National Taiwan Normal University/

Ostfalia UAS

Intelligent Humanoid Robotics

Assignment 2
Rock Climbing

Name: Philip Liebscher
Student ID: 91499142X
Name: Kevin 常凱文
Student ID: 11109356A
Name: Mohamed Aziz Laarbi
Student ID: id660931
Name: Ala Rejeb
Student ID: id315337

15 December 2025

Lecturer
Distinguished Prof. Hansjoerg Jacky Baltes

Prof. Dr. Reinhard Gerndt

I. INTRODUCTION

The goal of this assignment was to realise a humanoid rock-climbing robot. From designing the robot and setting up a simulation environment to detecting climbing holds in camera images, planning a climbing route to the top and executing the climbing motions.
To solve this rather complex task we work together as an international group of NTNU and Ostfalia students.
This report shows our results, explains our implementation and how we addressed the challenges in each part of the assignment. We also evaluate our results, discuss shortcomings and present ideas for further improvements.
The Code for the simulation can be accessed through this repository:
https://github.com/sibbl08/Climbing-Robot

II. IMPLEMENTATION

Designing the Robot

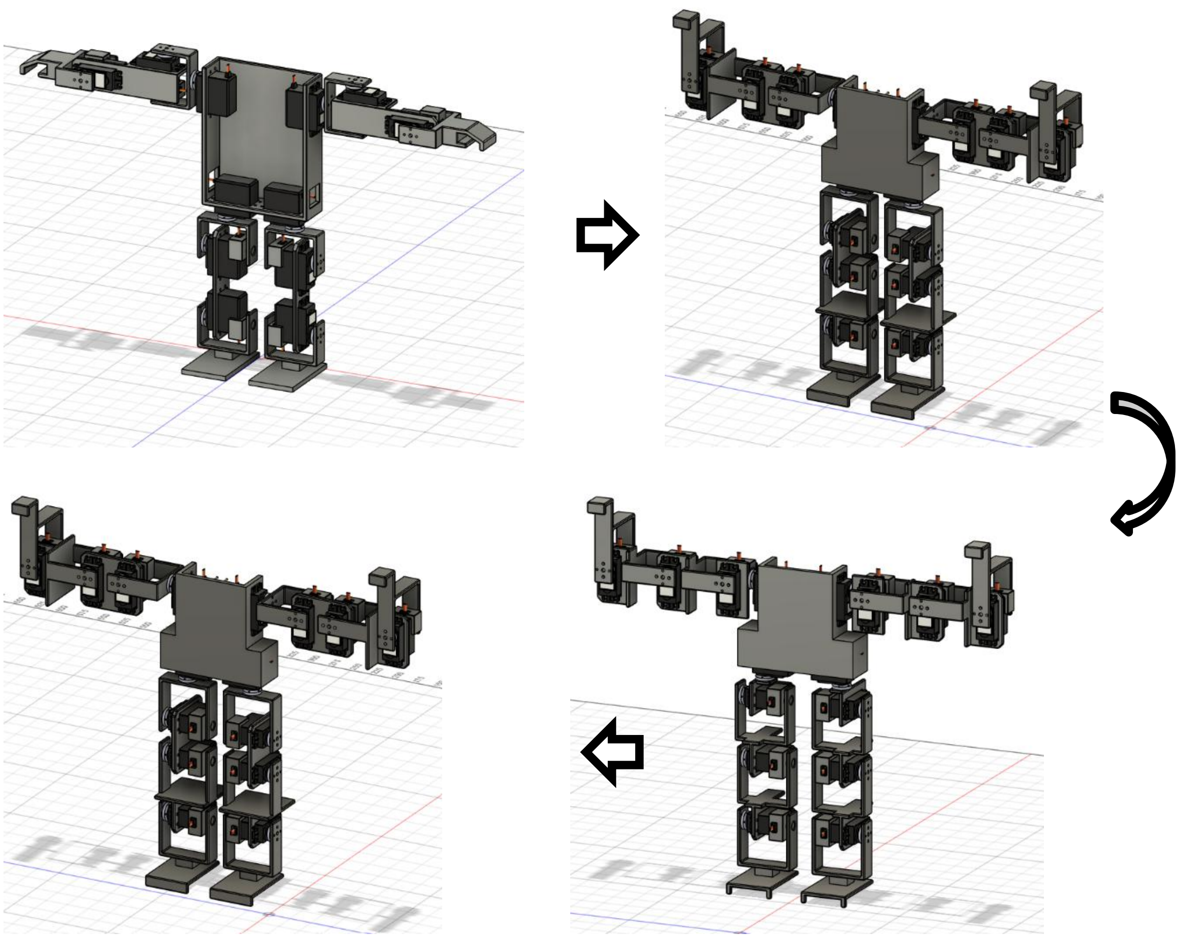


Figure 1: Design Process

Since we had never designed a humanoid robot before, this turned out to be a much harder task than expected. We went through many different robot versions until we came up with the final design. Figure 1 shows some of the different designs we created on the way to our final version.

This humanoid robot is specifically designed to fulfil the climbing task. Therefore, it is physically not able to perform any other tasks (which is not necessary in our case). We oriented ourselves on human climbing motions to figure out the joint configuration. As Figure 2 shows, when humans climb a straight wall, we often rotate our hips outward, bend our hips and knees, and slightly rotate our feet towards the wall to stand on climbing holds. This motion is sufficient for our simple climbing routes and therefore we decided on a hip_yaw, hip_pitch, knee_pitch, and ankle_yaw joint configuration for the legs. This joint configuration allows our robot to perform the motions shown in Figure 2.
The same applies to the arms. When human climbers want to reach holds that are above or to the side of their bodies, they move their arms parallel to the wall and then reach for the holds. Translated into our robot joints, this corresponds to shoulder_yaw, shoulder_pitch, elbow_pitch, and wrist_pitch. The wrist_pitch is necessary to fix the orientation of the robot's hand so it can grip the holds properly.
These joint configurations often include multiple joints in the same plane, which may seem odd. However, during the design process I noticed that climbing a straight wall is, most of the time, essentially a 2D problem. First, the end effector must be moved away from the wall so it can move freely. Next, it is positioned in 2D space to the correct location, and finally it is moved back towards the wall to grab the hold.
The hands are designed as hooks, which makes gripping much easier. The feet also have small hooks so they automatically "click" into the holds and do not slide off.
To design the robot we used a CAD program (Fusion360). This had the advantage that we could easily 3D-print all parts of the robot and assemble it. It also allowed us to export a MuJoCo .xml file to bring the robot into the simulation.

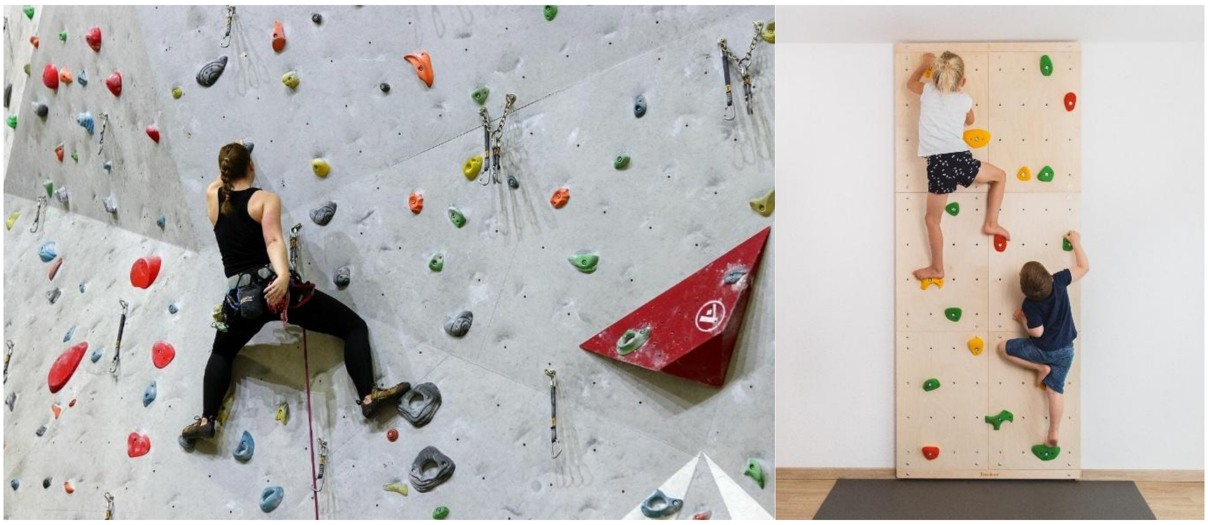


Figure 2: Humans Climbing

Setting up the Simulation

We used MuJoCo as the simulation software for this project. For the wall and the holds, we used the files that were provided and implemented them into our simulation. As a first climbing route, we set up *route_ladder_V1.xml*, where the climbing holds are arranged in a ladder-like pattern. This is an easy climbing route and serves as a starting point. Later on, we can construct harder routes to further test our robot.
Because we designed the robot in a CAD program, we were able to export it in MuJoCo .xml format. Before implementing it into the simulation, we had to make minor adjustments, such as

adding the position actuators and setting the root body as a freejoint. *Robot_V3.3.xml* is also the main XML file where we import the other parts of the simulation (wall, floor, route).

To start the MuJoCo simulation, we use the Python mujoco.viewer. To initialize the robot, we use *init_robot.py* to set the robot's position and the initial joint angles. This initialization allows us to have a defined starting position. Figure 3 shows the complete MuJoCo simulation at the beginning.

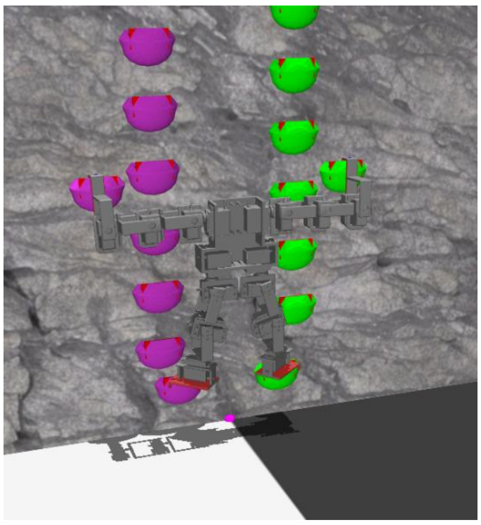


Figure 3: Mujoco Simulation

To make the robot actually collide with the holds, wall, and floor—so the robot can grab and stand on them—we had to implement collision modelling. For the wall and the floor this was easy: we only had to set *contype* and *conaffinity* to 1. For the climbing task, the only important collisions are the hands with the holds/wall and the feet with the holds/wall.
To model these collisions, we added simple geometries into our XML files. Figure 4 shows the simple geometry we added as collision boxes, displayed in red. We added one collision box for each hold and three collision boxes for each hand/foot.

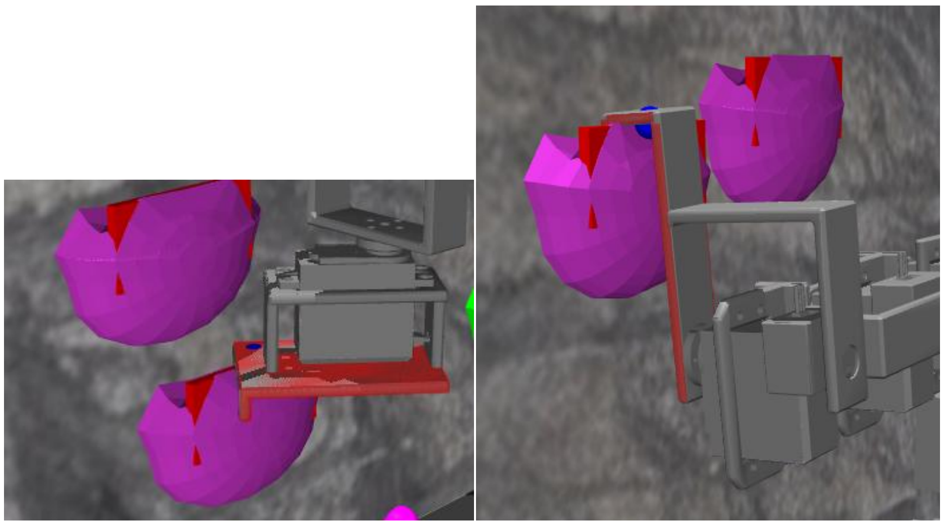


Figure 4: Collision Modelling in Mujoco

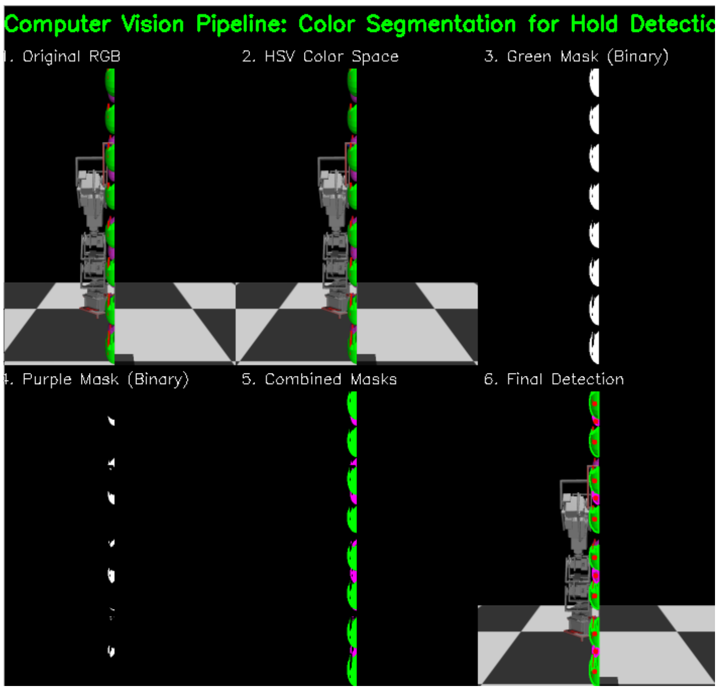
Implementing Forward and Inverse Kinematics

To set up the forward kinematics, we had to define the actual end-effector positions of our limbs, since the origins of the hands and feet are not the points we need for further calculations. Therefore, we added reference points into the XML (e.g., *hand_r_ref*), which are displayed as blue spheres in Figure 4. We positioned them in locations where we expect our hands and feet to collide with the holds. With these positions, further calculations become easier. These reference points are only positional markers and do not affect the simulation.
As the reference point for all limbs, we set the origin of the chest, which is located at the center of the robot. The translations between the joints and the end-effector reference points, as well as the joint axes, could be read directly from the robot XML file.
The forward kinematics function is implemented in *fk.py*. The function receives all 16 joint angles and the chest coordinates in the world coordinate frame. It then calculates the positions of all four end effectors in world coordinates and returns them.

To compute the joint angles for a desired end-effector position, we implemented an inverse kinematics function. Each limb uses its own reference point as the end effector. Instead of calculating positions from joint angles (as in forward kinematics), the IK solves the opposite problem: given a target position—and for the arms also a target orientation—it computes the joint values that reach that target.
The arms use a dedicated IK routine where reaching the correct 3D position is the primary objective. The orientation around the pitch axis is refined in the Jacobian nullspace, which is important for achieving a good grip on the holds. Joint limits are enforced at every iteration to keep the solution physically valid and prevent self-collisions.
The legs only solve for position, but they include additional nullspace objectives to keep the foot level and to maintain a desired hip yaw angle. Keeping the foot approximately level ensures that the robot can stand securely on the hold. Since the leg kinematics become singular when hip_pitch and knee_pitch approach zero degrees, we check the initial joint configuration and replace it with a stable default if necessary. Joint limits are also applied here.
All IK functions use numerical Jacobians and a damped least-squares update to iteratively reduce the position error. They receive the target position in world coordinates, the target orientation (arms only), the current joint angles, and the chest pose in world coordinates. They then compute the required joint angles and return them together with a flag indicating whether the target was reachable.
The IK dispatcher selects the correct solver for each limb and thereby allows the motion planner to test whether a hold can be reached before executing a movement. Together, *fk.py* and *ik.py* provide the essential kinematic tools used later in planning and execution.

Implementing OpenCV Detection on Rock Grips in Mujoco Simulation

Method uses color segmentation to detect the holds visually by using OpenCV contour, first of all we extract the holds position directly from the mujoco simulation data, it converts the image from RGB to HSV (hue, saturation and value) because color filtering is more robust in HSV. I create a binary mask using *cv2.inRange* and this will define the boundaries lower and upper mask, the output mask is a matrix of 0s and 1s, where 1 is white and 0 is black. At the end this deletes the wall, the floor and the robot from the image, leaving only white blobs floating in bacl space where the holds are.



a. Color Masking

Because in simulation each grip is in the same position, so we are considering to put the camera detection at the right side of the robot for simplicity purpose. The boundaries define the green and purple colour mask in format of np array. Unlike photoshop or the other color implementation tools, numpy uses a compressed values to fit into 8-bit integers (0-255):

green_lower = np.array([35, 50, 50],

green_upper = np.array([85, 255, 255]),

The Hue green boundaries here is approximately 70degree to 170 degree, it is a quite large yellow green to a teal-cyan green to holds every texture. The Saturation ensure the object has actual color because <50 is too close to white or grey. The Value reject the very dark pixel because the whole environment is dark except the floor, wall and robot, so we need to make sure not includde and dark value, and it is accepts the brightest possible pixel

purple_lower = np.array([125, 50, 50]

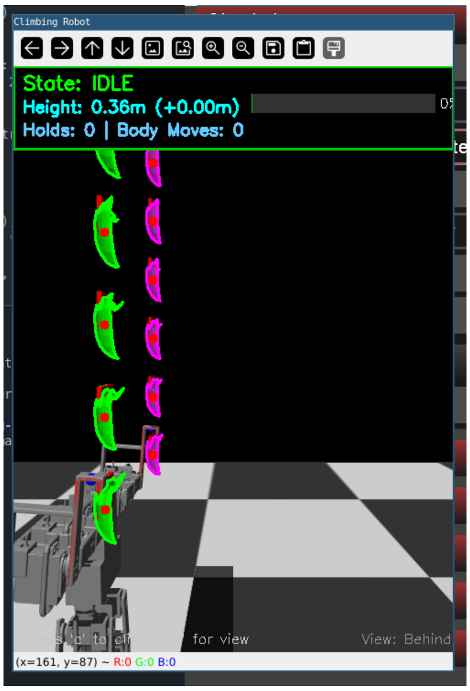
purple_upper = np.array([155, 255, 255])

The Hue purple in standard degrees are approximately 250degree to 310 degree which accept blue-purple-pink, which a wide range of color. The Saturation and Value are identical to the green filter, the logic are the same which to reject any dark or shadow color.

After the ensure the color has in the correct range, next we remove some noise by filter a small area. *if area < 100*: We calculate the area in pixels. If a shape is smaller than 100 pixels, it is considered too small to be real climbing hold and is ignored, so that this prevents from the noise because there are area not fully detected by the color masking and this will happen even we have defined the correct range of color.

b. Detection

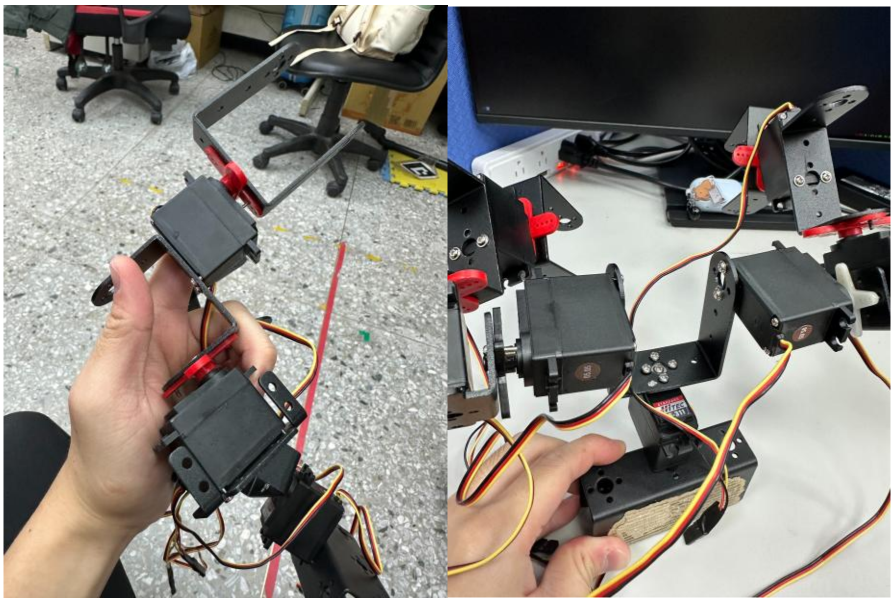
Cv2.findContours is the way we apply the detection or the bounding box through the masked color. We loop through each contour detected area and *M = cv2.moments(contour)* the robot needs a specific point to target, like the center of the hold, not just the outline, this code uses image moments(contour). At the end we visualize the detection as a result.



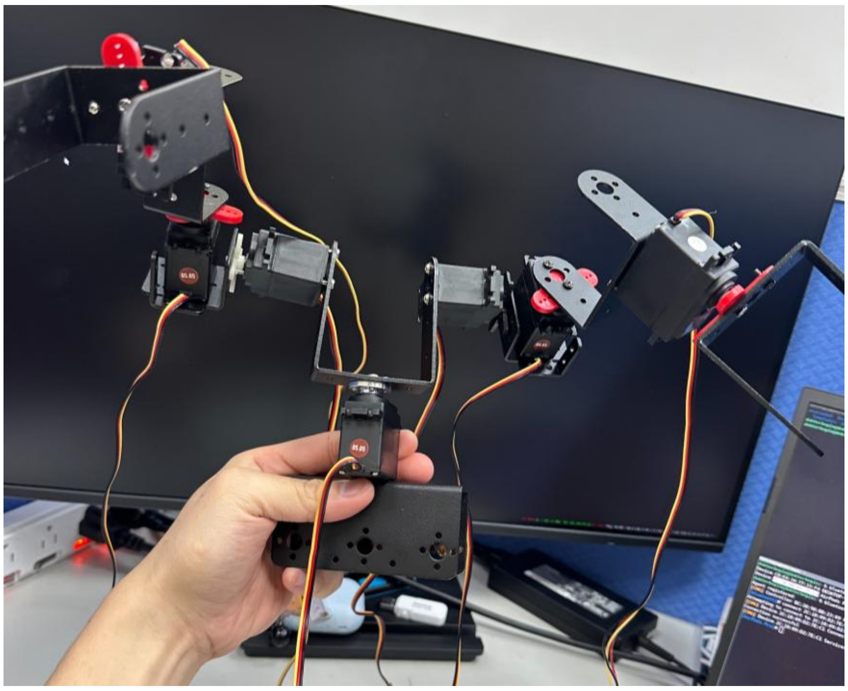
The result is showing in picture above, where the grips of the rock climbing are detected correctly and I'm even test this grips detection with a simple robot movement from the bottom to the peak of wall and all the holds are detected.

Construct The Real Robot

The Robot was constructed with Aluminium Servo Brackets, and HS-311 HiTEC servo, the aluminium is provided by our senior and the servo itself is provided by Professor Jacky. I got the material like screw and any other tools from ERC lab NTNU.



Left and right hand contain the same material and same length, approximately 20cm and the length of body approximately 15cm. It has 180 degree of freedom from all servo.



It also contain a hip that can moves left or right in the same 180 degree of freedom, the end effector have not ready yet because we are out of material to build the real end effector like a gripper that can grab something but that is our future goal. The arm could move in every possible direction like from left to right, up to down. We call each link as a *L1: shoulder_pitch_link, L2: arm_yaw__link, L3: roll_wrist_link, and the last hip_yaw_link.*

Hold Parsing from XML Files

In this project, I implemented the parsing of climbing holds directly from XML route files. The goal was to avoid relying on computer vision and instead use a predefined climbing route available in simulation.

The XML parser extracts all hold positions and metadata, then separates them into:

- **start holds** (initial contact points for hands and feet),
- **route holds** (holds to be used during the climb).

This structured representation allows the rest of the pipeline to work with clean and well-defined inputs.

Climbing Path Planning Algorithm

I implemented a climbing path planning algorithm that generates a sequence of movements for the robot.
The planner iteratively selects which limb should move next and assigns it to the next reachable hold above its current position.
The planner maintains an internal `RobotState` that tracks the current hold of each limb.
At each step, the state is updated after a movement is planned, allowing the algorithm to generate a complete climbing plan step by step.
The output of this module is a list of tuples `(limb, old_hold, new_hold)` representing the full climbing sequence.

Inverse Kinematics Integration

I integrated the inverse kinematics module into the planning pipeline.
For each planned movement, the target hold position is passed to the IK solver to compute the required joint angles.
To ensure compatibility between the planner and the IK module, I implemented a mapping between planner limb names (`right_hand`, `left_hand`, etc.) and IK limb identifiers (`right_arm`, `left_arm`, etc.).
This integration allows each planned movement to be translated into a valid joint-space command.

Motion Planning Pipeline

I designed and implemented a complete motion planning pipeline that connects all components:

Holds → Path Planner → Inverse Kinematics → MotionStep generation

This pipeline is implemented in `pipeline.py` and produces a sequence of `MotionStep` objects, each containing:

- the limb to move,
- the joint indices,
- the target joint angles,
- the motion duration.

This modular structure makes the pipeline easy to test independently from the simulation backend.

Generation of MotionStep Instructions

I implemented the conversion from IK results to executable `MotionStep` instructions.
Each `MotionStep` contains all information required by the simulator to execute a single limb movement.

The output of this step is a clean list of motion instructions that can be directly consumed by a MuJoCo execution loop.

Identification and Preparation of Simulation Outputs

A detailed analysis was conducted to identify all the simulation outputs required for the execution and evaluation of the planned task. This preparatory phase was essential to ensure that the simulation provides sufficient information for both real-time monitoring and post-processing analysis.
The selected outputs cover the robot's kinematic, dynamic, and interaction-related variables, enabling a comprehensive understanding of the system behavior. Special attention was given to the consistency and accessibility of these outputs within the MuJoCo framework.
The outputs were structured to allow efficient retrieval during simulation execution as well as systematic logging for offline evaluation. This preparation ensures that the simulation results can be directly reused for debugging, performance assessment, and future extensions of the control strategy.

Completion of MuJoCo Configuration

The configuration of the MuJoCo simulation environment was fully completed prior to starting the execution phase. This included installing and configuring the MuJoCo framework, loading the humanoid robot model, and verifying that all robot description files (XML, meshes, and assets) could be correctly parsed by the simulator.
Validation checks were performed to confirm proper model loading, correct initialization of joint states, and reliable interaction with the simulated environment. Special attention was given to understanding the mapping between the robot's kinematic structure and its simulation representation. This finalized configuration provides a stable and reproducible basis for subsequent simulation experiments.

Design of the MuJoCo Execution Interface

To support structured execution of the planned motions, a dedicated MuJoCo execution interface was designed and implemented. This interface acts as a bridge between the high-level control logic and the simulation engine.
The interface was designed with modularity in mind, enabling future extensions without requiring changes to the core simulation setup. It manages simulation resets, state updates, actuator commands, and data acquisition at each simulation step. By centralizing these operations, the interface improves clarity, robustness, and repeatability of simulation runs.

Generation of MotionStep Instructions

A key part of the simulation preparation was the definition and structuring of the MotionStep abstraction, which represents executable robot motions. This abstraction provides a clean interface between the inverse kinematics output and the simulation execution layer.
Each MotionStep contains joint indices, target joint angles, and execution duration for a specific limb movement. This design decouples motion planning from low-level simulation control and enables clean integration with MuJoCo.

III. EVALUATION OF THE RESULTS

Challenge 1 – Physical Robot Construction (Bonus)

Due to a lack of time, we were not able to transfer our robot model from CAD into the real world. Nevertheless, we want to discuss our general design here, because with the available CAD files it should be relatively easy to 3D print and assemble the entire robot in real life. We would only need a little more time to complete this task.
We are satisfied with the overall design. It is not universal, but we believe it is sufficient for the climbing task. Despite the generally human-like appearance of the robot, the design still slightly deviates from human motion capabilities. For example, we added a wrist pitch joint, which is important for climbing in order to always achieve the optimal orientation of the hand and obtain a secure grip on the climbing holds. This joint has a range of motion from −90 to +90 degrees, whereas a human wrist can only move approximately from −40 to +40 degrees in this plane. The same applies to the ankle yaw joint. A human foot can only yaw about −40 to +40 degrees while keeping the knee and hip locked in position. Our ankle yaw joint has a wider, non-human range of motion, which is important for our climbing task.
Overall, the design is sufficient for our application. Although it deviates from human motion abilities, these deviations are necessary to simplify the control of the robot.

Challenge 2 – Rock Climbing Robot Simulation

Since we had never worked with MuJoCo before, it took some time to become familiar with it and to start working on the tasks. The result is satisfying and meets our expectations. Modeling the robot in CAD was not strictly necessary, but it made it very easy to transfer the robot model into the MuJoCo simulation. With only minor adjustments, we obtained a nearly perfect virtual twin of our robot in MuJoCo.
For collision modeling, we used simple collision boxes. Since collision boxes are implemented only for the holds, hands, and feet, the rest of the robot cannot collide with the simulation environment. This could be improved in the future to achieve an even more realistic simulation by using tools such as CoACD. More complex meshes could be decomposed into a collection of convex shapes using a tool like CoACD.
We experimented with this approach, but found that it required too much effort and was not necessary for our use case. In an ideal climbing scenario, only the feet and hands should collide with the wall. To achieve this behavior, these collisions must be handled through control of the robot. Therefore, collision modeling for the rest of the robot is not required.
One potential improvement for the future would be to determine the actual torque limits of the servo motors and implement them in the simulation. Currently, we set the force range of all actuators to a common value, which does not accurately reflect the real-world behavior of the servo motors. To make the simulation more realistic, this should definitely be implemented.

Challenge 3 - Route Detection

For route detection, two complementary approaches were implemented. First, climbing holds can be parsed directly from predefined XML route files provided in the simulation environment.

This allows reliable access to hold positions and metadata without relying on visual perception and serves as a robust baseline for planning. In addition, a camera-based route detection method using OpenCV was developed to detect climbing holds visually in the MuJoCo simulation. By applying color segmentation in the HSV color space and contour detection, the system is able to identify and localize holds while filtering out the wall, floor, and robot. Although the XML-based approach was primarily used for planning due to its reliability, the vision-based detection demonstrates the feasibility of a perception-driven pipeline and provides a foundation for future extensions toward more realistic scenarios.

Challenge 4 - Motion Planning

Climbing Path Planning

The climbing planner successfully generates a complete climbing plan for the predefined route. For the provided ladder route, the planner produces a sequence of 12 climbing steps, covering both hands and feet movements.
The generated plan is consistent and deterministic and respects the predefined movement order.

Inverse Kinematics Integration

The inverse kinematics integration produces valid joint angle solutions for arms and legs when the target holds are within reach.
The IK solver achieves a position accuracy of less than 1 mm for reachable targets. The mapping between planner limbs and IK limbs works correctly, allowing seamless communication between modules.

MotionStep Output

The final output of the pipeline is a list of MotionStep objects, each containing:

- valid joint indices,
- target joint angles,
- a defined execution duration.

This confirms that the high-level planning and kinematics pipeline is complete and ready for execution in simulation.

Limitations

While the planning and kinematics pipeline is fully functional, the execution of the motions in MuJoCo depends on the simulation interface.
At this stage, the pipeline outputs correct motion commands, but full physics-based execution requires further controller tuning and simulator integration.

Challenge 5 - Plan Execution in Simulation

Forward and Inverse Kinematics

In order to execute the motion, we need forward and inverse kinematics to calculate the required joint angles. Both functions have been tested successfully, and the results are satisfying. When the holds are within reachable distance, the inverse kinematics successfully computes the required joint angles with a position tolerance of less than 1 mm.
However, there is still room for improvement in the inverse kinematics to make it more universal and robust. In our current solution, we avoid singular configurations by simply choosing a different initial position for the calculation. An algorithm that detects and intelligently avoids singular positions would be a more elegant and robust solution. For our use case, however, this simple approach works sufficiently well.
We also take another shortcut to solve the inverse kinematics for the legs. Since the leg joints are redundant, this easily leads to a loss of rank in the Jacobian, making it often impossible to solve the inverse kinematics problem. To avoid this issue, we lock the hip_yaw joint at −90 degrees, effectively removing it from the calculation. When the solution is close to the target position, we unlock the hip joint to improve accuracy. The hip_yaw motion is still limited to a range between −80 and −100 degrees. This does not correspond to a real human hip, but it prevents the leg from colliding with the wall.
This approach works well enough for simple climbing routes, as there is no need to rotate the hip back to 0 degrees while climbing. To achieve a more elegant solution that fully exploits the complete range of motion of all joints, the mathematical approach would need to be changed to properly handle rank deficiencies in the Jacobian.

Initial Integration and Execution Attempts

Following the configuration and interface design, core components of the control loop were integrated into the simulation environment. This allowed actuator commands to be applied and the robot state to be continuously observed.

Preliminary tests confirmed correct integration of the simulation setup, stable simulation behavior, and reliable extraction of the predefined outputs. These results demonstrate that the simulation infrastructure and execution interface are functional.

Testing and Debugging of Motion Execution

Multiple attempts were made to test and debug the execution of planned motions within MuJoCo. During these tests, several challenges were identified, including controller tuning, joint indexing consistency, and synchronization between planned motions and simulation timing.

Although a complete execution of the climbing motion could not be achieved within the project timeframe, these tests provided valuable insights into the complexities of humanoid motion execution in physics-based simulators.

Limitations and Time Constraints

Due to time constraints, the full execution loop and extensive simulation experiments could not be completed. Priority was placed on establishing a correct and robust simulation setup, preparing all required outputs, and implementing the execution interface.

Despite these limitations, the current state of the project provides a solid foundation for future work. The simulation environment is stable, the execution interface is functional, and all necessary outputs are prepared for full task execution and further experimentation.

Challenge 6 - Plan Execution in Real World (Bonus)

The primary objective of this challenge was to bridge the "Sim-to-Real" gap by translating the kinematic trajectories calculated in our software into physical actuation on the constructed robot. While the theoretical model assumes ideal conditions, the physical world introduces friction, gravity, and hardware constraints.

For the real-world execution component of this project, our primary focus was the physical realization of the robotic agent to ensure it was mechanically capable of supporting the planned trajectories. Transitioning from a digital model to the actual hardware presented significant assembly challenges, particularly regarding the structural integration of the HS-311 servos with the aluminum brackets. Securing the servos was physically difficult; many of the mounting points were located in tight spaces between the U-brackets, making it incredibly hard to reach the screws with standard tools and requiring patience to ensure every joint was rigid. We also had to be precise with the initial alignment of the servo horns to match our theoretical "zero" pose, as any mechanical offset here would ruin future control attempts. Furthermore, cable management became a critical physical planning step to prevent the wires from tangling around the 20cm arm links. This phase successfully delivered a fully assembled, mechanically stable robot ready for the software control implementation.

IV. SHORT SUMMARY

This assignment presents the design and implementation of a humanoid rock-climbing robot in simulation. The project covers the complete pipeline from robot design and MuJoCo simulation setup to climbing hold detection, path planning, kinematic computation, and motion execution preparation.

We did not manage to fully finish the simulation or transfer our preparation to a real robot. The task turned out to be too complex for the given amount of time. All group members were new to robotics, and our knowledge was mainly based on the material covered in class. Additionally, none of us had worked with MuJoCo before, which required a significant learning effort. Kevin and Philip received the task in October, and it quickly became clear that the scope of the assignment was too large to be solved by only two students.

When Aziz and Ala joined the group at the beginning of December, the remaining time was already very limited, making it difficult to achieve a complete and polished result. Working in an international group with a large time difference was an additional challenge, but also a valuable and enjoyable experience that required good communication and coordination.

Nevertheless, we successfully designed a humanoid robot model, implemented a computer vision algorithm for hold detection, and prepared all essential components required for executing the climbing plan in simulation. We also took first steps towards transferring our work to a real-world setup. Given the circumstances, we are satisfied with what we achieved and with the amount of knowledge and practical experience we gained throughout the project.

V. DISTRIBUTION OF WORK

Work/Task	Author
Designing the Robot	Philip Liebscher
Setting up the Simulation	Philip Liebscher
Collision Modelling	Philip Liebscher
Forward and Inverse Kinematics	Philip Liebscher
Hold parsing from XML files	Ala Rejeb
Climbing path planning algorithm	Ala Rejeb
Inverse kinematics integration	Ala Rejeb
Motion planning pipeline (planner → IK → motion)	Ala Rejeb
Generation of MotionStep instructions	Ala Rejeb, Mohamed Aziz Laarbi
Initial MuJoCo setup and robot model loading	Mohamed Aziz Laarbi
Preparation of MuJoCo execution interface	Mohamed Aziz Laarbi
Testing and debugging of motion execution in MuJoCo (unable to complete it)	Mohamed Aziz Laarbi
Preparing Presentation	Mohamed Aziz Laarbi
Computer Vision	Kevin
Real Life implementation	Kevin

VI. SOURCES

Figure 2:

https://www.ehrenkind.de/cdn/shop/files/Kletterwand-Kinderzimmer.jpg?v=1713467253&width=1200

https://www.bergfreunde.de/blog/wp-content/uploads/2020/06/rock-climbing-wall-gbfaabc8a6_1920.jpg