such as testing—a more detailed, structured approach would have improved efficiency and helped avoid redundant work.

The success of the project was largely due to strong communication, mutual support, and collective problem-solving. This experience highlighted the value of combining individual strengths to achieve a shared technical goal.

References

- [1] K. van der Blonk. *Modeling and Control of a Ball-Balancing Robot*. Accessed: 2025-05-17. Master's thesis. University of Twente, 2014. URL: <https://essay.utwente.nl/65559/>.
- [2] Wikipedia contributors. *Inverted pendulum*. Accessed: 2025-05-17. 2025. URL: https://en.wikipedia.org/wiki/Inverted_pendulum.
- [3] Wikipedia contributors. *Linear-quadratic regulator*. Accessed: 2025-05-17. 2025. URL: https://en.wikipedia.org/wiki/Linear%E2%80%93quadratic_regulator.

BALL-E

Alice Löving¹ Dante Neckmar² Jonathan Stenroos³ Michael Ruetschi⁴
Rasmus Carnehult⁵

¹alice.lovung@live.se ²dante@neckmar.se ³jonathanstenroos@hotmail.com
⁴mickeruetschi@hotmail.com ⁵rasmus.carnehult@gmail.com

Abstract: This project aims to develop a controller for a ball-balancing robot. The robot consists of three omni-wheel-equipped stepper motors, an Arduino Due microcontroller, and an IMU chip providing accelerometer and gyroscope readings. The goal is for the robot to be able to drive its stepper motors such that it can remain upright and balanced when placed on a basketball. One could visualize the basketball as a movable ball joint and the robot as a three-dimensional inverted pendulum. To model the system, inspiration from the classical inverted pendulum on a cart problem was taken. Two independent controllers were developed, handling pitch and roll, respectively. From our results, it was concluded that only a simple PD controller was needed to achieve stability. The model-derived LQR controller did, however, not give us a stable system, and we think that a new modeling approach is needed if a more advanced controller is to be developed.

1. Introduction

This report covers the development of a ball-balancing robot named Ball-E. The robot is shown in Figure 1. The project goal is for Ball-E to be able to balance on a basketball. This has been accomplished using techniques from signal processing and automatic control. The heart of the robot is an Arduino microcontroller connected to three stepper motors placed in a tripod configuration. The wheels on the motors are so-called omni wheels, whose circumference is made up of small rubber rollers perpendicular to the turning direction. Importantly, the design of the omni wheels allows them both to be driven along their turning direction like a normal wheel and to slide laterally with low friction. Besides stepper motors, the microcontroller also has access to an accelerometer and a gyroscope, measuring force and angular velocity of the robot, respectively. These sensors will be used to estimate the tilt and position of the robot, the states that will be used by the controller.

To achieve our balancing goal, we have decided to use a Linear-Quadratic regulator (LQR). Contrary to a PID controller, this type of controller requires a model of the system. To derive this model, reports from similar projects have been studied. Mainly, the work of Koos van der Blonk [3] has been used. Importantly, van der Blonk discusses a modeling approach that allows us to describe the system using two independent but identical 2D models, simplifying the mathematics significantly. The author details how the system's three drive wheels can be modeled as two perpendicular virtual wheels placed on top of the basketball, representing directions in which the robot can apply force. Using this formulation, the system's behavior in the x-z and y-z planes can be modeled equally.

The models for these perpendicular planes of motion, based on the virtual wheel formulation, will be derived from the classical inverted pendulum on a cart problem. This will be discussed in later sections. With the 2D models, one can



Figure 1. The Ball-E robot

develop two independent LQR controllers, which will be responsible for maintaining balance in the robot's x-z and y-z planes, respectively.

The purpose of the project is to get practical knowledge about modeling and automatic control. In the report, various improvements and capabilities that can be applied to future work will be discussed.

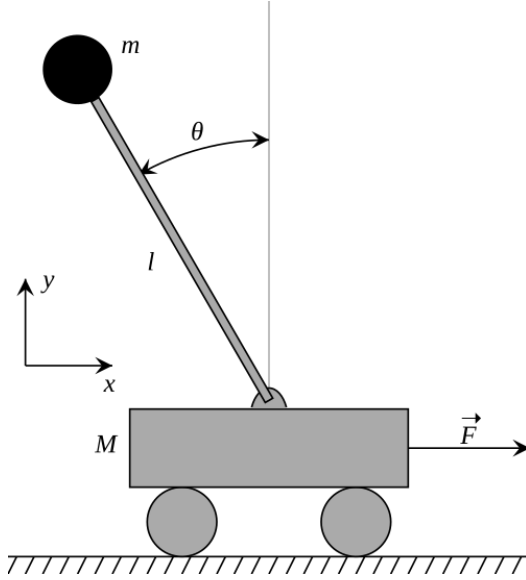


Figure 2. The cart system [7]

2. Modeling

This section will cover our modeling approach. As stated in the introduction, the system will be described using two 2D models. Each model will capture the dynamics of a vertical slice (x-z or y-z plane) of the system based on van der Blonk's [3] virtual wheel formulation. From the point of view of the vertical slices, the robot and basketball can be seen as a 2D inverted pendulum with a base consisting of a virtual drive wheel mounted on a larger wheel.

2.1 Inverted Pendulum on a Cart

The inverted pendulum on a cart is a classical automatic control problem. The goal is to control a cart moving along a horizontal track such that the inverted pendulum mounted on top of the cart does not fall over. The system is shown in Figure 2. The system is inherently unstable, as any deviation from the upright position causes the pendulum to fall unless corrective action is taken [7].

This corrective action is applied as a force $\vec{F}$ acting on the cart, resulting in a torque at the pivot. Ideally, $\vec{F}$ is controlled such that $\theta \approx 0^\circ$. The mass distribution of the system is described as the mass above the pivot m , the mass of the cart M , and a massless pole of length l , distancing the masses m and M . The model has four states: the tilt of the pendulum, the position of the cart, and their respective derivatives [7].

To use this model in our system, a fundamental difference between our system and that of the cart problem needs to be circumvented. The fact that our drive wheels are powered by stepper motors makes it non-trivial to emulate the control signal $\vec{F}$. This issue is discussed in Section 2.3.

2.2 Virtual wheel conversion

The strength of the virtual wheel-based drive system model that van der Blonk [3] proposes is that it is easy to translate the virtual wheel control signals to the real system. To move from the virtual to the real system, one simply needs to apply a transformation matrix. The matrix is described in the author's

Table 1. Variables

Description of variable	Variable
Horizontal position of cart	x
Horizontal position of pendulum	x_p
Mass of pendulum	m
Mass of cart	M
Force on cart	F_{cart}
Length of pendulum	L
Angle of pendulum	θ
Translational energy	T_{trans}
Rotational energy	T_{rot}
Total Energy	T_{tot}
Horizontal reaction force	H
Vertical reaction force	V
Pendulum Inertia	I_1
Center of mass for pendulum	COM
Ball Inertia	I_2
Radius of virtual wheel	r_1
Radius of ball	r_2
Mass of virtual wheel	m_1
Mass of ball	m_2
Angular velocity of virtual wheel	ω_1
Angular velocity of ball	ω_2

report and is shown in Equation (1). Here, α is the angle of the wheel relative to the horizontal plane and β is the angle between the wheels, offset by 90° .

$$\begin{bmatrix} \frac{2 \cos(\beta)}{3 \cos(\alpha)} & \frac{2 \sin(\beta)}{3 \cos(\alpha)} \\ -\frac{\cos(\beta) - \sqrt{3} \sin(\beta)}{3 \cos(\alpha)} & -\frac{\sin(\beta) + \sqrt{3} \cos(\beta)}{3 \cos(\alpha)} \\ 0 & -\frac{\sin(\beta) - \sqrt{3} \cos(\beta)}{3 \cos(\alpha)} \end{bmatrix} \quad (1)$$

2.3 Mapping virtual angular acceleration to cart force

As stated earlier, the classical cart model uses a force applied to the cart as the control signal. In our system, we will instead be able to control the angular acceleration of the virtual drive wheels. Therefore, to apply this model to our system, a mapping between the angular acceleration of the virtual wheel and this force has been formulated. Below follows the derivation.

Using the formula for total kinetic energy for the cart

$$T_{tot} = T_{trans} + T_{rot}. \quad (2)$$

Together with the knowledge that the virtual wheel and the basketball are moving at the same velocity at the contact point

$$\begin{cases} v_1 = \omega_1 r_1 \\ v_2 = \omega_2 r_2 \end{cases} \implies \omega_2 = \omega_1 \frac{r_1}{r_2}. \quad (3)$$

We get

$$\frac{d}{dt} (\omega_1^2) = 2\omega_1 \dot{\omega}_1. \quad (4)$$

We can derive an expression for the force of the cart

$$\begin{aligned} T_{tot} &= \frac{m_2 v_2^2}{2} + \frac{I_2 \omega_2^2}{2} = \\ &= \frac{1}{2} (m_2 r_2^2) (\omega_1)^2 \left(\frac{r_1}{r_2} \right)^2. \end{aligned}$$

And

$$I_2 = \frac{2m_2 r_2^2}{3}, \quad (5)$$

where I_2 is the moment of inertia for a spherical shell. We then take the derivative of the energy with respect to time

$$\frac{dT}{dt} = (m_2 r_2^2 + I_2) \left(\frac{r_1}{r_2} \right)^2 \omega_1 \dot{\omega}_1.$$

We know that

$$\begin{cases} W = F \cdot d \\ P = \frac{W}{t} \end{cases} \implies P = \frac{F \cdot d}{t} = F \cdot v. \quad (6)$$

We also know that the derivative of energy with respect to time is power, P , so

$$P = F \cdot v = F \cdot \omega_1 r_1 = \frac{dE}{dt}. \quad (7)$$

Therefore, the relationship between the cart force and the angular acceleration of the virtual wheel is

$$\begin{aligned} F_{cart} \cdot \omega_1 r_1 &= (m_2 r_2^2 + I_2) \left(\frac{r_1}{r_2} \right)^2 \omega_1 \dot{\omega}_1 \\ \implies F_{cart} &= \left(m_2 + \frac{I_2}{r_2^2} \right) \dot{\omega}_1 r_1. \end{aligned}$$

2.4 The state space model

To develop the two LQR controllers, we need to design a force-based 2D state space model of the cart system shown in Figure 2. This model could then be modified to describe our system using the cart force and angular acceleration relationship.

This section covers the model derivation using Newtonian equations of motion. We will write $\sin(\theta)$ and $\cos(\theta)$ as s_θ and c_θ , respectively. The notation follows Figure 2.

$$\rightarrow: M\ddot{x} = F_{cart} - H \quad (8)$$

$$\uparrow: N - V - M_g = 0 \quad (9)$$

$$\rightarrow: m\ddot{x}_p = H \quad (10)$$

$$\uparrow: m_p \ddot{y}_p = V - mg \quad (11)$$

$$\widehat{COM}: I_1 \ddot{\theta} = -vL \cdot s_\theta + HL \cdot c_\theta \quad (12)$$

$$x_p = x + L \cdot s_\theta \implies \dot{x}_p = \dot{x} + L \cdot c_\theta \cdot \dot{\theta} \quad (13)$$

$$\implies \boxed{\ddot{x}_p = \ddot{x} - L \cdot s_\theta \cdot \dot{\theta}^2 + L \cdot c_\theta \cdot \ddot{\theta}} \quad (14)$$

$$y_p = L \cdot c_\theta \implies \dot{y}_p = -L \cdot s_\theta \cdot \dot{\theta} \quad (15)$$

$$\implies \boxed{\ddot{y}_p = -L \cdot c_\theta \cdot \dot{\theta}^2 - L \cdot s_\theta \cdot \ddot{\theta}} \quad (16)$$

We can use the equations (8), (9), (10), (11) and (12) to find two differential equations that describe our 2D system.

In equations (8), (9), (10), (11) and (12), there are 5 unknown variables, $\ddot{x}, H, V, N, \ddot{\theta}$. Equation (14) in (10) and (16) in (11), and $I_1 = 0$ since we assume that all mass is concentrated at the top of the pendulum gives the following:

$$\begin{cases} m(\ddot{x} - L \cdot s_\theta \cdot \dot{\theta}^2 + L \cdot c_\theta \cdot \ddot{\theta}) = H, \end{cases} \quad (17)$$

$$\begin{cases} m(-L \cdot c_\theta \cdot \dot{\theta}^2 - L \cdot s_\theta \cdot \ddot{\theta}) = -mg + V, \end{cases} \quad (18)$$

$$\begin{cases} 0 = -VL \cdot s_\theta + HL \cdot c_\theta, \end{cases} \quad (19)$$

$$\implies H \cdot c_\theta = V \cdot s_\theta. \quad (20)$$

Equation (17) and (8) removes the unknown H from equation (17)

$$\implies \boxed{F_{cart} = (m + M)\ddot{x} + mL(c_\theta \cdot \ddot{\theta} - s_\theta \cdot \dot{\theta}^2)}. \quad (21)$$

Dividing Equations (17) and (18) gives

$$\frac{H}{V} = \frac{\ddot{x} - L \cdot s_\theta \cdot \dot{\theta}^2 + L \cdot c_\theta \cdot \ddot{\theta}}{-L \cdot c_\theta \cdot \dot{\theta}^2 - L \cdot s_\theta \cdot \ddot{\theta} + g}. \quad (22)$$

Combining with Equation (20) we get

$$c_\theta \left(\ddot{x} - L \cdot s_\theta \cdot \dot{\theta}^2 + L \cdot c_\theta \cdot \ddot{\theta} \right) \quad (23)$$

$$= s_\theta \left(-L \cdot c_\theta \cdot \dot{\theta}^2 - L \cdot s_\theta \cdot \ddot{\theta} + g \right) \quad (24)$$

$$\Leftrightarrow \boxed{\ddot{x} \cdot c_\theta + L\ddot{\theta} - g \cdot s_\theta = 0}. \quad (25)$$

For small angles

$$\begin{cases} s_\theta \rightarrow \theta \\ c_\theta \rightarrow 1 \\ s_\theta \cdot \dot{\theta}^2 \rightarrow 0 \end{cases} \quad (26)$$

$$\implies \boxed{\begin{cases} M\ddot{x} = F_{cart} - mg\theta \\ ML\ddot{\theta} = (m + M)g\theta - F_{cart}. \end{cases}} \quad (27)$$

Solving for $\ddot{x}$ and $\ddot{\theta}$ gives

$$\begin{cases} \ddot{x} = \frac{F_{cart}}{M} - \frac{mg}{M}\theta \\ \ddot{\theta} = \frac{m+M}{ML} \cdot g\theta - \frac{F_{cart}}{mL}. \end{cases} \quad (28)$$

The system state $\vec{X} \in \mathbb{R}^4$, describing the state of the cart, is defined as

$$def : x_1 = \theta, x_2 = \dot{\theta}, x_3 = x, x_4 = \dot{x}. \quad (29)$$

Using this, we can define the system matrices A and B , and the full state space representation of the cart system is

$$\dot{\vec{X}} = A_{cart} \cdot \vec{X} + B_{cart} \cdot F_{cart} \implies \quad (30)$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \\ \dot{x}_3 \\ \dot{x}_4 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ \frac{m+M}{ML}g & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -\frac{mg}{M} & 0 & 0 & 0 \end{bmatrix} \vec{X} + \begin{bmatrix} 0 \\ -\frac{1}{ML} \\ 0 \\ \frac{1}{M} \end{bmatrix} F_{cart}. \quad (31)$$

By applying the results given in section 2.3, we can transform this model to more accurately match our system. The A matrix will remain the same, but the B matrix needs adjustments as seen below to account for the transformation between horizontal applied force and applied angular acceleration on the virtual wheels

$$A_{sys} = A_{cart}, \quad (32)$$

$$B_{sys} = B_{cart} \cdot \frac{1}{\left(m_2 + \frac{I_2}{r_2^2}\right) \cdot r_1}. \quad (33)$$

3. System design

3.1 Controller design

Starting from the 4-state cart-pendulum model in the previous Section 2.4, the derived continuous-time A and B matrices were imported into MATLAB and discretized with a sampling period of 4 ms using the MATLAB command `c2d` that results in a ZOH-model. The discrete LQR gain was then obtained with the Matlab command `dlqr`. A weighting on the input ($R = 0.001$) was retained, while the state-weighting matrix Q was tweaked to mostly emphasize the angular states θ and $\dot{\theta}$ as can be seen in Equation (36). LQR applies the state-feedback law:

$$u = -Kx, \quad (34)$$

which minimizes the quadratic cost

$$J = \sum_{k=0}^{\infty} \left(x[k]^T Q x[k] + u[k]^T R u[k] \right). \quad (35)$$

Large values in Q penalize deviations of the chosen states, while a large R discourages excessive control effort. Solving the discrete Riccati equation yields the gain K , which places all closed-loop poles inside the unit circle and therefore stabilizes the system. The resulting K matrix can be seen below, along with the Q matrix that yields the result.

$$Q = \begin{bmatrix} 1000 & 0 & 0 & 0 \\ 0 & 10 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (36)$$

$$K = [39270 \quad 3927 \quad 1106 \quad 3141.6]. \quad (37)$$

When implementing this in the real system, however, it was concluded that these values needed to be tweaked extensively due to instability in the system. The K that was used in the end was derived through extensive trial-and-error and ended up as

$$K = [26000 \quad 2600 \quad 0 \quad 0]. \quad (38)$$

By weighting only the angular states, this reduced the regulator to a pure PD-controller. This controller successfully balances the Ball-E.

3.2 The system states

As detailed in section 2.4, our model contains four state variables: Tilt θ and position x together with their time derivatives $\dot{\theta}$ and $\dot{x}$. Below follows how these variables are approximated using the IMU sensors together with properties of the stepper motor.

3.2.1 Tilt Estimating tilt means estimating the pitch and roll of the robot; both the accelerometer and gyroscope can be used for this purpose. The accelerometer outputs a vector in $\mathbb{R}^3$ describing the acceleration applied to the sensor measured in m/s^2 . Through trigonometric relationships, one can use this information to estimate the direction of gravity and thereby tilt. The gyroscope instead outputs a vector in $\mathbb{R}^3$ describing the angular velocity applied to the sensor, measured in rad/s . Integrating this vector over time gives tilt, assuming that the robot was standing upright initially.

The two methods both have severe shortcomings, however. The accelerometer method relies upon the assumption that all forces acting on the sensor are gravitational. This does not hold when the robot is in motion, leading to incorrect tilt readings as soon as the robot starts losing its balance. The gyroscope method relies on integration and is thereby prone to integration errors. Small errors in the sensor readings will accumulate over time and grow indefinitely.

To counteract these shortcomings, a complementary filter is used. In [4], the filter is described as a way to estimate one variable (tilt in this case) by combining two measurement sources (accelerometer and gyroscope tilt readings) and thus achieving better accuracy. The approach works since the accelerometer is susceptible to high-frequency noise and the gyroscope is susceptible to low-frequency noise. In practice, the filter is implemented using the following equation:

$$\theta(t+1) = \gamma \cdot \theta_{acc} + (1 - \gamma) \cdot [\theta(t) + dt \cdot \dot{\theta}_{gyro}], \quad (39)$$

where $\theta(t)$ is the estimated angle in iteration t , γ determines the influence of the two sources, θ_{acc} the accelerometer source's tilt estimate, dt the time passed since last iteration, and $\dot{\theta}_{gyro}$ the raw gyroscope readings. Through heuristic tuning, $\gamma = 0.999$ has been determined to work sufficiently well for our project, ensuring sufficient filtering.

3.2.2 Position To approximate the robot's position, the discrete and deterministic steps of the stepper motor are taken advantage of. The microcontroller has full control over the number of steps the motors should take, and each step corresponds to an exact amount of angular rotation. This means that we can, with great certainty, determine the angular velocity of the stepper motors. Given the angular velocities for the three stepper motors, one can determine the robot's linear velocity by using the inverse of the transformation matrix described in section 2.2. The robot's position can then be computed by integrating the linear velocity.

Without filters, the approach results in large integration errors. Aspects such as small inconsistencies in step length or the stepper motor wheels temporarily losing grip on the basketball will worsen our position estimate over time.

To circumvent this issue, the following calculation is used instead of pure integration:

$$x(t+1) = \lambda \cdot x(t) + dt \cdot \dot{x}(t), \quad (40)$$

where $x(t)$ and $\dot{x}(t)$ is the approximated position and linear velocity respectively. The constant $\lambda \approx 1$ represents a damping of the previous estimate. The idea bears some resemblance to that of the complementary filter used in our tilt estimation; the low-frequency error from integration is mitigated through damping. In this case, however, information is lost by using this approach. If the robot moved to some coordinate (a, b) and then stopped there indefinitely one would still have $\lim_{t \rightarrow \infty} x(t) = (0, 0)$. We found that this limitation does not matter, as we are not interested in the ball's absolute position. The purpose of the position state is to counteract changes in the ball's position, slowing down its motion, and the relative position is sufficient for this.

3.3 Software implementation

Here, we will provide an overview of how our analytically derived LQR controller was realized in practice using an Arduino microcontroller. There are a few key aspects to be discussed. Firstly, applying acceleration control signals to the stepper motors. Making sure that the motors are stepped such that the robot moves in the desired direction with the desired acceleration. Secondly, how all the systems are used together in the control loop to balance the robot.

3.3.1 Stepper motor driver The developed driver has two key functionalities. Firstly, the driver can step the three stepper motors such that the robot moves following a given target angular acceleration for the two virtual wheels. This acceleration is denoted ω_1 , see table 1. Secondly, it keeps track of the angular velocity of the two virtual wheels ω_1 . Below follows a description of how this was implemented.

A stepper motor's movement is not continuous like that of a DC motor. This means that there is no inherent natural acceleration; the behaviour needs to be emulated in software. Hardware interrupts have been used for this purpose. Specifically, the `TimerOne` [6] library has been utilized. The library uses one of the hardware timers in the Arduino such that code can be scheduled to be executed in a specified period (1ms in our case). Two operations are performed in the interrupt code. Firstly, the angular velocities of the wheels are updated using $\omega(t+1) = \omega(t) + T_{int} \cdot \dot{\omega}(t)$, where $\omega(t)$ is the angular velocity in the last interrupt, T_{int} is the interrupt period, and $\dot{\omega}(t)$ is the target angular acceleration, specified by the controller. Angular velocities for the virtual (ω_1) and the real omni-wheel (ω_r) wheels are kept track of separately, where ω_r is derived from ω_1 using the transformation matrix described in Section 2.2. Secondly, ω_r is used to determine if the stepper motors should be stepped in this interrupt. The number of interrupts since the stepper motors were last stepped n_{idle} is being tracked for this purpose. If $|\omega_r| \cdot T_{int} \cdot n_{idle} > 1$, it is determined that a step is due, and the corresponding motor driver's step pin is toggled on and off to trigger a step.

3.3.2 LQR The LQR controller runs in parallel to the stepper motor driver using the standard Arduino `loop()` function [1]. The code inside the `loop()` function will be executed as

fast as possible and loop indefinitely, stopping only to handle hardware interrupts. To set the control signal's desired update period, the Arduino `millis()` function [2] is used to keep track of time since last update and determine if a new update should occur. Control signal updates consist of the following operations:

- (1) Using the techniques described in Section 3.2, compute the system state $X(t)$.
- (2) Calculate the control signal using (34). Matrix operations are performed using the Arduino `BLA` linear algebra library [5].
- (3) Apply the control signal to the stepper motors through the custom virtual wheel-based stepper driver.

4. Hardware

This section details the hardware part of the project. Importantly, most of the hardware was designed and assembled by previous groups who worked on the same project. However, several crucial improvements were made during the project and this will also be covered here.

A detailed description of each hardware component is compiled in table 2. Notably, the stepper motors are not connected to the Arduino directly but rather controlled by the A4988 drivers. Using these drivers allows us to change the otherwise static step length of the motors through something called micro-stepping. The microsteps can be configured such that the motor steps in 1/1-, 1/2-, 1/4-, 1/8-, or 1/16-step increments of a full step. The trade-off for a shorter step length is a big loss of torque, especially at higher angular velocities.

Table 2. The hardware components

Name	Amount
Stepper motors, Nema 17, 0.4Nm	3
Arduino Due	1
Omni-wheels, 58mm, dual rollers	3
A4988 drivers	3
LSM9DS1 MEMS IMU	1
HM-10 BLE bluetooth Module	1
LiPo-battery, 3-cells, 2200mAh	1

4.1 Improvements made

There were some issues with the hardware at the start of the course. Firstly, one of the motor axles was wobbling. Secondly, the wheels were not tangent to the ball — causing them not to roll as intended. Thirdly, the microsteps were set too short, making the motors weak and unable to reliably control the robot.

The following was worked on to fix these issues: The wobbling motor axis was corrected with assistance from the university's control department. The wheels needed to be moved outward by around 11 mm to become tangent with the ball at a 45-degree angle. This was solved by 3D-printing brackets and placing them between the motors and their previous mounting points. The micro-stepping setting was initially at 1/16th,

resulting in only a fraction of the torque being available, but achieving smoother motor motion. After testing and research, the setting is now at 1/4th to try and minimize vibrations from larger steps while still preserving some torque.

5. Results

In this section, the performance of the robot will be described. The data will include empirical observations together with system state measurements made during balancing.

Three performance metrics have been decided upon. Firstly, when placing the robot upright on a basketball, how long does it take before it falls over? Secondly, how much does the robot oscillate when balancing? Thirdly, can the robot keep its balance even if subjected to disturbances? Ideally, we want the robot to be able to balance indefinitely and for it to stand completely still when no external forces are applied. As mentioned in Section 3.1, the k -matrix that was derived from the system model had to be tweaked extensively to achieve balance. From now on, these two k -matrices will be denoted K_m (model) and K_t (tweaked). The performance for the two k -matrices is subsequently described.

K_m : By manually holding the robot in an upright position and applying small tilts in all directions, it could be determined that the controller applied accelerations following our intuitive understanding of how the ball should be balanced. The robot was consistently controlled such that it moved towards the direction from which it was about to fall and thus straightened itself up. However, if we ever let go of the robot completely, let it balance on its own, it would fall over instantly because of overshoot.

K_t : Using this k -matrix, proper balancing was achieved. In our most successful tests, the robot could stand upright for multiple minutes without human intervention. For this performance to be possible, however, it was noted that great care had to be taken to make sure that the robot was as straight as possible before activating the controller and letting go of the robot. Figure 3 shows the first ten seconds of the system state and control signal for a successful run. Since K_t does not use the position states, these are not plotted. It can be noted that the robot never swayed further than $0.02 \text{ rad} \approx 1.14^\circ$ off centre in either direction. In this test, no disturbance was applied to the robot. In other tests, we have concluded, however, that the robot can survive light pushes without losing balance.

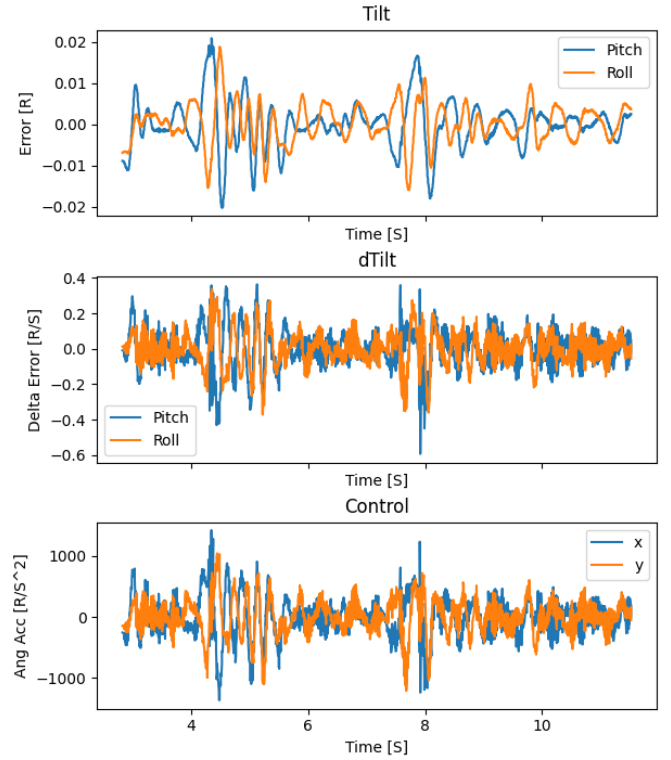


Figure 3. The first ten seconds of system states and corresponding control signals when the robot successfully balanced. R and S stand for radians and seconds, respectively.

6. Discussion

From the results, we can conclude that our balancing goal is achieved. Considering that no group member had worked with a control system as complex as this project before, we are very pleased that we were able to get this far. Every design decision presented in this report was made through extensive trial-and-error testing, through identifying and finding solutions to numerous amounts of hardware and software obstacles. When we started working on the project, we did not expect the controller to be anywhere near as consistent and accurate as it ended up being. That the robot can recover when pushed or shoved is a characteristic that significantly surpassed our initial beliefs.

Despite the strong performance of K_t , a fundamental issue is exposed through the K_m results. We conclude that the reason for the poor performance of K_m most likely stems from the fact that the used cart model does not capture enough of the real system's kinematics. Nonetheless, achieving balance is, as the results show, possible without a model. But the lack of one will limit our ability to extend the capabilities of the controller. One extension that has been discussed is, for example, to be able to steer the robot with a remote control like an RC car. This requires controlling all four state variables, brute-forcing the k -matrix values like we did for K_t , and is therefore most likely not a reasonable approach.

6.1 Model issues

A radical simplification of the system's dynamics was needed for us to be able to derive a state-space model within the project timeline. As mentioned, we did not have much experience modeling systems of this complexity. We choose to model the system as a cart because this is a famous automatic control problem, meaning that there would be a lot of available modeling resources. Still, a significant amount of dynamics is lost as a result of this approach. One significant difference between the real system and the cart model is how the pendulum affects the position of the ball. According to the cart model, if the pendulum falls to the left, the cart would accelerate to the right. However, in the real system, the same scenario would give a leftward acceleration because the basketball is a part of the pendulum itself.

6.2 Possible State issues

Throughout the project's timeline, we have encountered many problems with estimating the system states. Errors in the software implementation have caused bugs that have been hard to identify and solve. A lot of times, these bugs have only exposed themselves in specific situations and scenarios, making the debugging process cumbersome. Because of this history of erroneous states, we do not find it unlikely that there still could be some hidden bug in the code causing issues for one of the state variables. The position approximation is the most likely candidate in this case. This code has been subjected to the least amount of testing out of all the state variables, and major changes have been made to the stepper motor driver since the initial implementation.

References

- [1] Arduino. *Loop()*. <https://docs.arduino.cc/language-reference/en/structure/sketch/loop/>. Accessed: 2025-04-28. 2024.
- [2] Arduino. *Millis()*. <https://docs.arduino.cc/language-reference/en/functions/time/millis/>. Accessed: 2025-04-28. 2024.
- [3] K. van der Blonk. *Modeling and Control of a Ball-Balancing Robot*. MA thesis. University of Twente, 2014.
- [4] S. Park and J. How. *Examples of estimation filters from recent aircraft projects at mit*. https://ocw.mit.edu/courses/16-333-aircraft-stability-and-control-fall-2004/9bc94307e01f85dbdfce05e87851361a_lecture_15.pdf. 2004.
- [5] T. Stewart. *Basic linear algebra*. <https://github.com/tomstewart89/BasicLinearAlgebra>. Accessed: 2025-04-28. 2024.
- [6] P. Stoffregen. *Timerone: arduino timer library*. <https://github.com/PaulStoffregen/TimerOne>. Accessed: 2025-04-27. 2023.
- [7] Wikipedia. *Inverted pendulum*. https://en.wikipedia.org/wiki/Inverted_pendulum. Accessed: 2025-05-05.

Making Crazyflie Crazy

Wang Tianci¹ Cai Yaliang² Zu Haobo³ Vahid Faraji⁴ Hao Changsheng⁵

¹ti4050wa-s@student.lu.se ²ya8258ca-s@student.lu.se ³ha2218zu-s@student.lu.se
⁴va6321fa-s@student.lu.se ⁵ch1123ha-s@student.lu.se

Abstract: This study investigates the implementation and evaluation of TinyMPC—a lightweight model predictive control (MPC) solver optimized for resource-constrained microcontrollers—on the Crazyflie 2.1 nano-quadrotor. The primary objective is to achieve high-speed, constraint-aware control despite severely limited onboard computation. Two demanding tasks are addressed: rapid stabilization from extreme attitude deviations and aggressive trajectory tracking under stringent motor limitations. Both tasks are formulated as finite-horizon optimal control problems, and TinyMPC is deployed via a custom out-of-tree firmware module. The overall architecture combines offline trajectory optimization in Julia using the JuMP framework, onboard state estimation from Lighthouse positioning, and a real-time C++ MPC kernel running at 500 Hz. Control is executed through a receding-horizon approach based on linearized six-degree-of-freedom dynamics, quaternion attitude representation, and Runge–Kutta discretization. Ongoing experiments evaluate recovery performance from initial tilt angles up to 90°, assessing robustness under highly non-nominal conditions. The results demonstrate the feasibility of onboard MPC for autonomous flight in miniature aerial robots operating under extreme hardware constraints.

1. Introduction

MPC constitutes a prevalent framework in contemporary robotics, owing to its capacity to enforce actuator and state constraints, predict future system behavior, and facilitate dynamic, feedback-based control. Its utility becomes especially pronounced in aggressive maneuvers and safety-critical operations, where strict adherence to physical limits is imperative. Despite these theoretical and practical advantages on high-performance platforms, real-time MPC has remained largely inaccessible to small-scale aerial vehicles constrained by limited onboard computational resources [1, 2].

Nano-quadrotors such as the Crazyflie 2.1 [3] offer compelling applications—including indoor inspection, swarm deployment, and agile navigation within confined environments—yet their minimal processing power, memory footprint, and energy budget pose significant barriers to the implementation of optimization-based control. Conventional MPC solvers (e.g., OSQP, CVXGEN) typically demand more RAM and computation time than available on microcontroller units, relegating control to offboard processors or simplistic schemes such as PID loops.

To address these limitations, TinyMPC—a lightweight, ADMM-based convex MPC solver tailored for microcontrollers—has been employed. By exploiting problem structure, precomputing Riccati-style updates, and removing matrix inversions during online execution, TinyMPC achieves real-time, constraint-aware control directly on the Crazyflie 2.1 without compromising performance [4, 5, 6]. The software architecture couples offline trajectory generation (via Julia and the JuMP framework) with quaternion-based orientation tracking and a 500 Hz MPC loop integrated into the firmware as an out-of-tree controller module [7].

Designed to satisfy stringent timing requirements (2 ms

per control step), TinyMPC is presently under experimental evaluation for dynamic path tracking at speeds up to 1.5 m/s. Concurrent studies examine recovery from 90° attitude deviations to quantify controller robustness under extreme initial conditions. Collectively, these investigations confirm the viability of deploying fully onboard, real-time MPC on miniature aerial robots with severe hardware constraints [3, 7].

Figure 1, The Crazyflie 2.1 nano-quadrotor used in this project. It features a lightweight carbon frame, and four coreless brushed motors with plastic propellers. The platform also supports various expansion decks, such as the Lighthouse positioning system.



Figure 1. The Crazyflie 2.1 nano-quadrotor used in this project

2. Problem Formulation

This section formulates the control problem addressed in this project, encompassing both the stabilization and trajectory tracking tasks for the Crazyflie 2.1 quadrotor. The control objective in both tasks is to compute an optimal sequence of control inputs over a finite prediction horizon, subject to system dynamics and input/state constraints.

2.1 System Model and Constraints

Let the quadrotor state at time step k be denoted by

$$x_k = (p_k, v_k, q_k, \omega_k),$$

where $p_k \in \mathbb{R}^3$ is the position, $v_k \in \mathbb{R}^3$ is the linear velocity, $q_k \in \mathbb{H}$ is the unit quaternion representing orientation, and $\omega_k \in \mathbb{R}^3$ is the angular velocity.

The control input at time step k is:

$$u_k = (u_{1,k}, u_{2,k}, u_{3,k}, u_{4,k}) \in \mathbb{R}^4$$

representing the individual thrust commands sent to the four motors.

The system dynamics are written abstractly as:

$$x_{k+1} = f(x_k, u_k)$$

where f represents the discrete-time evolution of the system.

At each time step, control inputs are subject to actuator limits:

$$u_{\min} \leq u_{i,k} \leq u_{\max}, \quad \text{for } i = 1, \dots, 4$$

Additional state constraints (e.g., minimum altitude, maximum tilt) can be imposed as needed, and quaternion normalization is preserved implicitly through the system dynamics.

2.2 Stabilization Task

The stabilization task requires the quadrotor to recover from an initial attitude disturbance and return to a steady hover. The target is a fixed reference state:

$$x_{\text{ref}} = (p_{\text{ref}}, 0, q_{\text{ref}}, 0)$$

where $p_{\text{ref}} \in \mathbb{R}^3$ is the desired hover position and q_{ref} is the quaternion corresponding to level upright orientation. Both linear and angular velocities are zero at the reference.

To achieve this, the controller minimizes the following stage cost at each time step:

$$\ell(x_k, u_k) = \|p_k - p_{\text{ref}}\|_{Q_p}^2 + w_q \theta_k^2 + \|u_k - u_{\text{hover}}\|_R^2 \quad (1)$$

where: $Q_p \succ 0$ is the position error weight matrix, w_q is a scalar weight on orientation error, $R \succ 0$ is the input cost matrix, u_{hover} is the nominal input that balances gravity, θ_k is the angular distance between q_k and q_{ref} , which can be computed from the relative quaternion rotation.

Here, the notation $\|x\|_Q^2$ denotes the weighted quadratic form $x^\top Q x$, where $Q \succ 0$ is a symmetric positive definite matrix. This generalizes the squared Euclidean norm and allows direction-dependent penalization.

2.3 Trajectory Tracking Task

In the trajectory tracking task, the quadrotor is required to follow a predefined time-varying reference over a finite horizon. The reference state varies over time and defines a desired flight path:

$$\{x_{\text{ref},k}\}_{k=0}^N = \{(p_{\text{ref},k}, q_{\text{ref},k}, v_{\text{ref},k}, \omega_{\text{ref},k})\}$$

and the reference inputs denote the desired motor thrust command at time step k , typically provided by an offline optimizer or a high-level planner:

$$\{u_{\text{ref},k}\}_{k=0}^{N-1} = \{(u_{1,\text{ref},k}, u_{2,\text{ref},k}, u_{3,\text{ref},k}, u_{4,\text{ref},k})\}$$

To achieve accurate tracking, the controller minimizes the following cost function:

$$J = \sum_{k=0}^{N-1} \left[\|p_k - p_{\text{ref},k}\|_{Q_p}^2 + w_q \theta_k^2 + \|v_k - v_{\text{ref},k}\|_{Q_v}^2 + \|u_k - u_{\text{ref},k}\|_R^2 \right] + \|p_N - p_{\text{ref},N}\|_{Q_f}^2 \quad (2)$$

where θ_k denotes the rotation angle between the current orientation q_k and the reference $q_{\text{ref},k}$. The matrices Q_p , Q_v , R , and Q_f are positive definite and tune the relative importance of position accuracy, velocity tracking, input smoothness, and terminal error minimization, respectively. The formulation ensures that the system tracks general three-dimensional trajectories with constraint-aware and smooth control behavior.

3. System Setup

To enable real-time MPC on a resource-constrained nano-quadcopter, the system is implemented as a modular architecture comprising onboard control, offboard trajectory generation, and a host-side interface. This section outlines the hardware platform, software stack, and trajectory generation pipeline.

3.1 Hardware

The experiments are conducted on the Crazyflie 2.1 quadrotor [8], an open-source nano aerial vehicle equipped with an STM32F405 ARM Cortex-M4 microcontroller. For indoor positioning, we employ the Lighthouse deck. Four Lighthouse base stations are installed in the test laboratory, enabling sub-millimeter pose estimation through infrared time-of-flight measurements. This setup provides the high-accuracy localization required for aggressive control maneuvers.

3.2 Software

The software system spans three layers: the offboard host interface, the onboard real-time controller, and the offline trajectory generator.

Host-Side Control Interface The host PC communicates with Crazyflie using the Crazyradio PA dongle and the official crazyflie-lib-python [9] library. A high-level control script written in Python handles initialization, mode switching, and motion command. Specifically, the MPC controller is activated by setting the parameter `stabilizer.controller` to a custom-defined value corresponding to our Out-of-Tree MPC module.

This interface allows loading of optimized reference trajectories, sending hover and landing commands, and logging onboard telemetry for analysis. It plays a supervisory role and is not involved in closed-loop control.

Onboard MPC Controller Figure 2 illustrates the end-to-end control loop. The onboard controller is implemented in C++ within the Crazyflie firmware, utilizing the TinyMPC library. This controller operates at a frequency of 100 Hz, computing control inputs in real-time based on the current state of the quadrotor.

State estimation is performed using the Crazyflie's built-in Kalman filter, which fuses data from the IMU and the

Lighthouse positioning system to provide accurate position and orientation estimates.

The control algorithm linearizes the quadrotor dynamics around the hover condition, resulting in a LTI model. This model is used to formulate a quadratic programming (QP) problem, which the TinyMPC solver addresses using the alternating direction method of multipliers (ADMM). The solver computes a sequence of control inputs over a finite horizon, but only the first input in the sequence is applied to the quadrotor. This process repeats at each control step, enabling real-time trajectory tracking.

The controller also incorporates constraints on the control inputs to ensure they remain within feasible bounds, enhancing the safety and reliability of the flight.

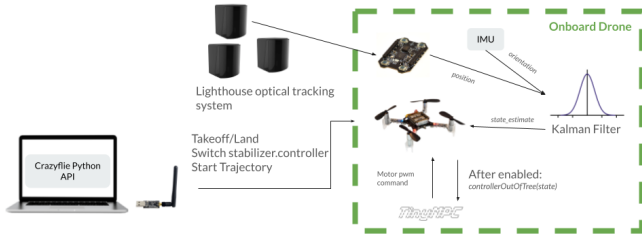


Figure 2. System architecture

Offline Trajectory Generation Reference trajectories are generated offline using the Julia-based JuMP optimization framework. The quadrotor dynamics are linearized around hover, and the resulting linear system is used to solve a finite-horizon quadratic optimal control problem. Cost terms include position and attitude tracking errors, velocity deviation, and control effort, with an optional terminal cost to enforce convergence.

The trajectory optimization process outputs a discrete-time sequence of reference states and control inputs, which are exported to a C header file and embedded in the firmware. This enables fully onboard trajectory tracking without runtime external dependencies.

Figure 3 illustrates the full control architecture, showing data flow from offline planning to onboard execution.

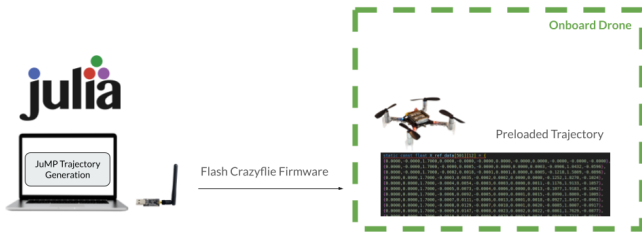


Figure 3. Offline trajectory generation and firmware integration.

4. Controller Implementation

This section outlines the implementation of a real-time MPC for the Crazyflie 2.1 quadrotor, using the open-source TinyMPC framework [10] [11]. The controller is deployed on a resource-constrained embedded platform and is capable of

tracking dynamic reference trajectories while satisfying physical constraints on input and state. The control objectives and optimization criteria used in this section follow the problem definitions outlined in Section 2.

4.1 Implementation Context and Objectives

The controller is implemented based on the TinyMPC library, which is optimized for embedded execution on micro-controllers. Our goal is to deploy this controller within the Crazyflie firmware, tune its parameters, and integrate it with the reference trajectory generator and onboard state estimation modules.

4.2 Reference Integration and Runtime Access

The predictive controller uses the linear system model and cost formulation defined in Section 2.1 System Model and Constraints. The reference trajectories, including both the state sequence $\{x_{\text{ref},k}\}_{k=0}^N$ and the control input sequence $\{u_{\text{ref},k}\}_{k=0}^{N-1}$, are generated offline using a direct transcription method and exported to the Crazyflie firmware as C header files.

These references are stored as statically defined arrays and accessed via pointer dereferencing during runtime. This structure allows tight memory control and deterministic execution time, which is critical for real-time onboard optimization.

4.3 QP Structure and ADMM

The controller solves a quadratic program (QP) of the form:

$$\min_{\Delta u_0, \dots, \Delta u_{N-1}} \sum_{k=0}^{N-1} \|x_k - x_{\text{ref},k}\|_Q^2 + \|\Delta u_k\|_R^2 + \|x_N - x_{\text{ref},N}\|_{Q_f}^2$$

subject to the linear dynamics:

$$x_{k+1} = Ax_k + B\Delta u_k + c \quad \text{and} \quad u_k = \Delta u_k + u_{\text{ref},k}$$

To solve this QP problem efficiently on embedded hardware, the alternating direction method of multipliers (ADMM) is employed. ADMM is well suited to problems where the cost function is separable from the constraints, allowing decomposition into simpler subproblems that are solved iteratively.

The problem is reformulated by introducing slack variables z_k and w_k to decouple the state and input constraints from the dynamics:

$$x_k = z_k, \quad u_k = w_k,$$

with $z_k \in X$ and $w_k \in U$. The associated augmented Lagrangian is

$$\begin{aligned} \mathcal{L}(x, u, z, w, y, g) = & J(x, u) + \sum_{k=1}^N I_X(z_k) + \sum_{k=1}^{N-1} I_U(w_k) \\ & + \sum_{k=1}^N \left[y_k^\top (x_k - z_k) + \frac{\rho}{2} \|x_k - z_k\|^2 \right] \\ & + \sum_{k=1}^{N-1} \left[g_k^\top (u_k - w_k) + \frac{\rho}{2} \|u_k - w_k\|^2 \right], \end{aligned} \quad (3)$$

where y_k and g_k are scaled dual variables, and $\rho > 0$ is a penalty parameter. The indicator functions $I_X(\cdot)$ and $I_U(\cdot)$

equal zero when the argument is in the set, and infinity otherwise.

Each ADMM iteration proceeds in three steps:

1. **Primal update:** Minimize $\mathcal{L}$ with respect to (x^+, u^+) while holding (z, w, y, g) fixed. This leads to a quadratic program with linear dynamics and modified cost terms that is equivalent to a linear-quadratic regulator (LQR) problem with shifted linear terms due to dual contributions.

2. **Slack update:** Perform projection onto the constraint sets:

$$z_k^+ \leftarrow \text{proj}_X(x_k^+ + y_k), \quad w_k^+ \leftarrow \text{proj}_U(u_k^+ + g_k).$$

3. **Dual update:** Update Lagrange multipliers using residuals:

$$y_k^+ \leftarrow y_k + x_k^+ - z_k^+, \quad g_k^+ \leftarrow g_k + u_k^+ - w_k^+.$$

This iterative procedure exploits the structure of the MPC problem, separating the unconstrained dynamics optimization from the constraint handling, and enables efficient implementation on resource-limited platforms through precomputations and Riccati-based updates.

The solver operates at 100 Hz and uses warm-starting from the previous solution to accelerate convergence. Reference values and matrices are preloaded during firmware compilation to minimize memory allocation during execution.

4.4 Combining and Pre-Computing LQR

To solve a modified LQR problem in primal update, an infinite-horizon solutions K_∞, P_∞ is computed offline using the discrete-time Riccati equation. Define

$$C_1 = (R + B^\top P_\infty B)^{-1}, \quad C_2 = (A - BK_\infty)^\top.$$

Then

$$d_k = C_1(B^\top p_{k+1} + r_k), \quad p_k = q_k + C_2 p_{k+1} - K_\infty^\top r_k.$$

Precompute for discrete ρ values and adapt online based on residuals. This solution is applied at 500 Hz, independently of the TinyMPC loop.

The final control law is:

$$u = \Delta u_0 - K_\infty(x - x_{\text{ref},1}) + u_{\text{ref},0}$$

where Δu_0 is the feedforward control increment from the MPC solution, x is the current state estimate, $x_{\text{ref},1}$ is the one-step predicted reference state.

This dual-rate structure allows TinyMPC to handle global planning over the horizon, while the LQR layer compensates for local disturbances and model inaccuracies at a higher rate. The adopted controller leverages a lightweight QP solver and a dual rate feedback architecture to enable accurate, constraint-aware trajectory tracking onboard a highly resource-limited platform. The controller parameters are configured, incorporated into the embedded control stack, and the performance are evaluated in both simulation and physical experiments.

This implementation exploits problem structure to achieve real-time execution on resource-constrained hardware.

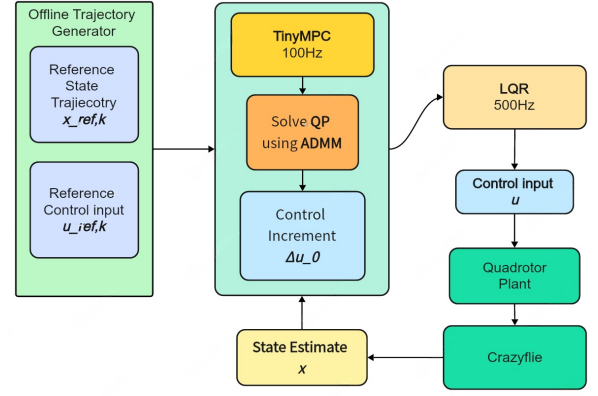
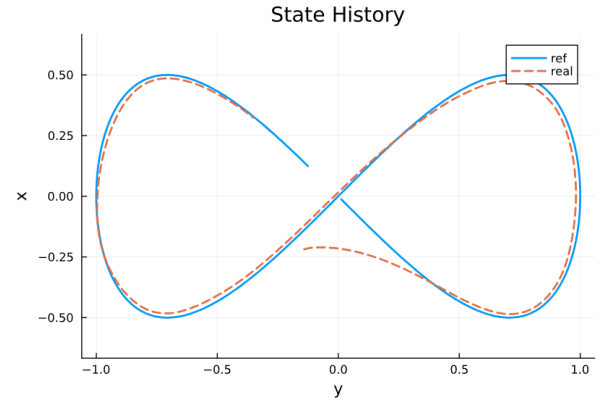


Figure 4. Controller system architecture

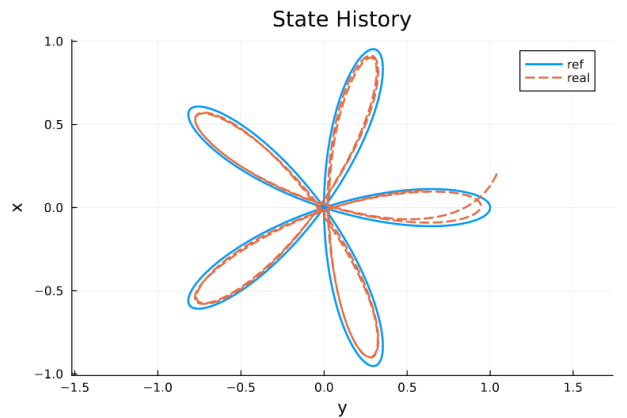
5. Experiment

The TinyMPC controller is evaluated through two experiments designed to assess its trajectory tracking performance under high-dynamic conditions and constrained control inputs.

5.1 Evaluation of Simulation



(a) Figure-eight trajectory



(b) Petal trajectory

Figure 5. Quadrotor tracking in simulation.

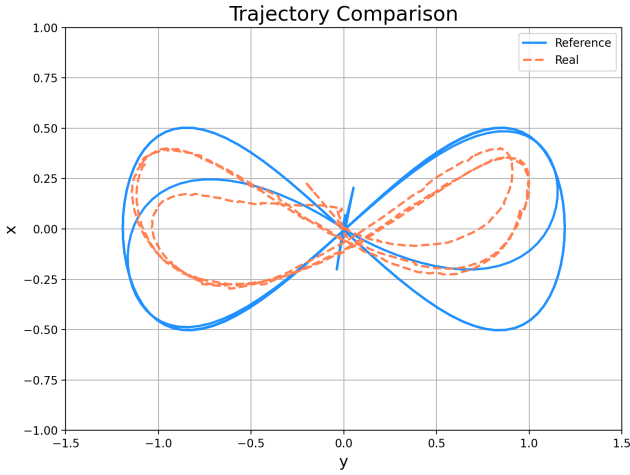
In this section, the tracking performance is evaluated through simulation by utilizing the offline-generated reference trajectory from TinyMPC, in conjunction with a time-invariant Lin-

ear Quadratic Regulator (LQR) controller operating in real-time.

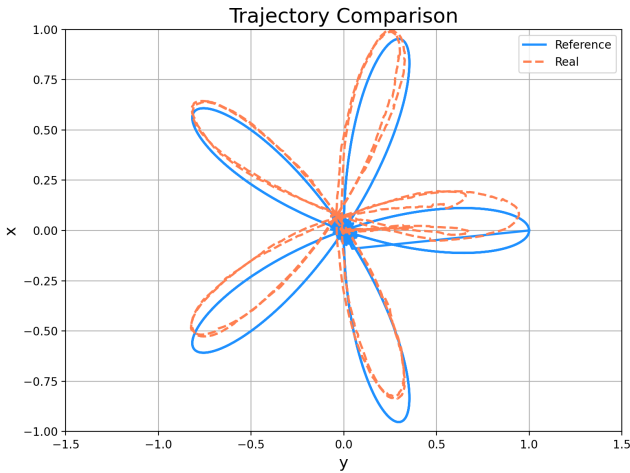
Closed-loop behavior is simulated for $N = 100$ steps using a fourth-order Runge–Kutta integrator with a fixed step size of $\Delta t = 0.01$ s.

As depicted in Figure 5, the maximum position tracking error remains below 0.05 m across both trajectories, confirming sub-decimeter accuracy with millisecond-scale computation per step.

5.2 Evaluation of Real-Time



(a) Figure-eight trajectory



(b) Petal trajectory

Figure 6. Quadrotor tracking in real-world test.

The trajectory tracking results demonstrate the effectiveness of the proposed MPC-based control scheme for agile quadrotor flight. Across both the flower-shaped and figure-eight reference paths, the Crazyflie was able to reproduce the global geometric structure and directionality of the trajectories with high fidelity. Notably, the controller maintained smooth and continuous flight even through complex maneuvers, without abrupt transitions or discontinuities. This behavior highlights a key advantage of MPC: the generation of naturally smooth control inputs due to horizon-based optimization.

The system operated in real time at 500 Hz using a prediction horizon of $N = 15$, which is a nontrivial constraint for onboard computation. Nonetheless, the controller exhibited robust performance throughout the flight, maintaining stability and avoiding divergence even in high-curvature regions and during trajectory crossings. These results validate the real-time feasibility of TinyMPC on resource-constrained hardware, as well as the robustness of the formulation under physical disturbances and modeling uncertainties.

Moreover, the controller generalizes well across different trajectory classes. While the flower trajectory tested the controller’s ability to handle multiple tight turns and symmetry, the figure-eight trajectory challenged the system with sharp transitions and crossing paths. In both cases, the MPC framework achieved reliable path adherence without compromising safety or stability. These observations confirm that the controller is not overfitted to a particular trajectory and is capable of general-purpose reference tracking for dynamic and geometrically complex flight tasks.

5.3 Evaluation—Extreme Initial Poses

To evaluate the robustness of TinyMPC under challenging initial conditions, an experiment is designed in which the quadrotor is initialized with extreme attitude errors, such as a 90° pitch or roll. This setup aims to assess the controller’s ability to stabilize the vehicle from highly non-nominal states while adhering to actuator constraints.

In the experiment, the Crazyflie quadrotor is manually positioned in a significantly tilted configuration and then released. The system is subsequently controlled solely by TinyMPC. The recovery behavior is monitored, including position error over time, convergence speed, and adherence to thrust limits throughout the maneuver.

6. Discussion

The experimental results demonstrate that TinyMPC offers a viable solution for implementing real-time, constraint-aware MPC on highly resource-constrained platforms, such as the Crazyflie 2.1. Despite the limited onboard computational resources, the system achieved high-frequency control at 500 Hz, reliable convergence within a strict 2 ms compute budget. This capability enabled the execution of aggressive flight maneuvers, including fast figure-eight rapid entirely onboard without dependence on reliance computation.

A key insight from the project is the critical importance of careful problem formulation and model discretization. The linearization of the 6-DOF quadrotor dynamics around a hover condition, combined with a quaternion-based attitude representation, enhanced numerical stability and robustness during large-angle rotations. The workflow, which includes offline trajectory generation in Julia and automated code export to embedded C++, facilitated an efficient and modular integration process.

The tuning of MPC weights and the length of the prediction horizon were found to significantly impact tracking accuracy and overall stability. While TinyMPC effectively managed control constraints, the limited prediction horizon—due to onboard memory limitations—occasionally hindered long-term

planning performance during complex maneuvers. Nevertheless, the controller’s rapid response and consistent adherence to constraints facilitated safe and precise flight.

It is also worth noting that some of the originally intended experiments—such as full stabilization from extreme initial states like a 90° tilt or a dynamic airborne release—were not successfully realized using TinyMPC alone. Although the original work proposed that TinyMPC could control the entire recovery process from such highly non-nominal conditions, our implementation did not reproduce this behavior. In both scenarios, the quadrotor failed to respond when TinyMPC was activated from these initial states—no thrust was generated and the system remained unresponsive.

To enable further testing, we adopted a practical workaround: the Crazyflie was first stabilized using its built-in PID controller to achieve a hover state, and TinyMPC was then engaged for trajectory tracking. This adjustment allowed the successful execution of figure-eight and petal-shaped trajectories, but deviated from the fully onboard recovery described in prior literature.

These findings underscore the feasibility of implementing advanced control algorithms, such as MPC on ultra-low-power microcontrollers. Future research will concentrate on optimizing the onboard solver and facilitating dynamic trajectory generation, aiming to enhance the autonomy and adaptability of nano-quadrotor systems.

References

- [1] S. M. Neuman, B. Plancher, B. P. Duisterhof, S. Krishnan, C. Banbury, M. Mazumder, S. Prakash, J. Jabour, A. Faust, and G. C. de Croon. “Tiny robot learning: challenges and directions for machine learning in resource-constrained robots”. In: *2022 IEEE 4th International Conference on Artificial Intelligence Circuits and Systems (AICAS)*. IEEE. 2022, pp. 296–299.
- [2] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein. “Distributed optimization and statistical learning via the alternating direction method of multipliers”. *Foundations and Trends in Machine Learning* **3**:1 (2011), pp. 1–122.
- [3] W. G. et al. *Crazyflie 2.0 quadrotor as a platform for research and education in robotics and control engineering*. Available: https://www.bitcraze.io/papers/giernacki_draft_crazyflie2.0.pdf. 2020.
- [4] *Controllers in the crazyflie — bitcraze*. Available: <https://www.bitcraze.io/documentation/repository/crazyflie-firmware/master/functional-areas/sensor-to-control/controllers/>.
- [5] G. Guennebaud and B. Jacob. *Eigen v3*. Available: <http://eigen.tuxfamily.org>. 2010.
- [6] *Teensy® 4.1*. Available: <https://www.pjrc.com/store/teensy41.html>. 2024.
- [7] D. Mellinger and V. Kumar. “Minimum snap trajectory generation and control for quadrotors”. In: *2011 IEEE International Conference on Robotics and Automation*. IEEE. 2011, pp. 2520–2525.
- [8] B. AB. *Crazyflie 2.1*. Product documentation. 2020. URL: <https://www.bitcraze.io/crazyflie-2-1/> (visited on 2025-05-19).
- [9] B. AB. *Crazyflie-lib-python: installation guide*. 2024. URL: <https://github.com/bitcraze/crazyflie-lib-python/blob/master/docs/installation/install.md> (visited on 2025-05-19).
- [10] RoboticExplorationLab. *Tinympc-crazyflie-firmware*. 2025. URL: <https://github.com/RoboticExplorationLab/tinympc-crazyflie-firmware> (visited on 2025-05-19).
- [11] K. Suresh. *Controller_tinympc.cpp*. 2024. URL: https://github.com/krish-suresh/magperch/blob/main/controller_tinympc.cpp (visited on 2025-05-19).



Jenga Tower

Axel Lagerstedt¹ Axel Sweger² Pontus Sjöstedt³ Sander Smådal⁴ Zhehao Chen⁵

¹ax3124la-s@student.lu.se ²ax3862sw-s@student.lu.se ³po7853sj-s@student.lu.se
⁴sa7808sm-s@student.lu.se ⁵zh3235ch-s@student.lu.se

Abstract: This report describes a robotic system designed to build a Jenga tower. It consists of two main parts: first a computer vision system which approximates the pose of each Jenga piece, and, then a robot arm system, which, is in charge of moving the robotic arm. The robot arm system is controlled through MoveIt with the use of a Python script. Path planning is done in Python script with hand-crafted Cartesian paths. In the implementation phase of the project, the computer vision and robot-arm system were integrated to work in tandem.

1. Introduction

The objective of this project is to construct a Jenga tower using a robot arm assisted by external sensors, in this case a camera. The camera identifies scattered blocks and provides coordinates for the robot which will then create a trajectory and follow it to pick them up and then set them down in a predetermined tower area.

The report will in parallel present the work from the computer vision part and the robot part. These parts were worked on independently and later on integrated to work together.

2. Modeling

No modeling was needed for this project. Since the robot was already configured and test on it could be ran in real-time. The necessity of modeling fell outside the scope of this project.

3. System Design

The goal of the system is to move Jenga pieces laying on the ground onto a tower. The process is explained in figure 1. This thought process will continue until all the blocks are placed in a designated tower location.

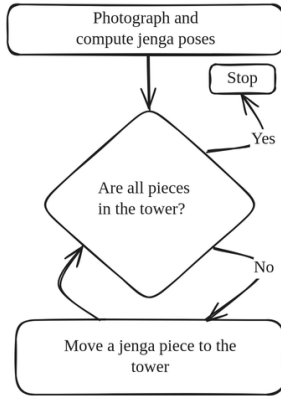


Figure 1. Flowchart for System

3.1 Computer Vision

The end goal of the computer-vision part is to obtain a 3D pose of the Jenga pieces. This is done through a few steps.

3.2 Camera calibration

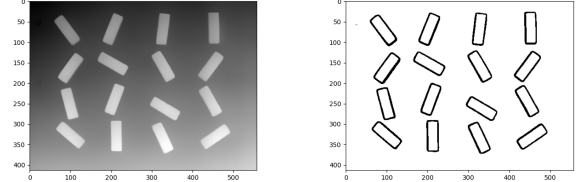
The raw image will be distorted causing straight lines in the real world to be curved in the image. Working with none linear distortions will make the work of retrieving the 3D coordinates harder. Therefore the camera was calibrated using a checkerboard. Using the obtained camera matrix all future images were "undistorted".

3.3 Obtaining the height map

DepthAnythingV2[2] was used to get a depth-map. This depth-map does not work well enough to approximate a usable distance to the camera, it does however create a strong contrast between the Jenga blocks and the table. This turned out to be very useful in segmenting the Jenga pieces.

3.4 Edge Detection for Detection and Segmentation

Starting out with a depth-map of the Jenga pieces seen in figure 2a a Sobel-filter was applied to the image. It works by taking the derivative of the image in the x - and y -axis. Afterwards the image was threshold, resulting in a binary image of where there are edges and where there is not. The result can be seen in figure 2b.



(a) Height map

(b) Edges

Figure 2. Height map and Edges

The edges were then contoured using `cv.findContours[5]` which returns a list of polygons i.e the contours and the hierarchies of the contours i.e which polygon is inside which. The contours are then filtered keeping only the contours without any children, contours with the correct ratio between width and height, and, the square contours.

Then each contour was fitted to a rectangle using '`cv.minAreaRect`' which yields the center of each rectangle and it's angle. The result can be seen in figure 3

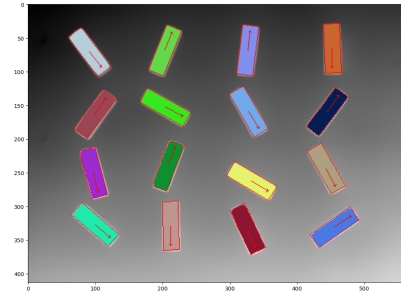


Figure 3. 2D pose estimation result

3.5 Eye hand calibration

The environment is constrained so that all Jenga pieces are laying in the same plane $z = \frac{\text{height(Jenga)}}{2}$ similar to what is seen in figure 3. This will make everything easier as occluded Jenga pieces will not have to be accounted for.

Assuming that the camera is infinitely far away and perpendicular to the Jenga pieces a similarity transform is assumed:

$$\begin{cases} \vec{x}_{1,image} \vec{s} + \vec{c} = \vec{x}_{1,robot} \\ \vec{x}_{2,image} \vec{s} + \vec{c} = \vec{x}_{2,robot} \end{cases} \iff \begin{cases} \vec{s} = \frac{\vec{x}_{1,robot} - \vec{x}_{2,robot}}{\vec{x}_{1,image} - \vec{x}_{2,image}} \\ \vec{c} = \vec{x}_{1,robot} - \vec{s} \vec{x}_{1,image} \end{cases}$$

where $\vec{s}$ is a scale factor, $\vec{c}$ is an offset, $\vec{x}_{i,image}$ are the calibration points in pixel-space, and, $\vec{x}_{i,robot}$ the calibration points in robot-space.

The pixel-space coordinates were obtained by first designing a penholder in Fusion 360 [3] which then was printed. This can be seen in figure 4. The holder has a spring underneath the pen pushing it outwards. This regulates how hard the pen is pushed against the drawing surface. The holder was then gripped by the robot and made to draw a square with corners in known locations for the robot. The corners could then be seen from the camera: giving the coordinates in pixel-space.

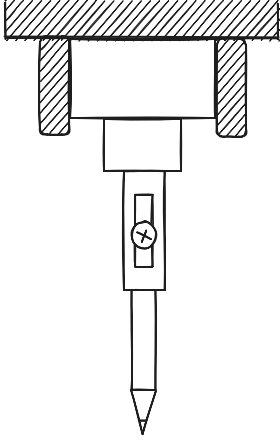


Figure 4. Illustration of penholder with a spring. The gripper can be seen as the blocks with diagonal lines.

3.6 A Parallel Vision Approach

In parallel to the vision method based on contour above, an alternative object detection and pose estimation pipeline was developed, which utilized a YOLOv8 model[7] with a depth camera RealSense D435 to determine the 3D coordinates of Jenga pieces.

Detection and Segmentation with YOLOv8 The YOLOv8 model was trained to detect Jenga blocks in RGB frames captured from the camera. For each frame, the model outputs instance masks along with the class label and confidence scores. Each predicted mask corresponds to the pixel-wise segmentation of a jenga block.

To extract the position and orientation of each block, the binary mask was processed using the OpenCV function `cv2.minAreaRect`, which fits a minimum-area rotated rectangle around the mask contour. This function returns the center point (u, v) in image coordinates, the dimensions (width and height), and the in-plane rotation angle θ relative to the horizontal axis.

Once the 2D center point (u, v) was obtained, its depth value Z_c was retrieved from the aligned depth map. Using the camera's intrinsic matrix parameters (f_x, f_y, c_x, c_y) , the 2D center and depth value were back-projected into a 3D point

(X_c, Y_c, Z_c) in the camera coordinate system by:

$$X_c = \frac{(u - c_x) \cdot Z_c}{f_x}, \quad Y_c = \frac{(v - c_y) \cdot Z_c}{f_y}, \quad Z_c = d(u, v)$$

Eye-to-Hand Calibration To transform the 3D coordinates from camera system to robot based coordinates, eye-to-hand calibration was performed using the *easy hand-eye* tool [4]. The transformation matrix was computed by recording multiple robot poses and capturing corresponding poses of an ArUco marker attached to the gripper, with its center precisely aligned with the Tool Center Point. A series of such pairs were collected semi-automatically by moving the robot to various configurations. Then the $\mathbf{AX} = \mathbf{XB}$ problem was solved, where $\mathbf{A}$ represents the relative motion of the robot's end-effector between two poses, and $\mathbf{B}$ is the corresponding relative motion observed from the camera. Using the computed $\mathbf{X}$, all points in the vision system can be accurately expressed in the robot's coordinates system.

An example of the real-time detection output is shown in Figure 5, where the estimated 3D coordinates and angle of the Jenga block are displayed in the robot's coordinate frame, verifying the success of the coordinate transformation pipeline.

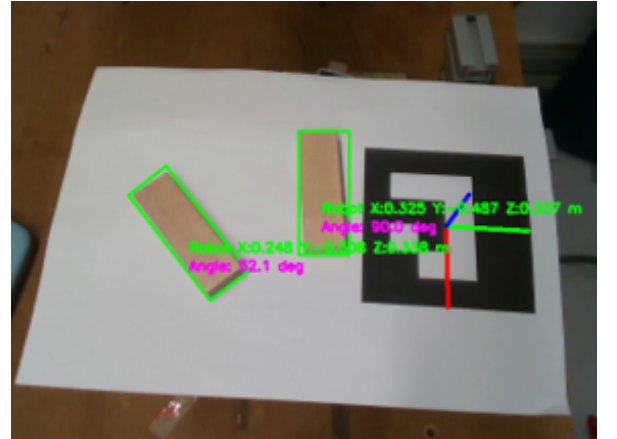


Figure 5. Real-time jenga detection result in robot coordinates

3.7 Robot Arm

The robot arm in use is the UR5e[11] which is a collaborative robot. Which has a gripper at the flange. The robot has six joints, which means it's a 6-dimensional matrix for its movements.

The goal of the robot arm is to move Jenga blocks from positions given by the computer vision system to a predetermined location to build a tower. This is done using trajectory planning via MoveIt [9] which has continuously added spatial constraints to avoid hitting the tower. The control algorithm is written in Python and communicates with the robot arm using ROS [8] (Robot Operating System).

3.8 ROS

For this project ROS 1 was used. This is due to the handler having the most experience with this. ROS allows for communication from the computer to the robot software. This program is commonly used when using and developing robotics.

For this project it will just be used for communication as an addition plugging called MoveIt will be in charge of control.

3.9 MoveIt

MoveIt is an add-on to ROS, MoveIt allows for easier control of the robot using motion planning. The motion planning have built in solvers, which finds a path from a starting position to a desired location. These solvers can be switched to use a preferred one, however in this project the trajectories are hand made in a python script. The reason for this is to increase the speed and control of the robot. Since a solver will almost always find a path if it's feasible. However this can be problematic since this path necessarily isn't the fastest. This was discovered in the testing phase, which sometimes resulted in self-collision. In order to solve this, hand made trajectories was implemented.

3.10 Python to MoveIt

As previously mentioned, MoveIt will be used to control the movements and trajectories of the robot. MoveIt also allows for a python script to control the planning and actions of the robot. This is what was used for the final solution. A skeleton for the script was found on the MoveIt website [6] which had some prebuilt function for control. These were implemented into the script in order to solve the problem. The script works through iterations where in each iteration a block of the Jenga tower is placed in a desired location. The script will continue to iterate till each of the Jenga blocks are picked up and placed in the tower. As mentioned in the previous part, handcrafted trajectories were created. These trajectories are made with a path planner which was found from the online script. This path planner was used to force the robot to move along a Cartesian path. The Cartesian path results in slower speed but a safer path. Which is favorable as the speed doesn't matter if the robot isn't able to finish the construction. The Python script also allowed for easier integration between computer vision and control. In the script all the blocks are added in a list containing the x,y and a angle. The angle is around the z-axis. This list is "popped" in each iteration, allowing the control to get the location of the next Jenga block. The MoveIt program was unable to identify the gripper, so a virtual block was placed in it place to avoid self-collision. The wall next to the robot and the table the robot was standing on was also added as constraints to avoid collisions, especially before implementing cartesian paths when the robot took seemingly more random routes.

3.11 Jenga Theory

In order to build the tower with speed and precision one must understand how to approach this in real life, then apply the constrains of the robot. The main constrain of the robot is the gripper. The gripper in this case is a clamp, which means the grip force will be applied from two sides of an axis. This isn't a large concern in the picking process but rather a problem in the placement process. Since the clamps have thickness of 7 mm which is required to be taken into account when placing the blocks. This thickness will correspond to the spacing between the placed blocks, which wont result in a correct Jenga tower. A representation of this can be found in figure 6. In order to correct this a push function is required to be implemented

in order to get rid of the space. The push function will push the blocks from each side so the blocks are side to side. The original idea was to grab the pieces by the smallest sides, i.e. the sides that are furthest away from each other, which couldn't be implemented due to the grippers small size.

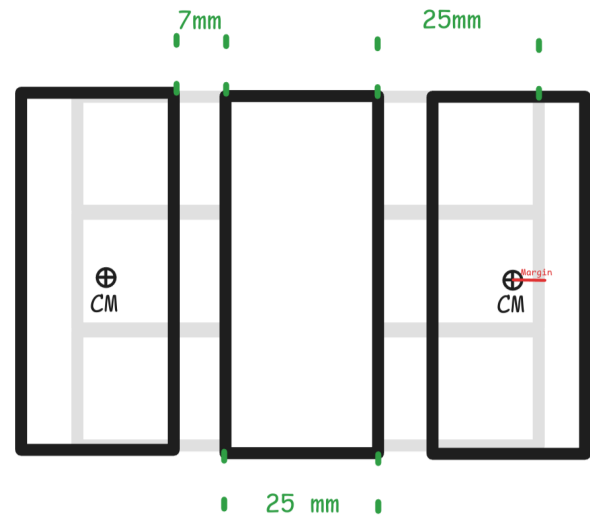


Figure 6. Top view: Illustration of building process. The margin is 6mm

4. Implementation

In the following section the implementation processes will be presented. This is done for the computer vision, robot arm.

4.1 Computer Vision

The computer-vision system worked directly upon integration with the rest of the program. The calibration had to be redone a few times since the camera was moved.

4.2 ROS

ROS was required to be installed in order to setup the communication from the computer to the robot. This was done following the ROS installation guide on their website[1]. ROS 1 melodic was used.

4.3 MoveIt

Installing MoveIt was a necessity in order to achieve easy control of the robots motion. The installation was done by following their online tutorial[10] and having access to the robots Xacro file which was also from their website. The Xacro file plays a central role in the installation as it holds the data for the robot arms dynamics and settings. With the use of the Xacro file and MoveIt setup assistant, a joint chain was configured to represent the joints of the real robot. This setup allows the MoveIt add-on to have control over the robot and plan paths which avoids self-collision.

4.4 Jenga process

The final process for building the tower consist of three for-loops. Where one encloses the other two. The for-loops enclosed represent the loops for building a layer, one where the length is in y-direction and the other in x-direction. This process will continue until the list containing the Jenga blocks

location is empty or the limit in the enclosing loop is reached. In the enclosed loops an offset was implemented to the placement of the middle block, causing in to be further out in x or y direction. This was done to compensate for potential offsets from the computer vision and from the table not being perfectly aligned in z-plane. This will result in the tower not being a complete square when viewed from above, but it will ensure that blocks doesn't fall off due to offsets. The push function that was discussed in System design is applied after one of the enclosed for-loops completes. This means that the blocks will have no space between them when the next layer is applied.

5. Results

The result will be briefly presented in this section. A further discussion of the results, the working process and future work will be presented in the next section.

5.1 Computer Vision

The computer-vision-system works well enough to pick up Jenga pieces, there is however an error resulting from the assumption that the camera is perpendicular, rotated correctly in the z-axis and that the camera is far away to be assumed to be telescopic. Figure 7 shows how the error changes as the Jenga pieces are moved away from the camera.

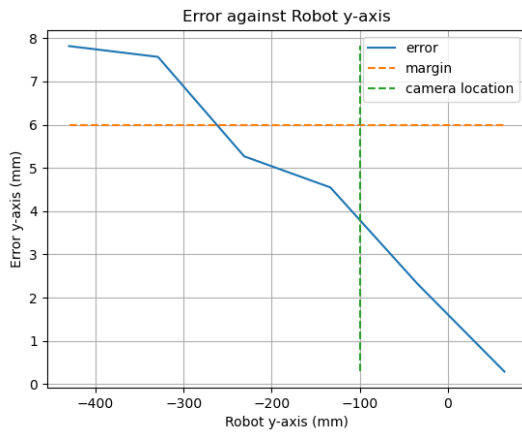


Figure 7. Error of computer-vision system.

5.2 Robot Arm

The robot moves in the desired way in safe trajectories using hand made Cartesian trajectory. The installations and set ups including the python script are working fine. The robot is able to move to a desired location with a safe path, that doesn't collide with the Jenga tower.

First, the robot was implemented without using Cartesian trajectories which sometimes caused unsafe trajectories in which the arm collided in on itself. And there were issues with the coordinate system to begin with, due to the hand extending the arm. The robot moves the wrist to a certain coordinate and was solved by adding constants to the coordinates which worked since the arm is always oriented downwards.

The robot is still only tested at low speed, it could have

issues with higher speeds which would likely be closer to a real world application. The speed has been set to 10% for safety reasons. With this speed setting each layer takes an average of 40 seconds to build.

5.3 Integration

The computer-vision-system and robotic movement were well integrated, enabling the robot arm to accurately locate and pick up the Jenga piece near its center. Grasping the piece near the center was essential to maintaining stability of the tower and a sign of smooth integration of the hand-eye coordination.

6. Discussion

In the followings section a discussion regarding the computer vision and the robot arm will ensue. Here the result will be discussed as well as future improvements.

6.1 Computer Vision

Comparing the error seen in figure 7 to the margin seen in figure 6; it is understood that the accuracy is not good enough for Jenga pieces laying far away. In practice it worked better than expected as the layer below also had som errors increasing the margin. In an experiment building a 6 layers tall tower: 1 piece fell out.

The error could have been lowered given more time by doing a measure like that in figure 7, and, then interpolate any new point from the error measurement and subtracting that error from the result. This should work quite well considering that there is a clear trend in figure 7. A problem could be if the error is not independent in the x- and y-axis.

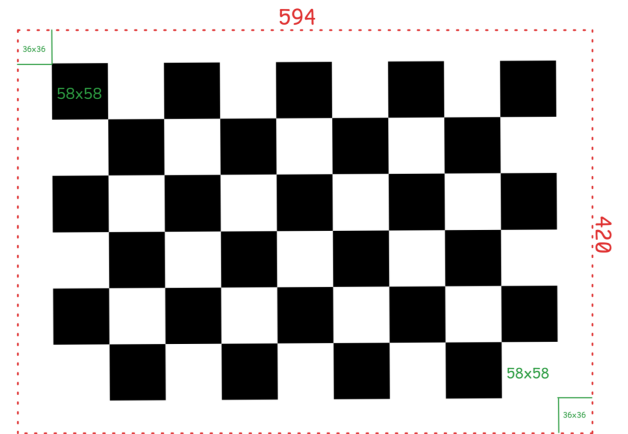


Figure 8. Image of chessboard to be used for calibration. The unit is millimeters

Another method could be to put a grid pattern as seen in figure 8 underneath the camera as a calibration step. This grid pattern would have known dimensions and be placed in a known location to the robot. Afterwards one could simply extract the corners and then do bilinear-interpolation to approximate the point. This method will be evaluated in the future. Outside of the pattern one could use the current method as interpolation will not be possible if outside of the data.

6.2 Robot arm

The installation of ROS and MoveIt posed a greater challenge than expected at the beginning of the project, mainly due to the inexperience of the group with these programs. Handling Linux posed a challenge as well. This resulted in a large portion of the time being spent on this, so the plan for the final solution had to be changed, leaving much room for future improvement. The scope changed to making it work with the framework MoveIt instead of creating a control algorithm and trajectory planner which was more reasonable for the time frame.

If more time was available, the first thing to improve would be to implement a heuristic for choosing which block to pick up. As it is now it just pick the block which is at the top of the list, this list is not sorted either. A heuristic which takes the motion of the arm into account would be able to tell which block is fastest to pick up from the given location. In the grand scheme this would not increase the speed that much as all the blocks are required to be picked up. If able to re-attempt this project with current knowledge, a different control method would be approach. Were it would be possible to achieve optimal control. This would most likely increase the building speed of the Jenga tower.

6.3 Future Work

Currently the project works for creating a Jenga tower at a fixed location. If one were to implement the actual Jenga-playing process one would consider using a custom made gripper, for higher accuracy when playing. The code also needs further development if Jenga-playing is desired, since in its current state it only works for building the tower.

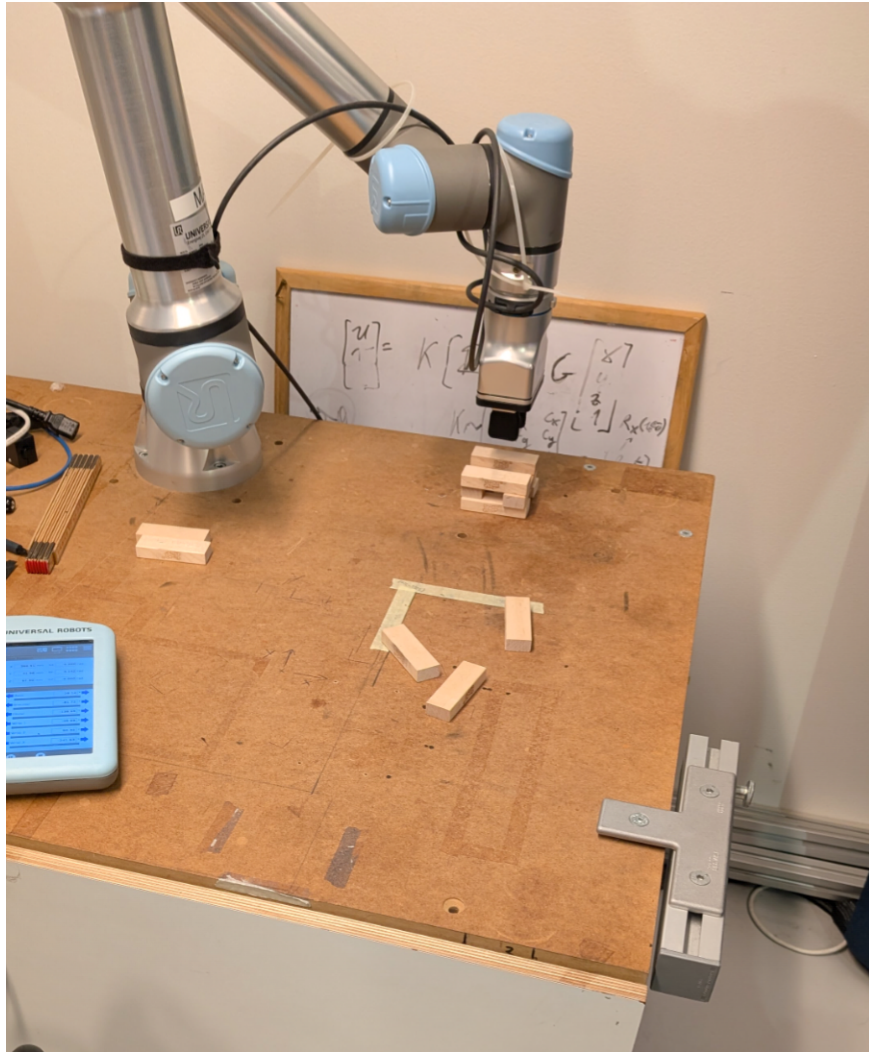
Since the scope of the project was changed from implementing optimal control and trajectory generation to simply making it work, there is much work to be done in the lower levels of the program. The largest task is to choose and implement an optimal control strategy using a kinematic model of the system, with particular emphasis on achieving velocity control. The current implementation operates in the same speed at all times, and the robot could be more efficient and faster by implementing a good algorithm for velocity control.

The computer vision can now only identify blocks laying flat on the table due to hardware limitations and also due to the algorithm being very advanced if depth was an issue too. It would require stereo cameras to handle depth. The main issue with the algorithm is when blocks are on top of each other which makes discriminating between them hard. The computer vision could be expanded upon to handle height in blocks as well.

ROS 1 was used in this project which is obsolete in May 2025. ROS 2 is harder to use but more advanced in areas such as real-time performance, scalability, and communication which would likely make the robot more efficient. And since ROS 1 was used, an old version of MoveIt had to be used with it. Newer versions of MoveIt are more advanced and has better software.

References

- [1] C. C. A. 3.0. *Ubuntu install of ros melodic*. URL: <https://wiki.ros.org/melodic/Installation/Ubuntu> (visited on 2025-05-20).
- [2] bingykang. *Depth-anything-v2*. URL: <https://github.com/DepthAnything/Depth-Anything-V2> (visited on 2025-04-10).
- [3] A. Inc. *Fusion 360*. URL: <https://www.autodesk.com/products/fusion-360/personal> (visited on 2025-05-15).
- [4] marcoesposito1988. *Easyhandeye*. URL: https://github.com/IFL-CAMP/easy_handeye (visited on 2025-05-13).
- [5] OpenCV. *Contours : getting started*. URL: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html (visited on 2025-04-15).
- [6] rhaschke. *Moveit python tutorial*. URL: https://github.com/moveit/moveit_tutorials/blob/melodic-devel/doc/move_group_python_interface/scripts/move_group_python_interface_tutorial.py (visited on 2025-05-03).
- [7] Roboflow. *Yolov8*. URL: <https://roboflow.com/model/yolov8>.
- [8] O. Robotics. *Robot operating system*. URL: <https://www.ros.org/> (visited on 2025-04-20).
- [9] P. Robotics. *Moveit*. URL: <https://moveit.ai/> (visited on 2025-04-24).
- [10] P. Robotics. *Moveit tutorials*. URL: https://docs.ros.org/en/melodic/api/moveit_tutorials/html/index.html (visited on 2025-04-28).
- [11] U. Robots. *Ur5e*. URL: <https://www.universal-robots.com/se/produkter/ur5-robot/> (visited on 2025-05-20).



Robot Jenga

Alexander Hansson¹ Shuying Liu²

¹nils.alexander.hansson@gmail.com ²liushuying.blaise.2490@gmail.com

Abstract: This paper describes a robotic system for the autonomous construction of Jenga towers using the UR5e manipulator, which integrates perception, reasoning, and execution modules. The computer vision module uses a single camera and initially uses YOLO for Jenga block bounding box detection, assuming a shared Z-value of zero. The reasoning module computes the pick-and-place pose based on these perceptions, following a leveled grid model. The execution module is managed by ROS Melodic and MoveIt using inverse kinematics. A socket-based communication resolves Python version conflicts between perception and planning. The finalised system demonstrates reliable control, with the ability to continuously stack blocks and build towers, highlighting the importance of strong system integration for complex operational tasks.

1. Introduction

The aim of this project is to design and implement a robotic system that is capable of autonomously building a simple tower using a UR5e robotic arm. The system integrates computer vision for detecting scattered blocks, planning algorithms to determine the position of each block, and a robot control module to perform pick-and-place operations.

Autonomous operation is a central challenge in robotics, especially in tasks that require precision, planning and perception. Building the Jenga tower is representative of such tasks, combining object detection, motion planning, and control into a coordinated system. With this project, we aim to deepen our understanding of autonomous robotic systems and demonstrate vision-based manipulation.

1.1 Literature Review

Robotic manipulation tasks, such as pick-and-place operations and structured assembly, have been extensively explored in recent research, often utilising vision-guided control and robotic arms such as the UR series.

In terms of perceptual components, Marchionna et al. [4] proposed a cost-effective robotic system to play the Jenga game using an e.DO 6-DOF manipulator, a depth camera and a unidirectional force sensor. Their system applies deep instance segmentation to detect Jenga blocks and uses visual servoing to accurately align the manipulator to successfully extract up to 14 blocks in a row.

For reasoning and decision making, Fazeli et al. [1] developed a strategy that fuses visual and haptic information to enable the robot to determine which blocks can be safely extracted without destabilising the tower. Using an ABB IRB 120 robot and force-sensing wrist cuffs, the robot learned to predict the removability of blocks by correlating visual and haptic feedback. More recently, Puthuveetil et al. [5] introduced a graph-based model for representing the dynamics of the Jenga tower and simulating the consequences of removing specific blocks. This approach facilitates more strategic planning and decision-making, especially in simulation environments.

On the execution side, Hundt et al. [3] implemented a system playing Jenga to support block detection and manipulation using a UR5 robot, a custom-designed end-effector, and sensors such as a miniature load cell and LiDAR. Their system, integrated in ROS, successfully performed multiple block moves autonomously and demonstrated precise control in a real-world robotics setup.

In contrast to these existing works, our project focuses on real-world system integration with ROS Melodic and MoveIt using the UR5e robot running on Ubuntu 18.04. Instead of using ROS native communication between perception and planning, we implemented a socket-based interface that connects a YOLO-based object detection module written in Python 3.9 to a robot planner implemented in Python 2.7. This system allowed the construction of the Jenga tower to be done autonomously within practical constraints and limited hardware.

2. Modeling

This section describes the model design of the robotic system, with a focus on the reasoning and execution components. These modules work together to translate vision-based perception into actionable robot motions for stacking Jenga blocks.

2.1 Computer Vision training and precision

During training, the YOLO model followed a train-validate-test sequence, a common method for training a model that eventually works on unseen data. The model is first trained on one part of the dataset. Then, hyperparameters such as learning rate and number of layers are tuned on another part of the dataset. Lastly, the model is tested on the last part. The performance in each section is measured by comparing the bounding boxes of what the model is confident corresponds to a Jenga block, with the bounding box in the labelled data, calculating the IoU between the two bounding boxes, serving as a measurement of precision:

$$\text{IoU} = \frac{\text{Area of Intersection}}{\text{Area of Union}}$$

This serves as the basis for a mAP@50 precision score for the model, where a score of 100% corresponds to perfect performance, where all predicted bounding boxes overlap with the true boxes by at least 50%, thus having an $\text{IoU} \geq 0.5$

2.2 Reasoning Module

The reasoning module functions as the *task-level planner* for the Jenga-playing system. Its primary responsibilities are to determine:

1. The pose from which to pick up a Jenga block, based on input from the perception module.
2. The pose where the picked block should be placed to construct the Jenga tower, based on the tower's current state and construction rules.

The module receives the pose of a Jenga block to be picked from the perception module. This is typically a 3D pose:

$$\mathbf{P}_{\text{block}} = (x_s, y_s, z_s, \theta_s)$$

where (x_s, y_s, z_s) are the world coordinates of the block's center, and θ_s is its yaw angle (in-plane rotation) in the robot's base frame. For blocks lying flat, z_s might be relative to the surface they are on.

Based on $\mathbf{P}_{\text{block_supply}}$, the reasoning module generates the pick-up pose $\mathbf{P}_{\text{pick}}$. This often involves adjusting the Z-coordinate for a pre-grasp approach and ensuring the gripper aligns with the block:

$$\mathbf{P}_{\text{pick}} = (x_s, y_s, z_s + \Delta z_{\text{approach}}, R_z(\theta_s))$$

where $\Delta z_{\text{approach}}$ is a small offset above the block for a safe approach, and $R_z(\theta_s)$ is the rotation matrix corresponding to the yaw θ_s to align the gripper with the block's orientation. The exact values also incorporate gripper-specific offsets.

The core of the placement logic is encapsulated within the JengaTowerPlanner class. This class is responsible for calculating the precise pose $\mathbf{P}_{\text{place}}$ for the next Jenga piece in the tower.

- Tower Origin in Robot Frame: The planner is initialized with the origin of the Jenga tower's own coordinate system, expressed in the robot's base frame:

$$\mathbf{O}_{\text{tower_in_robot}} = (x_{\text{origin}}, y_{\text{origin}})$$

- Jenga Tower's Local Coordinate System: The JengaTowerPlanner internally reasons about piece placement within the Jenga tower's own local coordinate system. The origin (0,0,0) of this local system is at the horizontal center of the tower's base. The Z=0 plane of this local system corresponds to the vertical center of the pieces in the first layer.
- Piece Placement Logic in Tower Local Frame: Let k be the current layer index (starting from 0) and j be the piece index within that layer (0 or 1, for two pieces per layer). Let h_{block} be the height of a Jenga block, and d_{xy} be the piece_xy_offset_m , which is the distance from the tower's center to the center of a piece along its primary axis for that layer.

- Z-coordinate (center of the piece) in Tower Frame:

$$z_{p,\text{tower}} = k \cdot h_{\text{block}}$$

- Yaw Angle in Tower Frame: The yaw $\theta_{p,\text{tower}}$ alternates with each layer:

$$\theta_{p,\text{tower}} = \begin{cases} 0 \text{ rad } (0^\circ), & \text{if } k \bmod 2 = 0 \\ \pi/2 \text{ rad } (90^\circ), & \text{if } k \bmod 2 = 1 \end{cases}$$

- X, Y-coordinates in Tower Frame: Let s_j be a sign factor for the piece index within the layer, typically $s_0 = -1$ (for the first piece, $j = 0$) and $s_1 = +1$ (for the second piece, $j = 1$). The specific implementation maps piece indices to positive or negative offsets.

- * If layer k is horizontal ($k \bmod 2 = 0$):

$$\begin{aligned} x_{p,\text{tower}} &= \pm d_{xy} \\ y_{p,\text{tower}} &= 0 \end{aligned}$$

- * If layer k is vertical ($k \bmod 2 = 1$):

$$\begin{aligned} x_{p,\text{tower}} &= 0 \\ y_{p,\text{tower}} &= \pm d_{xy} \end{aligned}$$

- Transformation to Robot Base Frame for Placement Pose $\mathbf{P}_{\text{place}}$:

The JengaTowerPlanner transforms the tower-local coordinates $(x_{p,\text{tower}}, y_{p,\text{tower}}, z_{p,\text{tower}}, \theta_{p,\text{tower}})$ into the robot's base frame to compute the final placement pose components.

The placement pose can be conceptualized as $\mathbf{P}_{\text{place}} = (x_t, y_t, z_t, \theta_t)$, where (x_t, y_t, z_t) are the Cartesian coordinates and θ_t is the scalar yaw angle in the robot's base

frame. These are calculated as:

$$\begin{aligned} x_t &= x_{p,\text{tower}} + x_{\text{origin}} \\ y_t &= y_{p,\text{tower}} + y_{\text{origin}} \\ z_t &= z_{p,\text{tower}} \\ \theta_t &= \theta_{p,\text{tower}} \end{aligned}$$

The `get_next_piece_info()` method of the JengaTowerPlanner returns these transformed values as a tuple $(x_t, y_t, z_t, \theta_t)$. The robot's execution module would then use θ_t to derive the necessary orientation (e.g., by constructing $R_z(\theta_t)$).

Both the calculated $\mathbf{P}_{\text{pick}}$ and $\mathbf{P}_{\text{place}}$ poses are formatted into a consistent data structure. This data is then typically sent to an execution module (e.g., a robot motion planner or controller) for physical execution. The description mentioned socket communication, which is a viable method for inter-process communication, especially if the reasoning module and the robot control stack are in different processes or even programming language environments.

2.3 Hand-eye Coordination

Since it is assumed that all Jenga blocks share the same z-value, only the x- and y-values need to be correctly translated from $[u \ v]$ pixels in the camera to $[x \ y]$ values in the robot coordinate system. Furthermore, since the camera coordinate system is π radians rotated in comparison to the robot coordinate system, the angle of the Jenga block in the robot frame, θ_{robot} , can be found using of the the angle in the camera frame, θ_{cam} as follows:

$$\theta_{\text{robot}} = \theta_{\text{cam}} + \pi$$

Given this setup, the hand-eye coordination can be simplified immensely in comparison to traditional hand-eye coordination, especially since the intrinsic of the camera is unknown¹. The difference between the camera pixel $[u \ v]$ and the robot location $[x \ y]$ can be estimated using a scalar and translation. The scalar and translation were found by selecting two pairs of camera coordinates and robot coordinates,

$$[u_1, v_1] \mapsto [x_1, y_1] \quad [u_2, v_2] \mapsto [x_2, y_2]$$

In other words, the scalars and translations can be found by solving:

$$\begin{aligned} \begin{bmatrix} x \\ y \end{bmatrix} &= s \cdot \begin{bmatrix} u \\ v \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix} \\ &\Rightarrow \\ \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} &= \begin{bmatrix} s & 0 & t_x \\ 0 & s & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \end{aligned}$$

By solving the system for the scalar and translation, any pixel in the camera frame can be translated to the 3D location in the robot coordinate system. The estimation will be most precise close to the two sample points, as those locations have the lowest distortion.

¹This approach was inspired heavily by Group E, of course with their permission.

2.4 Execution Module

The execution module is responsible for interpreting the received pose data and controlling the UR5e robot to execute corresponding pick-and-place operations. Implemented in Python 2.7 under ROS Melodic, this module uses MoveIt's `move_group` interface for joint-space planning.

Once pick and place poses are received, the execution process follows a fixed sequence:

1. **Approach pick pose:** Move the robot above the block position at a safe vertical offset.
2. **Pick operation:** Open the gripper, descend to the block, close the gripper, and lift up slightly.
3. **Approach placement pose:** Move above the target stack position.
4. **Place operation:** Descend, open the gripper slightly to release the block, lift, and close the gripper.
5. **Return to home:** Move to a predefined home pose to reset for the next cycle.

MoveIt computes inverse kinematics solutions and plans collision-free trajectories for each motion. Vertical movements are executed to ensure safe and precise placement. The process is repeated until a termination signal is received or no blocks remain.

This design separates high-level decision-making from low-level control, improving modularity, maintainability, and reproducibility.

3. System Design

This project is built around a real-world robotic system capable of constructing a Jenga tower through autonomous manipulation. The system integrates perception, planning, and control components, with the central hardware being the UR5e robotic arm. The perception serves to detect pieces and estimate their position and orientation. The reason component handles the internal logic of the stacking task. It maintains the state of the tower. It is used to determine the next placement of a piece, given the current state, based on a predetermined order in which blocks have been decided to be placed. Lastly the actuation is implemented in a robot control, which is responsible for executing the physical actions of the robot. It receives commands from the Reasoning module in the form of target positions for picking and placing blocks and translates them into motion commands. This section focuses on the physical setup and overall control concept of the system.

3.1 Computer Vision

As only one camera was available for the computer vision, the depth could not be deduced in the traditional sense. Therefore, the computer vision was implemented under the assumption that all Jenga pieces start off having the same z-value, i.e., none were lying in a pile. On a high level, the computer vision initially uses machine learning to detect individual Jenga pieces. Furthermore, the computer vision part works under the assumption that the camera sees the Jenga pieces from a

bird's eye view, as seen in figure 2. The computer vision was divided into two components, one for detecting individual Jenga pieces and one for refining the detected pieces. Initially, machine learning is used to "bound box" all the Jenga pieces visible camera. Having found the Jenga piece, it needs to be turned into a digital representation of the Jenga block, so the gripper can pick it up, meaning the location of the piece in 3D space needs to be found, as well as its orientation. However, the bounding box is not precise enough to be used by the gripper to pick up the block, nor is it useful for finding the orientation of the Jenga piece. Therefore, the bounding box needs to be optimized. Therefore, the exact pixels corresponding to the Jenga piece contained in the bounding box are found. Lastly, these pixels are used to provide the exact location and orientation of the Jenga piece.

3.2 Hardware Components

The primary hardware components used in the system are:

- **UR5e Robotic Arm:** A 6-degree-of-freedom industrial manipulator from Universal Robots, known for its precision and flexibility. The UR5e serves as the actuation platform for executing pick-and-place operations in the Jenga stacking task.
- **Two-Finger Gripper:** A parallel gripper is mounted on the end-effector of the UR5e for grasping and releasing Jenga blocks. It is electrically actuated and capable of performing basic open/close operations with sufficient repeatability for structured tasks.
- **Jenga Workspace:** A flat tabletop is used as the workspace for stacking the blocks. The base height of the table is used as a reference for all vertical motion planning.

No mechanical modifications were made to the robot. The setup was assembled using off-the-shelf hardware. A photo of the system setup is shown in Figure 1.



Figure 1. UR5e robot arm with gripper mounted, operating in the Jenga stacking environment.

3.3 Control Architecture Overview

The robotic system is structured around a modular control architecture that separates perception, reasoning, and execution. The perception module detects Jenga blocks and sends pose information to the reasoning module. The reasoning module determines pick and place poses and sends them to the robot controller. The UR5e arm executes the motions using an internal control loop, responding to pose targets and gripper commands.

The communication between modules is implemented using socket-based messaging, allowing for compatibility between different software environments. This design also supports asynchronous operation and modular debugging, making the system robust and adaptable.

4. Implementation

This section serves to explain how the system from the "system design" section was implemented. The code was divided up into three parts, namely computer vision, reasoning, and robotics, as aforementioned.

4.1 Computer Vision

To detect individual Jenga pieces, a model called YOLO was used. The YOLO model was forked from a project, courtesy of gannhan [2] on Roboflow, a website used to build and deploy computer vision models, trained on 1000 labelled images of Jenga pieces, this ending up with (THE VERSION USED IN THE PROJECT) YOLO provides a simple bounding box of the Jenga piece, meaning a rough estimate of its location. However, as aforementioned, the rough estimate needs to be improved for the robot to be able to pick up the Jenga piece using the gripper. Therefore, the Segment Anything Model (SAM) was used. Segment Anything takes an initial guess of an arbitrary object and creates a precise binary mask, stored as a matrix of the same size as the image, representing only pixels that are part of the Jenga block with true values. See the red pixels in figure 2, corresponding to the binary mask.



Figure 2. The binary mask in red, on a Jenga piece. The red represents the binary mask produced by SAM, the blue represents the estimated center of the block, and the yellow represents the orientation of the block.

The center of the bounding box was fed into SAM as an initial guess, thus providing a binary mask of the Jenga block.

Using the binary mask, the center of the block in 2D space was found with the median of the x, respectively the y "true"

values. The median was used rather than the mean as it, in contrast to the mean, is not affected by any potential outliers produced by the Segment Anything model. Lastly, the orientation of the block was found using principal component analysis (PCA). PCA is a dimensionality reduction technique that finds the directions of maximum variance in the data, the so-called "principal components". Applying it to the binary mask thus provides the exact orientation of the block, as the first principal component follows the length of the block. See the yellow arrow in figure 2 corresponding to the first component of the binary mask.

4.2 Execution Implementation

This section describes the practical implementation of the final integrated system, which successfully performed autonomous Jenga tower construction. The setup was built using Ubuntu 18.04, ROS Melodic, MoveIt for motion planning, the `ur_modern_driver` for robot communication, and a custom socket-based interface for cross-module messaging.

Environment Setup The implementation was done on a local machine running **Ubuntu 18.04**. The following software packages were installed:

- **ROS Melodic** — Installed via apt package manager, serving as the middleware for robot control and integration.
- **MoveIt** — Installed as a ROS package for inverse kinematics, planning, and execution.
- **ur_modern_driver** — Used to communicate with the UR5e robot. It subscribes to `trajectory_action` topics and sends commands via a TCP connection to the UR controller.
- **Python 2.7** — Required by ROS Melodic for core functionality. The execution module was written in Python 2.7.
- **Python 3.8** — Used separately for the perception module, which required more recent computer vision libraries.

Socket Communication Integration Due to version incompatibilities between ROS Melodic (Python 2.7) and modern computer vision libraries (Python 3.9), inter-process communication was implemented using TCP **socket communication**. A server socket was created in the execution module (Python 2.7), while the reasoning module (or external vision system) acted as a client in Python 3.9.

Pose data was serialized and transmitted in the following format:

```
data = [0.35, -0.12, 0.02, 1.57]
```

The same structure was used for both pick and place messages, with an additional stop flag sent to signal task termination.

MoveIt Configuration A MoveIt configuration package was generated using the MoveIt Setup Assistant with the following parameters:

- `ur5e.urdf.xacro` as the robot description
- `tool0` as the end-effector link
- Planning group: `manipulator`
- Robot base link: `base_link`

The execution module used the `move_group` Python interface to perform motion planning. For each received pose, MoveIt computed a collision-free inverse kinematics solution and executed the trajectory through the driver.

Motion Sequence Execution The robot performed the following sequence for each block:

1. Check the status of jenga pieces to continue or stop
2. Move to a pre-pick position above the block
3. Descend to the pick position and close the gripper
4. Lift the block and move to a pre-place position
5. Descend to the place position and release the block
6. Return to the home pose and wait for the next instruction

The gripper was controlled through custom open/close commands issued via digital I/O, synchronized with the robot’s movement phases.

5. Results

5.1 Object Detection

The object detection model was tested in two ways. First, during training, the model is evaluated based on `mAP@50`. In our case, the validation and test sets had an `mAP@50` of 99.5% and 100% respectively. In other words, the model’s average precision, considering detections with $IoU \geq 0.5$, was nearly perfect. In the RobotLab, we found that a threshold of 80% worked the best, i.e., all objects that the model was 80% confident to be Jenga block, were considered Jenga blocks in the computer vision. This provided no false positives, nor any true negatives. Furthermore, the actual Jenga blocks, the model was always about 87% to 96% confident, meaning a threshold of 80% leaves a margin. Testing the model on surfaces outside the RobotLab, the model often performed well, but occasionally performed very poorly. For instance, with a black background, the model often predicted irrelevant floor tiles to be more likely to be Jenga blocks than the actual Jenga blocks.

5.2 Segment Anything Model

The SAM model was tested on both images taken in the robot lab and images found on RoboFlow. The tests were from the very start promising, as there were no visible outliers in the mask, i.e. no segmented pixels that are far away from the Jenga block. The lack of outliers is best visualized by plotting the distance to the center of the binary mask for each pixel found to correspond to a Jenga block, see figure 3. Since no pixel has a distance that is further away than any other pixel, it can safely be assumed that there are no outliers in the segmentation phase.

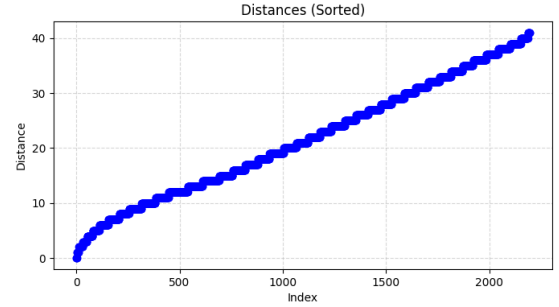


Figure 3. The distance to the center of the mask, for each pixel found by the segmenter

5.3 Execution

To evaluate the performance of our system, we conducted real-world tests using the fully integrated setup described in the previous sections. The robot was tasked with autonomously detecting, picking, and stacking Jenga blocks based on pose data received from the perception module via socket communication.

A critical aspect of the system was ensuring reliable communication between the Python 3.9-based perception and reasoning modules and the Python 2.7-based execution module. The terminal shows successful socket communication by “connected”. Messages are tagged with identifiers such as “jenga”, “tower”, and “status”, and include 4D pose information in the form $[x, y, z, \theta]$. This real-time communication bridge confirmed the correctness of data exchange, enabling accurate control flow and robot motion planning.

Once communication was verified, the robot successfully executed multiple stacking cycles using the transmitted pose data. In each cycle, the robot received the pick and place instructions, planned a trajectory using MoveIt, and controlled the gripper to manipulate the Jenga blocks. The tower orientation alternated correctly across layers, and the robot maintained proper alignment between the gripper and the blocks.

In successful trials, the robot constructed towers without human intervention. The execution was smooth, with no observed collisions, drop events, or missed poses. The motion planning system reliably produced collision-free paths, and the modular control structure proved stable and responsive.

Figure 4 shows a result of the stacking process in progress, illustrating the physical interaction of the robot with the blocks.

These results demonstrate that the proposed system architecture is robust, modular, and effective in performing struc-

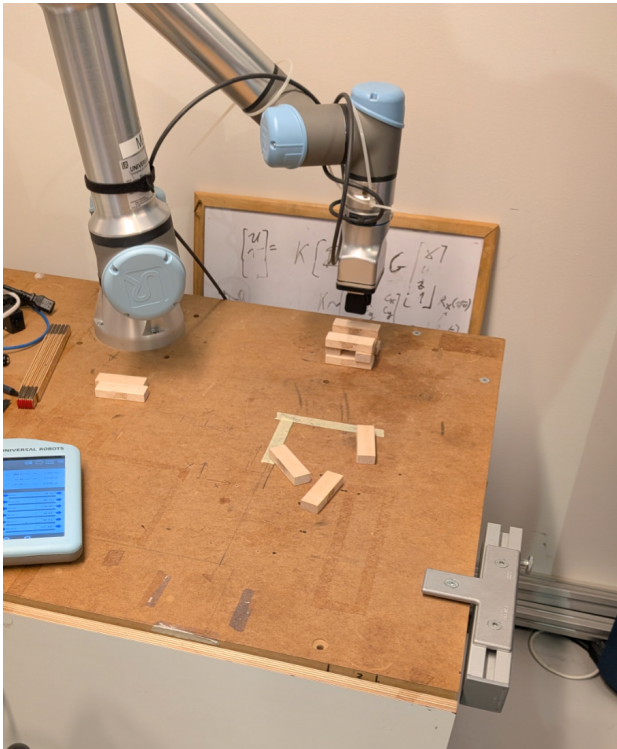


Figure 4. Result of robot actions during autonomous Jenga tower construction.

tured assembly tasks using vision-guided robotic control under real-world constraints.

6. Discussion

6.1 Object Detection

When creating the object detector, GroundingDINO was initially used. GroundingDINO is a zero-shot object detector model, meaning it can detect Jenga pieces without necessarily having been trained on Jenga blocks. Only the base model of GroundingDINO was used to detect pieces. The model was tested on images taken in the RobotLab, and consistently found by GroundingDINO, but with a very low confidence level (10% - 16%). Therefore, objects that were somewhat similar to a Jenga block were occasionally found with the highest confidence level, and thus selected to be picked up by the gripper. Furthermore, this was not solvable by preprocessing or adjusting the prompts of what to detect ("small wooden block", "jenga block", etc) provided to the model. This may highlight that the Jenga object is a uncommon enough object, where a general zero-shot model is ineffective, leading to a model specifically trained on Jenga blocks being used, the YOLO model

Given that the object detection model had validation and test sets with mAP@50 of 99.5% and 100% respectively, it seemed to be almost perfect. However, in the real world, the model often performed exceptionally, such as in the RobotLab, and occasionally very poorly. Looking at the dataset, it lacks a lot of variety as all the pictures were taken with similar lighting, background, and angle. This increases the risk of overfitting, i.e., giving near-perfect results on the validation

set, but performing poorly in the real world, as a consequence of only being capable of detecting objects that follow precisely the conditions used in the training data. In other words, has the lack of variety in the dataset likely led to a generalization issue, where the model performs exceptionally in the robot lab, but occasionally terribly. Had the model performed poorly in the robot lab, this would be solvable by adding more variety to the training data.

6.2 Segmentation

In terms of the SAM model, it performed exceptionally well, as there were no visible outliers. Furthermore, the fact that the median, a outlier-safe average calculation, of the binary mask was used to calculate the center of the mask, it is likely very precise. However, PCA, was used to calculate the orientation of the mask. Since PCA relies on variance and covariance, which are influenced by extreme values, PCA is sensitive to outliers. This did not seem to be a big issue when picking up the Jenga block, however, both since SAM was so precise and because the gripper was big enough to allow for some error in the orientation. Had this not been the case, RPCA could have been used, which has the same use case as PCA, but is not sensitive to outliers.

6.3 Environment Evolution and Debugging

Throughout the development of this project, one of the most challenging and time-consuming aspects was establishing a working software environment capable of controlling the UR5e robot while integrating modern computer vision tools. The planner module, in particular, was significantly delayed due to a series of failed attempts with incompatible or unstable setups.

- **v1-v2:** Ubuntu 20.04 with Docker or native, ROS 2 Humble, MoveIt 2, and VirtualBox. These versions failed to communicate with the real robot due to network configuration issues within Docker and limitations of the ROS 2 UR5e drivers at the time.
- **v3:** Ubuntu 22.04, ROS 2 Humble, and VirtualBox. Although the robot could be reached, MoveIt 2 could not reliably execute planned motions on the real hardware.
- **v4:** Ubuntu 22.04 with Windows Subsystem for Linux (WSL). This setup failed due to the lack of independent IP assignment in WSL, making robot communication impossible.
- **v5:** Ubuntu 18.04, ROS Melodic, MoveIt, and `ur_modern_driver`. While the robot could be reached, ROS node communication could not be launched reliably, possibly due to inconsistent dependency states.
- **v6 (final):** Ubuntu 18.04, ROS Melodic, MoveIt, and `ur_modern_driver`, with TCP socket communication replacing ROS topics. This version worked consistently and enabled reliable integration between the Python 3.9-based perception module and the Python 2.7-based robot planner.

6.4 Final Conclusions

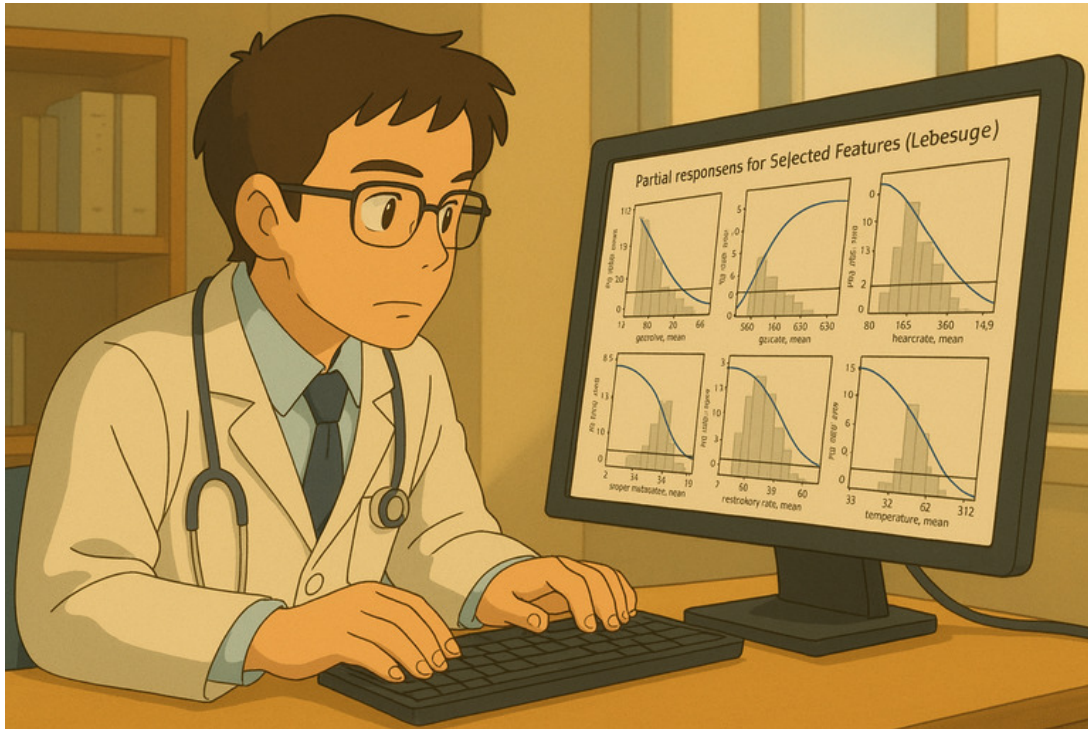
Much of the time originally scheduled for developing the planner logic was spent debugging and experimenting with the system environment. This poor scheduling stemmed from early assumptions that the environment setup would be straightforward and that we would remain as four team members. In reality, hardware communication, IP routing, and version compatibility presented persistent and complex challenges. These delays significantly compressed the timeline for implementing and testing the planner module. As a result, most of the integration and logic development had to be rushed after the environment was finally stabilized.

During this process, extensive support and advice from the project supervisor were critical. Suggestions such as switching to ROS Melodic, abandoning ROS-native communication in favor of socket messaging, and avoiding WSL ultimately led to the successful configuration. Additionally, when node packages and dependencies broke during testing, the supervisor provided essential help in identifying viable recovery options.

The key lesson is that environment setup and hardware communication should never be underestimated in a robotics project. Additionally, socket communication proved to be a lightweight and reliable alternative when ROS message compatibility issues arise between different Python versions. Most importantly, this experience emphasized the value of system-level debugging and clear architectural separation between perception, planning, and control. Although initially frustrating, resolving these issues helped build a more robust understanding of robotic system integration.

References

- [1] N. Fazeli, M. Oller, J. Wu, Z. Wu, J. B. Tenenbaum, and A. Rodriguez. “See, feel, act: hierarchical learning for complex manipulation skills with multisensory fusion”. *Science Robotics* **4**:26 (2019), eaav3123.
- [2] gannhan. *Farino_gripper_wooddataset*. https://universe.roboflow.com/gannhan-vry7x/farino_gripper_wood. Open Source Dataset. visited on 2025-05-18. 2025. URL: https://universe.roboflow.com/gannhan-vry7x/farino_gripper_wood.
- [3] A. Hundt, V. Jain, C.-H. Lin, C. Paxton, and G. D. Hager. “The costar block stacking dataset: learning with workspace constraints”. In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2019, pp. 1797–1804.
- [4] L. Marchionna, G. Pugliese, M. Martini, S. Angarano, F. Salvetti, and M. Chiaberge. “Deep instance segmentation and visual servoing to play jenga with a cost-effective robotic system”. *Sensors* **23**:2 (2023), p. 752.
- [5] K. Puthuveetil, X. Zhang, K. Yokoyama, and T. Narita. “Strategic jenga play via graph based dynamics modeling”. *arXiv preprint arXiv:2505.09377* (2025).



Explainable AI for Critical Care Prediction

David Knezevic¹ Harry Wik² Teo Bergkvist³ Rowaida Panai Ersbo⁴

¹knezevic.david99@gmail.com ²harrywiksmail@gmail.com ³bergkvist.teo@protonmail.com

⁴rowaida.panaiersbo@gmail.com

Abstract: This project explores the application of explainable artificial intelligence, to predict outcomes for intensive care unit patients using medical data from the MIMIC-III dataset. While traditional machine learning models offer high accuracy, their “black box” nature limits clinical trust and adoption. To address this, one approach is to decompose complex model predictions into interpretable components through ANOVA, followed by feature selection using LASSO to isolate statistically significant partial responses. The partial responses can be used to interpret the predictions of the black box model directly or, as a second step, they can be used to train a more constrained model that offer transparency with minimal performance tradeoff. The goal is not only to predict mortality risk, but to also provide interpretable insights that support critical care decision-making. The balance between performance and interpretability is essential in medical environments, where understanding the “why” behind predictions can directly influence patient outcomes. Previous findings have pursued similar approaches, notably Lisboa et al [3]. This paper aims to confirm their results, as well as comparing it to a new implementation variation: Lebesgue instead of Dirac measurements.

1. Introduction

Black box models have shown impressive performance in medical fields, but there is a need to explain the results. This project is about benchmarking a new version of a previous explainable method, presented by Lisboa et al [3]. The method that this project evaluates differs in its implementation – utilizing the more computationally heavy Lebesgue method instead of the Dirac method. Our goal is to evaluate how this alternate methodology differs in performance as measured

primarily by the AUC-metric (Area Under receiver operating characteristic Curve). The interpretable model architecture utilized in this paper is named PRISM - Partial Responses in Structured Models, the code of which is not yet released to the general public. The principal aim is to make the architecture readily available, but it is not our intellectual property and thus the extent to which details of it can be revealed is limited.

Whilst the primary focus of the investigation is to achieve state of the art performance on a binary classification task – classifying in-hospital mortality within 48 hours of admission on the MIMIC-III dataset of Intensive Care Unit (ICU) stays – the methodology utilized is a general approach and not restricted to medical prediction solely. It could prove useful in other contexts, where explainability is an important objective or requirement. One such example where the explainability of the PRiSM-architecture could be advantageous is in financial services in which anti-discrimination policy requires models to be interpretable.

Explainability is crucial if Artificial Intelligence (AI) is to be used in healthcare. The aim is not for AI to replace professional evaluation in the decision making process, but for the professionals to have an additional tool when making certain decisions. An assisting tool’s guidance is, however, of little use if it is unknown what is driving the AI-tool’s classification process.

1.1 Limitations

Since the code used in the Lisboa paper has not been published, it was very difficult to replicate an identical patient filtering, prior to training the machine learning model. The pre-filtering is thus solely based on the brief description in “Mimic III data” [3]. Differences in pre-filtering will cause differences in the results compared to the Lisboa paper.

This report will not include any code or any confidential data. The PRiSM MIMIC-code repository is private and should not be shared. Additionally, the MIMIC-III dataset is confidential and requires certification through training to access.

2. Theory

In this section, we outline the theory behind the model, focusing on anchored ANOVA decomposition, generalized additive models (GAMs), and sparsity/feature reduction via LASSO.

2.1 Explanation of Terms

To understand this report, it is essential to understand some common terms that are used throughout this paper and in Machine Learning (ML) in general.

Partial Response Refers to how the output of the model changes, when a single feature or bivariate feature combination is swept across its range, keeping other features fixed or averaged.

PRiSM Partial Response in Structured Models – the name of the utilized architecture.

LASSO Stands for Least Absolute Shrinkage and Selection Operator. It reduces the model complexity by penalizing weights in proportion to their absolute value. This drives some weights to zero and these features are thereby ignored.

AUROC Area Under the Receiver Operating Characteristic Curve (sometimes simplified as AUC). It’s a widely used metric to evaluate the performance of a binary classifier. It is

especially useful when dealing with sparse datasets. A value of 1 means perfect classification, whereas a value of 0.5 suggests performance equivalent to random guessing.

2.2 ANOVA Decomposition

ANOVA is a decomposition method used to analyze how input variables contribute to the output of a multivariate function. It expresses a function

$$f(x) = f_0 + \sum_{i=1}^d f_i(x_i) + \sum_{i < j} f_{ij}(x_i, x_j) + \dots + f_{1,2,\dots,d}(x_1, x_2, \dots, x_d) \quad (1)$$

as a hierarchical sum of lower-dimensional components. Each term quantifies the effect of specific variable subsets and their interactions.

ANOVA is usually performed anchored or unanchored. Dirac is an anchored type of ANOVA decomposition which is based on fixing (anchoring) a reference point—typically the mean of the input distribution—around which variations are measured. This reduces the computational cost and improves interpretability, especially in high-dimensional settings.

Lebesgue is an unanchored variant of this where instead of fixing the variables that are not under study, they are integrated out with respect to their probability distribution. This is described in detail by Lisboa et al [2].

In this work and Lisboa’s paper [3], the decomposition is restricted to first-order and second-order terms (i.e., individual variables and their pairwise interactions) to balance between expressiveness and interpretability.

2.3 Generalized Additive Models (GAMs)

A Generalized Additive Model (GAM) is a regression model where the expected response is modeled as a sum of smooth functions of the inputs

$$g(\mathbb{E}[Y]) = \beta_0 + \sum_{i=1}^d f_i(x_i). \quad (2)$$

g is a “link function” (e.g., the logit function for classification), β_0 is an intercept term, and each f_i is a univariate or bivariate shape function term from data. GAMs provide high interpretability, as the individual effect of each variable can be visualized and understood independently.

In the PRiSM model, we construct the GAM using the ANOVA-decomposed components f_i and f_{ij} , allowing for transparent/explainable modeling of the largest effects and interactions.

2.4 Sparsity via LASSO

Although the ANOVA decomposition can generate a large number of terms, many may be redundant or uninformative. To mitigate overfitting and promote interpretability, we incorporate sparsity using LASSO (see section 2.1), which adds an ℓ_1 -norm penalty to a loss function

$$z_j = w_{0j} + \sum_{i=1}^C w_{ij} x_i \quad (3)$$

$$\hat{y} = \sigma\left(c + \sum_{j=1}^H v_j g(z_j)\right) \quad (4)$$

$$\mathcal{L}_{\text{BCE}} = -\frac{1}{N} \sum_{i=1}^N \left[y_i \log \hat{y}_i + (1 - y_i) \log(1 - \hat{y}_i) \right] \quad (5)$$

$$\min_{w,v} \left\{ \mathcal{L}_{\text{BCE}} + \lambda \sum (|w| + |v|) \right\} \quad (6)$$

where C is the number of input components, H is the number of hidden nodes in the network, σ is the sigmoid activation function, g is the hidden layer activation function, w_{ij} and w_{0j} are the input layer weights and bias, and v_j and c , are the hidden layer weights and bias. Equation 3 and 4 describe the one-hidden-layer neural network, Equation 5 shows how the binary cross entropy loss is calculated, and Equation 6 shows how LASSO is used when minimizing the loss.

This regularization technique drives some coefficients $\beta_{ij,i>0}$ to zero, effectively selecting only the most relevant terms in the model. Note that the intercepts - β_{0j} , $\forall j$ and ω_0 - are not contributing to the final loss minimized over. The degree to which β -coefficients are driven to zero (rendering features or feature combinations redundant) can be tuned by the λ hyperparameter. The result is a sparse representation that enhances both generalization and interpretability.

2.5 Explainability and Clinical Relevance

By combining anchored ANOVA decomposition, GAMs, and LASSO regularization, a predictive model - that is inherently interpretable - is constructed. Each prediction can be traced back to contributions from specific variables or their interactions, making the model suitable for healthcare, where explainability is crucial.

3. Method

To have access to the medical data from a well-known benchmark dataset from Physionet, it was required that we finish a few basic CITI-Program courses:

- Conflicts of Interest
- Human Research – Data or Specimens Only Research

Once access was gained to the MIMIC-III dataset, preprocessing of the data was initiated. The goal of the preprocessing was not to achieve the best possible performance but instead to replicate the procedures followed by Lisboa et al in an effort to make results comparable.

The dataset contained records of 34,472 unique ICU patients, and to filter them our group used the same approach as described in the Lisboa study [3], that itself relied on methodology presented by Arutyunyan, H., Khachatrian, H., Kale, D.C. et al. [1]. The Lisboa paper only discussed the preprocessing very briefly, and our methodology may differ.

Variable	Mean	Std
Diastolic blood pressure	✓	✓
Glasgow coma scale total	✓	✓
Glucose	✓	✓
Heart Rate	✓	
Oxygen saturation	✓	✓
Respiratory rate	✓	
Systolic blood pressure	✓	✓
Temperature	✓	
Weight	✓	

Table 1. List of variables extracted from the MIMIC-III dataset.

3.1 Preprocessing Steps

1. Only data from the first 48 hours of the ICU stay was retained.
2. The same features as was used in the previously mentioned study was extracted.
3. To resemble the preprocessing made previously in the Lisboa paper it is necessary to filter out subjects depending on their "30 % level of missingness" [3]. The definition of missingness is not explained in the paper and unclear what the 30 % corresponds to. The preprocessing in the Lisboa paper resulted in a dataset of 7532 patients, which led us aim for a similar size. The approach in this project was to iterate over the patient time series and calculate the overall degree of missing values for a patient. This was done by comparing the amount of NaN values to actual values. This would then result in a new matrix containing every patient, the extracted features and the features corresponding missingness percentage ranging from 0 to 1.
4. Transform features that are considered "static". We assumed the weight as static. In practical terms, if weight has been measured once during the 48 hour stay, the missingness value for that feature would be transformed to 0.
5. Now that we have a matrix with corrected missingness levels, we calculate an overall missingness level for a specific patient. This is done by calculating the column sum of each feature's missingness for a given patient and dividing by missingness to get the mean feature missingness for the patient.
6. To match the desired amount of patients, we calculate the corresponding missingness threshold. This new threshold was calculated to be 48.7 %, while Lisboa et al. used a threshold of 30%. With the new missingness threshold, we can filter out all patients above the threshold and keep the remaining 7532 patients.
7. Now the data has been preprocessed and we have achieved indexing on which patients should be passed on to the next step.

3.2 Machine Learning

The objective of the machine learning analysis was to predict in-hospital mortality using clinical variables collected during the first 48 hours of ICU admission. After preprocessing and filtering, the dataset comprised approximately 7,500 ICU stays with reduced missingness and harmonized variable structure.

Data Splitting The dataset was randomly split into three elements according to the Lisboa papers – a training dataset (60%) and a validation dataset (20%) – which are used for model estimation and optimization. The third dataset is the test set (20%).

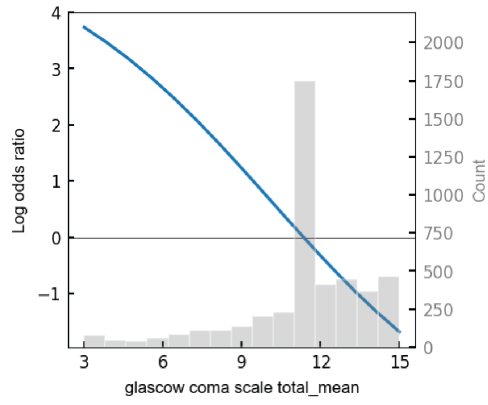
Feature Selection and ANOVA Decomposition To understand the contribution of each variable to the outcome, we performed an ANOVA-decomposition (see section 2.2) prior to training the model. This approach helps quantify the amount of variance in the dependent variable (mortality) that can be explained by each individual predictor.

Black-Box training, PRN construction and LASSO Although the all details of PRiSM will not be covered extensively, the workflow consists of training a black-box model, in our case an MLP with ten hidden nodes and hyperbolic tangent activation functions. The MLP is trained using early-stopping based on val-set performance. The next, second, step is to calculate the partial responses, using ANOVA decomposition as described in [2]. The third step is to reduce the number of partial responses corresponding to each and every component (univariate or bivariate features) by LASSO-regression, as explained in 2.4. One thing to note is the heuristic used to pick the λ : use the largest possible λ that achieves a score of at least 99.5% AUC on the validation set. When steps one to three have been completed the steps two and three are repeated yielding the final model. The approach is constructed in order to sacrifice very little performance whilst drastically reducing model complexity in addition to producing partial responses aiding explainability.

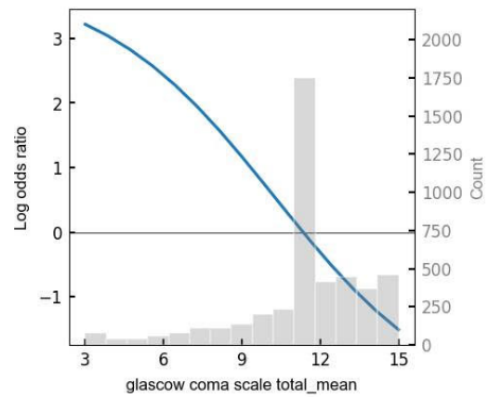
Implementation All machine learning experiments were implemented in Python using Prism Mimic. Model evaluation and visualization were conducted in Jupyter Notebooks. The confidential PRiSM code will be released as a public Python package in the future.

4. Results

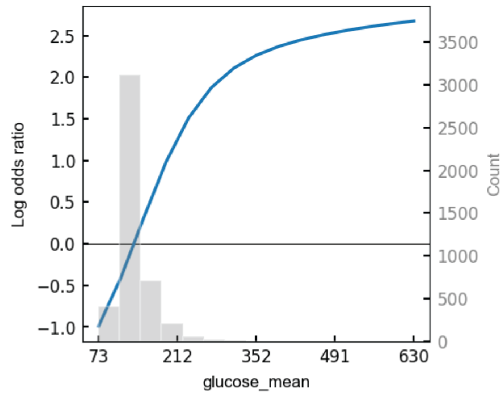
Out of the original features and their second order combinations, only seven features remained after the second LASSO-regression. None of the second-order components withstood the selection. The generated partial responses are presented in Figure 2. Performance of these partial responses in the classification is presented in Figure 3, as well as Table 2.



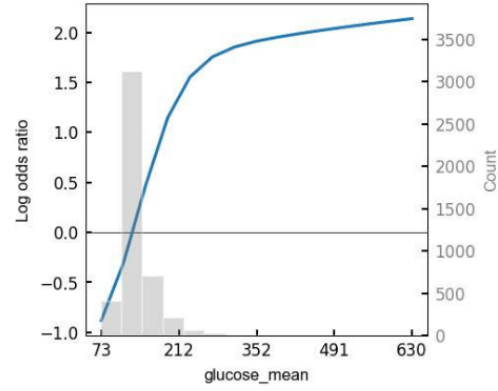
(a) Lebesgue: Glasgow coma scale



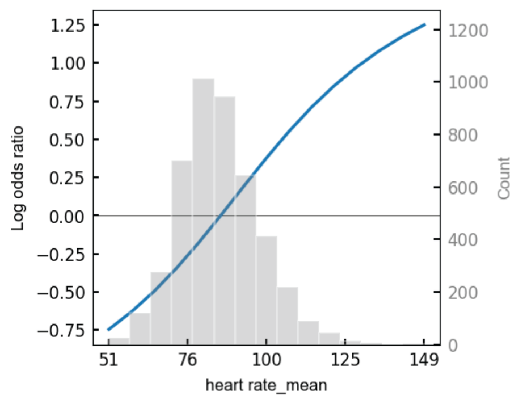
(b) Dirac: Glasgow coma scale



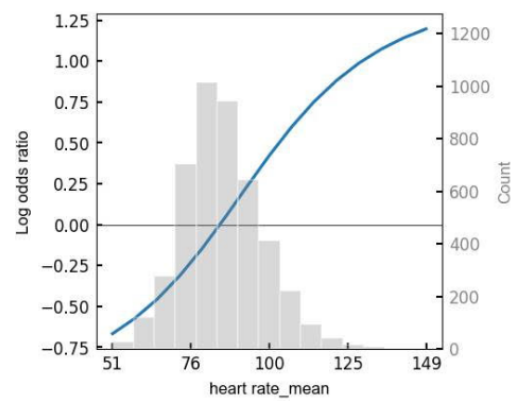
(c) Lebesgue: Glucose



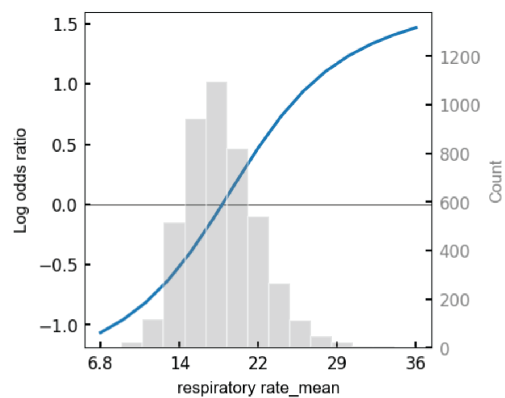
(d) Dirac: Glucose



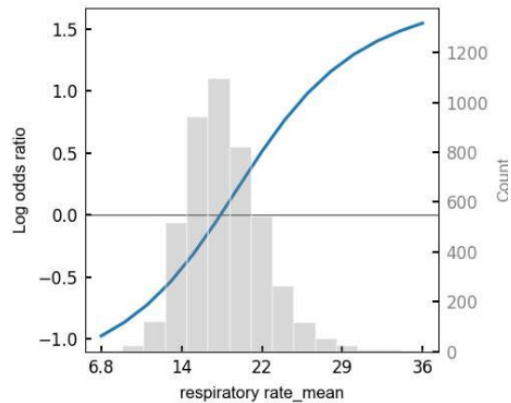
(e) Lebesgue: Heart rate



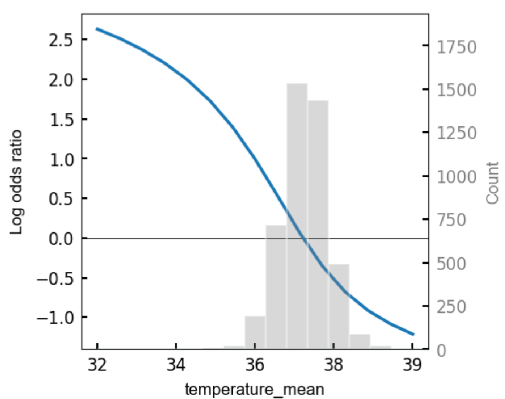
(f) Dirac: Heart rate



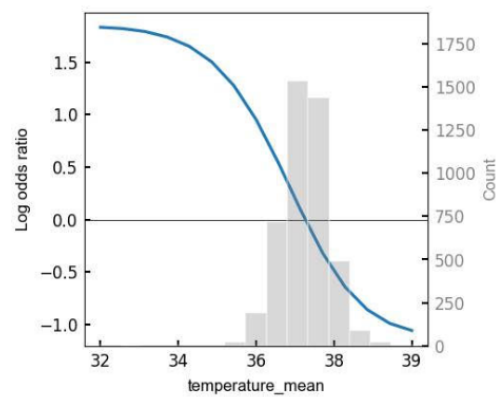
(a) Lebesgue: Respiratory rate



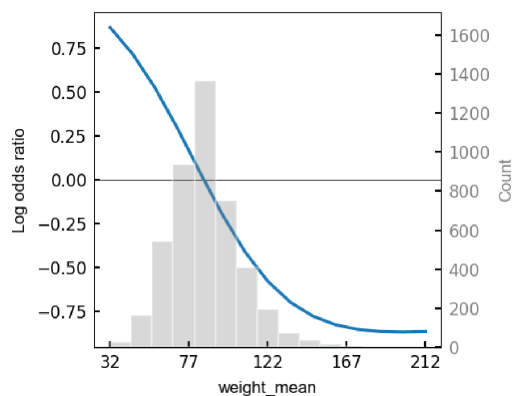
(b) Dirac: Respiratory rate



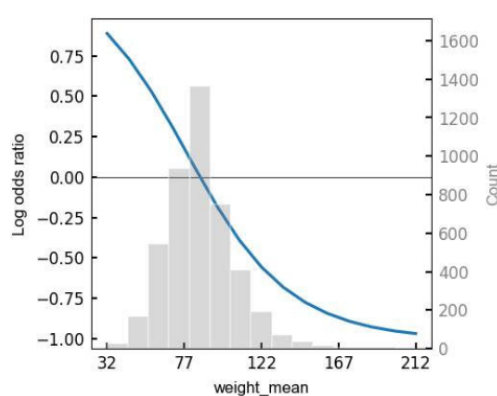
(c) Lebesgue: Temperature



(d) Dirac: Temperature



(e) Lebesgue: Weight



(f) Dirac: Weight

Figure 2. Comparison of partial response plots using Lebesgue and Dirac methods across selected clinical features (Part 2).

Method	Model	C	Train AUC	Validation AUC	Test AUC
Lebesgue	MLP	14	0.908 (0.890–0.921)	0.900 (0.877–0.927)	0.871 (0.841–0.897)
	PRN	18	0.860 (0.843–0.870)	0.873 (0.855–0.896)	0.863 (0.823–0.896)
	PRN-LASSO	7	0.855 (0.844–0.870)	0.871 (0.856–0.891)	0.863 (0.833–0.879)
Dirac	MLP	14	0.908 (0.892–0.915)	0.900 (0.880–0.920)	0.871 (0.841–0.897)
	PRN	10	0.858 (0.845–0.879)	0.876 (0.854–0.904)	0.863 (0.817–0.877)
	PRN-LASSO	7	0.855 (0.845–0.871)	0.873 (0.855–0.892)	0.862 (0.841–0.892)
Lisboa paper	MLP	14	0.828 (0.805–0.850)	0.810 (0.769–0.852)	0.815 (0.777–0.853)
	PRN	12	0.810 (0.786–0.833)	0.799 (0.756–0.841)	0.807 (0.769–0.845)
	PRN-LASSO	12	0.811 (0.787–0.834)	0.797 (0.755–0.840)	0.812 (0.774–0.850)

Table 2. Combined classification results across all experimental settings. C: number of components. Values in parentheses indicate 95% confidence intervals.

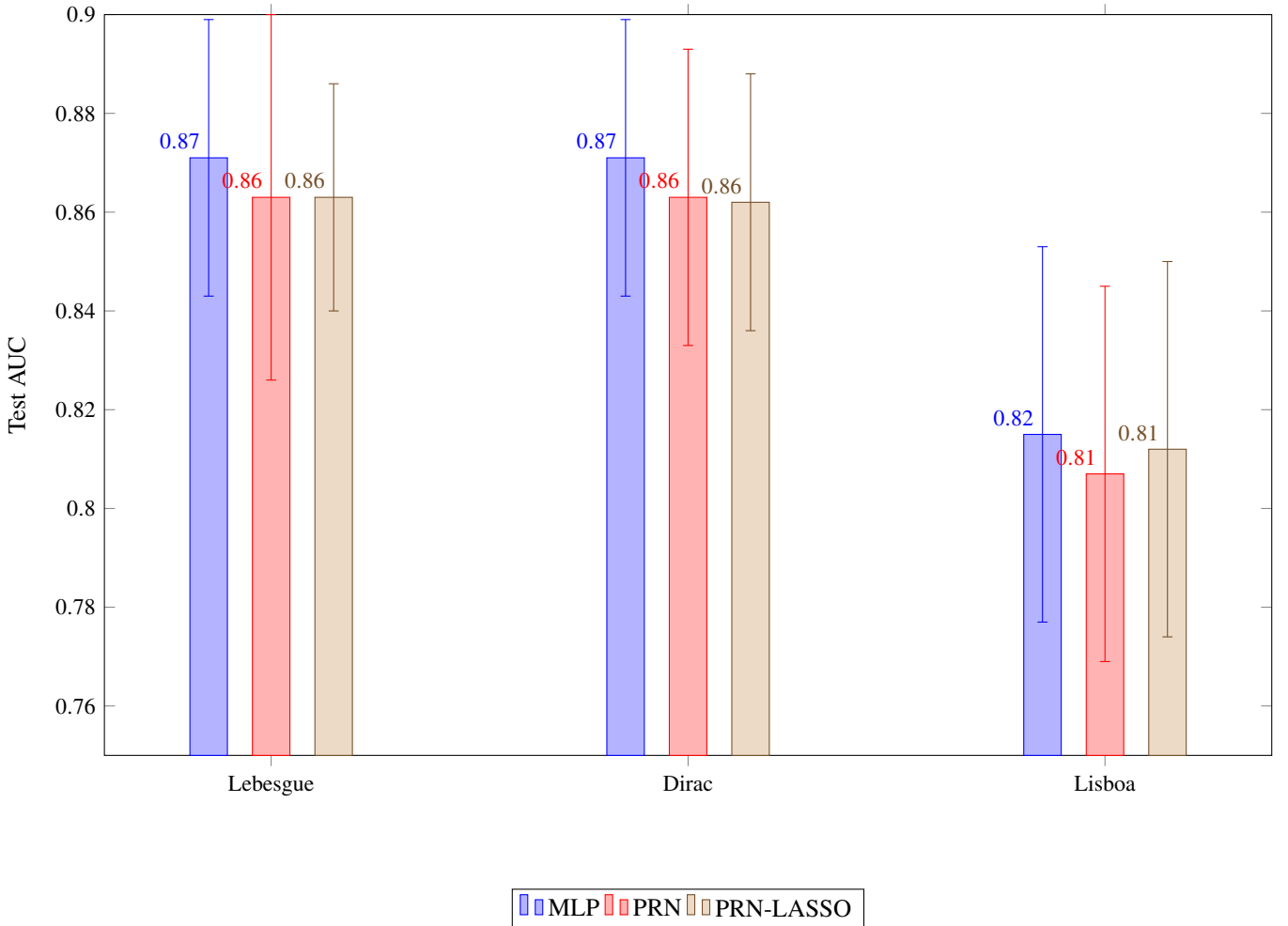


Figure 3. Test-set AUCs (bars) with 95% confidence intervals (error bars) for each model under the three experimental methods. Number of parameters in the models vary and are presented in Table 2

5. Discussion

The results show no significant difference in end result between using the Lebesgue or Dirac variation of the ANOVA decomposition. While this was not expected, it might be because of the mean values being sufficiently good at representing the underlying distributions in the data. This would however not be the case if the distributions were less concentrated around the mean, and in such a case the Lebesgue method could potentially result in more accurate partial responses. To draw conclusions, further study of this method applied on datasets with skewed distributions is needed.

Second-order partial responses, although none are present in this report, may be represented as a heatmap showing the log odds ratio's level curves as it relates to a pair of features on the x and y-axis. The interpretation of the partial responses is simple, yet still not as easy as the final (not by us revealed) end-user product.

The first partial response of GCS (Figure 2) has a monotonically decreasing log-odds contribution (in blue) from the variable "mean of the Glasgow coma scale" as the coma scale goes from "3" (in a coma) to "15" (widely awake). The horizontal line at log odds ratio zero indicates where - over the range of the considered variable - the mortality odds are unaffected. In the case of the Glasgow coma scale we see this intersection at a score of 11. On the left side of this intersection, when the score is below 11, an increased log-odds ratio is seen indicating that this feature contributes towards an increased predicted probability of a positive prediction (death before discharge). On the other hand, an increase in degree of consciousness indicated by a measured score of 12 or above on the coma scale, the model reasons this to be indicative of a decreased probability of an adverse outcome.

A side-by-side inspection of our plots with those reported by Lisboa et al. reveals similar patterns, providing evidence that we have replicated their methodology with a quite high degree of accuracy.

By having access to these types of plots, a clinician can confirm whether the plots are reasonable or not. Considering again the GCS, the clinician most probably would agree that an increase in consciousness decreases the probability of an adverse outcome and vice versa, yet the plot still serve a useful purpose. While most clinicians would agree on the general effect, they likely lack a precise understanding of how much a specific score, for example 12 compared to 7, alters the probability of an adverse event.

In the background of the partial response plots a histogram of ICU-stay patient records lies. An artifact of the method utilized by Lisboa et al, namely mean imputation, can be seen most directly in the Glasgow coma scale plot where one bin size stands out - the imputed one at score 11. Further exploration of alternative imputation methods could potentially improve the overall results.

Similarly to the log-odds score of the Glasgow coma scale, the other plots can be interpreted as to their contribution to the final log-odds and their results summed up. The formula for aggregate log-odds to classification probability decrease/increase is not as simple and easy to interpret as the end-user product, but do contain all the necessary information.

Note that the monotonicity is not a general feature of the log-odds ratio curve even though by chance the most significant components turned out that way over their inspected ranges.

In order to enhance the comparability of our results to those in the paper written by Lisboa et al., we applied patient filtering criteria that allowed us to match the amount of patients to that of the reference paper. Since the methodology of preprocessing was not clearly stated in the reference paper and our methodology is not confirmed to be identical, it is hard to draw reliable conclusions when comparing our results with those reported by Lisboa et al.

6. Conclusion

As have been shown, the performance presented by Lisboa et al. [3], is plausible given that we achieved similar albeit slightly higher AUC-scores with our re-implementation. The difference of results is most likely due to the dataset: Our group tried to yield the same records of ICU-stays but we can only confirm that the same fraction of mortality and the same amount of records was filtered out. These properties could hold, yet the records themselves could still be different and this constitutes the most likely cause of dissimilarity. Furthermore, this investigation did not show a performance improvement of the Lebesgue method compared to the much simpler Dirac measurement, as illustrated in Figure 3 and Table 2.

References

- [1] H. Harutyunyan, H. Khachatrian, D. C. Kale, G. Ver Steeg, and A. Galstyan. "Multitask learning and benchmarking with clinical time series data". *Scientific Data* **6**:1 (2019), p. 96.
- [2] B. Walters, S. Ortega-Martorell, I. Olier, and P. Lisboa. "How to open a black box classifier for tabular data". *Algorithms* **16** (2023), p. 181. DOI: [10.3390/a16040181](https://doi.org/10.3390/a16040181).
- [3] B. Walters, S. Ortega-Martorell, I. Olier, and P. J. G. Lisboa. "Towards interpretable machine learning for clinical decision support". In: *2022 International Joint Conference on Neural Networks (IJCNN)*. 2022, pp. 1–8. DOI: [10.1109/IJCNN55064.2022.9892114](https://doi.org/10.1109/IJCNN55064.2022.9892114).