Quadruped Robot with Path Planning in Simscape

Andrés Eduardo Renjifo Restrepo
Sigi Deliallisi
Leonard Renner
Isaac Charbel Tejeda de la Garza
Roman Podieiablonskii
Chien Yu Hua

Acknowledgements:
We sincerely appreciate Andreas Apostolatos, Jan Janse van Rensburg and Steve Miller for their invaluable guidance, insightful feedback, and unwavering support throughout this project. Their expertise and encouragement have been instrumental in its success.

1. Introduction

Throughout modern history, humanity has dreamed of creating machines that serve and help us with everyday tasks that are physically dangerous or strenuous. In the last couple of decades, the field of robotics has produced increasingly complex, capable, and reliable mechanical assistants for many different fields, like manufacturing in factories, space exploration, household vacuum cleaners or surgical devices; as engineers, we believe that our ethical duty is to apply our skills to develop tools and solve problems for the benefit of society. A particular situation where replacing human beings with robots is of value is search and rescue in hazardous environments, like collapsed buildings after an earthquake, floods, gas leaks or mining and radioactive accidents.

The goal of this work is to develop a digital twin of a walking quadruped robot that would assist in this kind of search and rescue missions. To do so, the robot should be able to scan the surroundings, act as a communication tool in real-time, lift debris to help free people, and transport small amounts of life sustaining goods. Before achieving this complicated goal, the robot first needs to move and avoid obstacles before it can offer help. The tools that were used were Simulink and Simscape Multibody from MathWorks. Simscape Multibody helped us build the 3D models of different environments our robot might have to navigate and the robot itself which can walk, trot, turn to the side, and walk to the side, just like a real-life dog can. Simulink, on the other hand, helped us control the interaction between the environment and the robot by "virtually scanning" the area and calculating the ideal set of actions that need to be performed by our digital twin for a successful navigation.

For this purpose, we divided our team into three sub teams: The *Environment*, the *Gait*, and the *Control* teams. The "Environment" team was responsible for the staging of different setups which would mimic real-world accidents. These setups were then converted into binary matrices that were the starting point for the "Control" team. The latter team was able to calculate the ideal path needed to navigate through these matrices and convert this math into a set of numerical commands that are understandable to the robot such as 0 (stand still), 1 (walk straight), 2 (trot), 6 (turn left), or 7 (turn right). These commands were then delivered to the "Gait" team who was able to control the local movements of every part of the robot's body to perform the numerical commands.

2. Multibody problems in general

In general, multibody problems describe systems where different elements interact with each other, which could be rigid, elastic or even plastic. They could be connected by joints of several types, where some degrees of freedoms (DOFs) are allowed, and some are blocked. Often, the goal is to investigate movements, deformation, or resultants of the bodies, while considering dynamic effects, contact and more nonlinearities.

A suitable environment where these kinds of simulations can be performed is the Simscape Multibody toolbox in Simulink [1], [2], [3], where it is possible to create such multibody systems comprised of the following blocks:

- Solid Blocks: This block describes the geometry and the appearance of an element.
- Rigid Transform Block: This block enables to rotate or translate the active coordinate system which makes it possible to move from one position to another at the same element.
- Revolut Joint Block: Joint blocks set a certain amount of DOFs free (here: one rotational DOF), while the remaining ones stay locked. The two frames are connected to two elements, whereas the exact position at the single element is described by a Rigid Transform Block.
- Contact Block: This Block takes care of contact between two elements. The geometry of a Solid Block is therefore taken to compute for example resulting forces and penetrations of the elements.

As a start of the project, we were provided with basic tutorials of simple multibody systems to get to know the workflow in Simulink and Simscape, which then acted as a starting point for the robot model. These systems included for example a model of a Bouncing Ball, where basic concepts of contact could be explored. Following that, the handling of joints and the interaction of two elements were examined by creating a double pendulum with the help of another tutorial and furthermore the implementation of mathematic functions, that determine the movement of a joint, in combination with contact were shown in the tutorial of creating a hopping leg.

Altogether these tutorials provided a good base for the next tasks to build a model of a quadruped robot with Simulink.

3. Implementation of simple walk gaits

Context:

For our next step, we use the previously described basic models and introduce two additional Simscape blocks [2], [3]:

- Rectangular Joint: A joint that restricts the movement in z direction and the rotation around the x and y axes.
- 6-DOF Joint: A joint that does not restrict any Degree of Freedom.

The key features of each base model are combined so that the four legs interact in a forward walking movement, representing a first version of the walking robot.

Implementation:

At first, the 'hopping leg' model is used to recreate the walking movement of one leg, while the hip joint is fixed in the air. Our first approach was to directly determine the angles of the hip $\theta_{hip}(t)$ and the knee $\theta_{knee}(t)$ in one cycle by the following trigonometric functions [4]:

$$0 \leq t \leq \tfrac{3}{8}T: \quad \theta_{hip}(t) = A\sin\left(\tfrac{2}{3}(2\pi ft - 1.5\pi)\right)$$

$$\tfrac{3}{8} \leq t \leq \tfrac{5}{8}T: \quad \theta_{hip}(t) = A\sin\big(2(2\pi ft - 1.0\pi)\big)$$

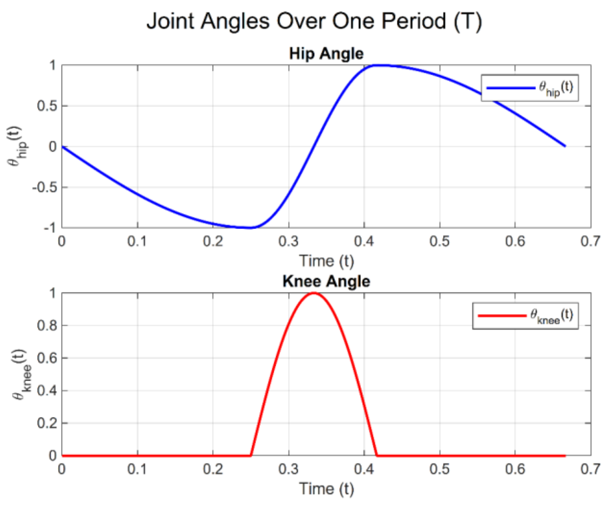
$$\tfrac{5}{8} \leq t \leq T: \quad \theta_{hip}(t) = A\cos\left(\tfrac{2}{3}(2\pi ft - 1.25\pi)\right)$$

$$0 \leq t \leq \tfrac{3}{8}T: \quad \theta_{knee}(t) = 0$$

$$\tfrac{3}{8} \leq t \leq \tfrac{5}{8}T: \quad \theta_{knee}(t) = A\sin\big(2(2\pi ft - 0.75\pi)\big)$$

$$\frac{5}{8}T \leq t \leq T: \quad \theta_{knee}(t) = 0$$

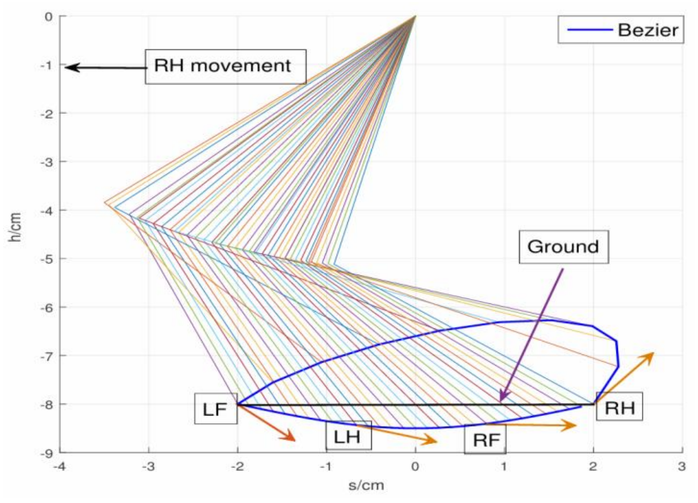
The two values θ_{hip} and θ_{knee} are added to the initial angle of the hip and the knee, that also define the position at rest for each leg. The value of the Amplitude A then depends on the selected minimum and maximum angle of each joint.



[Figure 01: Plot of the functions that determine the angles of the hip and knee joint [rad] over time.]

With this approach, it is possible to model the walk-gait of a leg in a simple way. But to handle changes of the ground, like steps or inclined terrain, the movement would need to be adjusted additionally.

To handle that in a more trivial way, we decided to implement a different approach, where the path of a foot is determined, and the angles are then calculated according to the position. This path is described by Bezier curves, where we can control the specific position of the Bezier points and accordingly are able to react on variations of the ground as shown in Figure 02.



[Figure 02: Path of one foot determined by mathematical functions. The swing phase is described by a Bezier curve while the support phase could be described linearly or by a sine function. [4]]

The angles of the hip and the knee joint are then calculated out of this path, which is called the inverse-kinematic approach [4], [5].

$$\theta_{knee} = \arccos\left(\frac{\left(x^2+y^2-(L_{upper}^2+L_{lower}^2)\right)}{2\cdot L_{upper}\cdot L_{lower}}\right)$$

$$\beta \;=\; \arcsin\left(\frac{(L_{lower}\cdot\sin(\theta_{knee}))}{(x^2+y^2)^{\frac{1}{2}}}\right)$$

$$\theta_{hip} \;=\; \arcsin\left(\frac{x}{(x^2+y^2)^{\frac{1}{2}}}\right)+\beta$$

Because of the non-differentiability of the arcus-functions at some points, we needed to implement borders for the values of the x and y coordinates of each foot. These correspond to the relative coordinate of each foot to the hip of the same leg. Hence, both values need to stay within the radius of $L_{upper} + L_{lower}$ around the origin, where L_{upper} corresponds to the length of the upper limb and L_{lower} to the length of the lower limb.

Four legs with these gaits implemented are attached to a torso and then the first version of a walking robot is created by shifting the cycles of each leg in time. In this state, the torso is attached to a rectangular joint. That makes it possible to investigate the path of the Bezier curve as well as the initial position of each foot to make the walking movement look natural. Afterwards, we attached the torso to a 6-DOF joint and adjusted the previously described points again until the robot was able to move stable without falling over.

Results:

In this state, the Robot is simply able to walk on flat terrain. But in case of a change of environment or inclined terrain, the robot would still perform the exact same gait and not be able to stay in balance. Already small obstacles like a thin plate would interfere with the strictly defined movement and could cause the robot to fall over or to be stuck in one position. Additionally, this state only allows walking forward in a straight line. The implementation of more gaits, for example for walking backwards, turning, and walking sideways, but also the transition phase from one gait to another will be focused on the following chapter.

4. Handling of changes in movement and gait

Context:

At first, each of the new gaits are explained in general terms and their implementation in Simulink is described, and then the progress of connecting the gaits with each other to ensure a smooth movement is detailed. To achieve this, we additionally needed the following Simulink functionalities [1]]

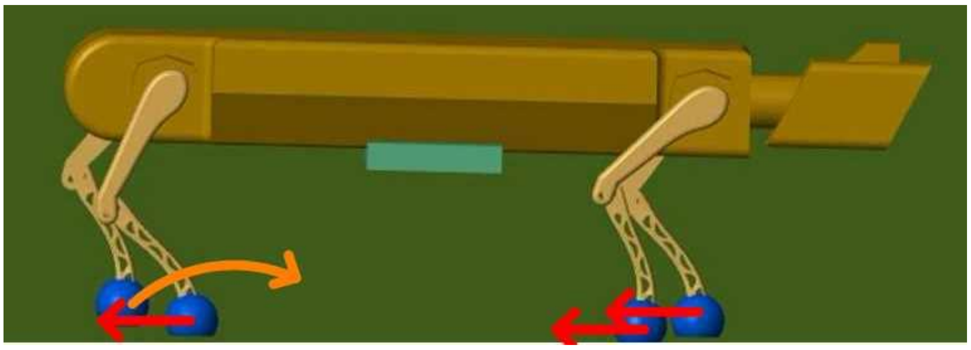
- Bus Selector / Bus Creator: Used to combine several signals in one.
- Switch Case / Action Subs. / Triggered Subs.: Depending on the state of a signal, the Switch Case block selects the corresponding subsystem. If this subsystem is declared as an Action subsystem, it is run as long as this statement is fulfilled, whereas if a subsystem is only triggered once, if it is declared as a Triggered subsystem.
- Memory: The memory block remembers the value of a signal for one simulation step.

Implementation:

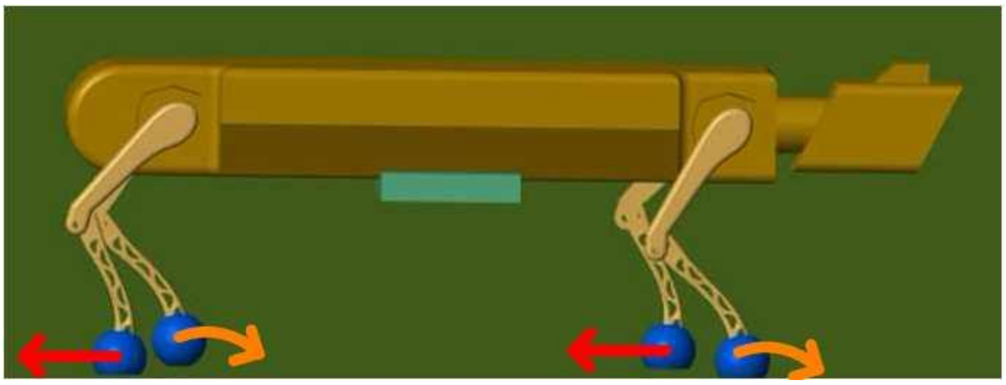
For the two distinct kinds of forward movement, the legs perform in the following order:

Walking: Front Left -> Back Right -> Front Right -> Back Left

Trotting: Front Left + Back Right -> Front Right + Back Left.



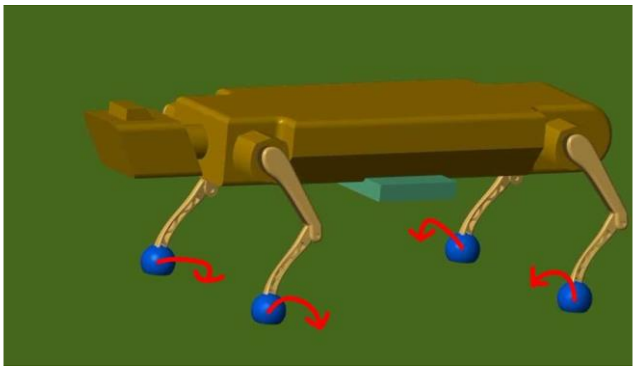
[Figure 03: Visualization of the walk gait. One leg performs a step while the other legs push the robot forward on the ground.]


[Figure 04: Visualization of the trot gait. Two legs that are diagonal to each other perform the step at the same time while the other two push the robot forward.]

In the walking gait, only one foot at a time is moving forward and the other three feet are slowly moving to the back to push the torso forward, while in the trotting gait the two diagonal pairs of legs move at the same time. This is implemented by using a local time for each leg, which only runs from zero to the full cycle time and is then reset to zero again. In this cycle every leg performs one step forward and by adjusting this value of the cycle time, the speed of the movement can be changed.

For the remaining gaits, it is not sufficient anymore to only shift the movement of each leg in time. Accordingly, the backwards gait already needs some more adjustments compared to the first two gaits, as it follows the same Bezier points, but in a switched order. The implementation of sideways movements, as expected during turning or walking sideways, requires even more modification with the necessity of a third joint for each leg, that needs to be integrated at the hip. This allows lateral movement of the feet, depending on the value of the angle, which is calculated in the same way as the other two joints. First a Bezier curve (2D) in lateral direction of the robot is created that again describes the path of a foot. Then the corresponding angles of these three joints are calculated. Here, the lateral joint and the knee joint are handled first, and these results are then used for the calculation of the hip joint, so that the foot only moves sideways and not in longitudinal direction.

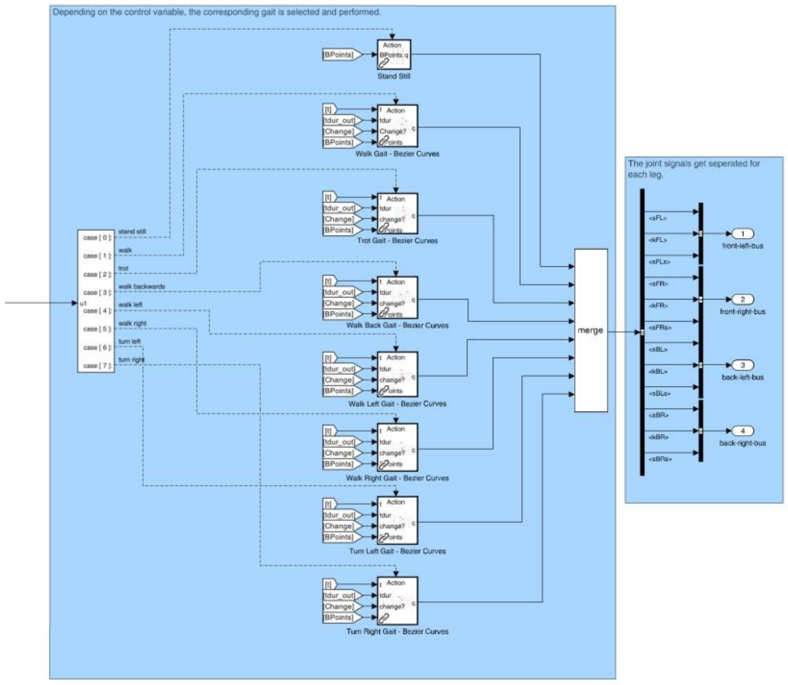
In the Walk Sideways Gait, all four legs perform their movement in the same sequence as in the Walk Forward Gait but based on the above-described calculation of the angles that enables sideways movement. The only difference of the Turn Sideways Gait to this implementation is that the rear legs perform their sideways movement in the opposite direction than the front legs.



[Figure 05: Visualization of the turn left gait. The two front legs perform a step to the left while the two rear legs perform a step to the right.]

All these gaits could in this state be run separately by selecting the corresponding control value:

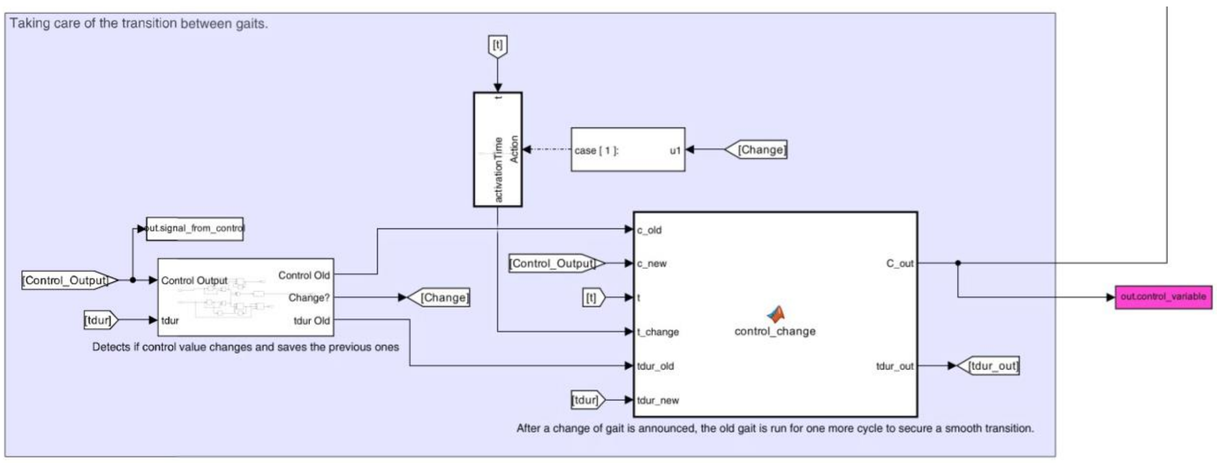
- Control Value = 0: Stand Still
- Control Value = 1: Walk Forward
- Control Value = 2: Trott Forward
- Control Value = 3: Walk Backwards
- Control Value = 4: Walk Left
- Control Value = 5: Walk Right
- Control Value = 6: Turn Left
- Control Value = 7: Turn Right



[Figure 06: Implementation of the gait selector. Depending on the control variable, the corresponding gait is performed and the angles for the joints of each leg are given as an output.]

Since the same functions were used in several different gaits, the use of Simulink libraries improved the workflow significantly. By that, changes of subsystems only needed to be implemented in one library, while all the different gaits then just refer to this specific library.

At this state, all the different gaits work independently, but it is not possible to change from one gait to another smoothly. Therefore, we decided that every gait starts their movement at equal initial positions and after the control value is changed, every foot moves back to the same position again to start the next movement. This is implemented in a way that after a change of gait is announced, the movement of the gait goes on for one more cycle time but with the modification that a foot stops when it reaches the initial position. These cycles are implemented in a way that at the initial position, the local time of a foot is equal to zero. In the process of keeping this local time at zero, the values of the previous timestep are checked with the help of the memory block.



[Figure 07: Simulink structure of the functions that handle the transition of one gait to another.]

With these modifications, the movements of each gait could be changed smoothly without the risk of losing balance and without discontinuities of the joint angles.

Results:

After finishing this chapter, the robot could walk in different directions and is able to follow a path, depending on the sequence of control values that are given as an input. Additionally, it is possible to regulate the movement's speed by adjusting the cycle time, that the robot needs to perform one step with each foot. Regarding the environment, this state only consists of flat terrain without any changes of height that could interfere with the movement of the robot. Because of that, the following chapter concentrates on the development of the environment.

5. Implement an environment and create occupancy grid

Context:

To increase the challenge difficulties and simulate real-world environments more accurately, we designed an inclined terrain and incorporated an S shaped channel as an obstacle.

We first set up an environment library block to organize all our obstacle components inside. Using Simscape brick solid, we constructed the obstacles by assembling several flat walls flawless together into a cohesive structure. To achieve a more adjustable surface, we replaced the infinite plane with a grid surface that we can easily modify the height of specific sections of the terrain [6].

Furthermore, we defined an occupancy grid to assign the exact position for the obstacles so that the robot can know which areas are occupied or free while navigating through the channel.

Implementation:

To ensure realistic interactions between the robot and the environment, we first assigned each leg and body to designated labels due to clarity. Afterwards, we connected contact force blocks to independent labels. These were then integrated into a contact force subsystem. Additionally, every obstacle component within the environment library was connected to a contact force subsystem to form a collision between the obstacles and robot itself instead of passing through the obstacles directly.

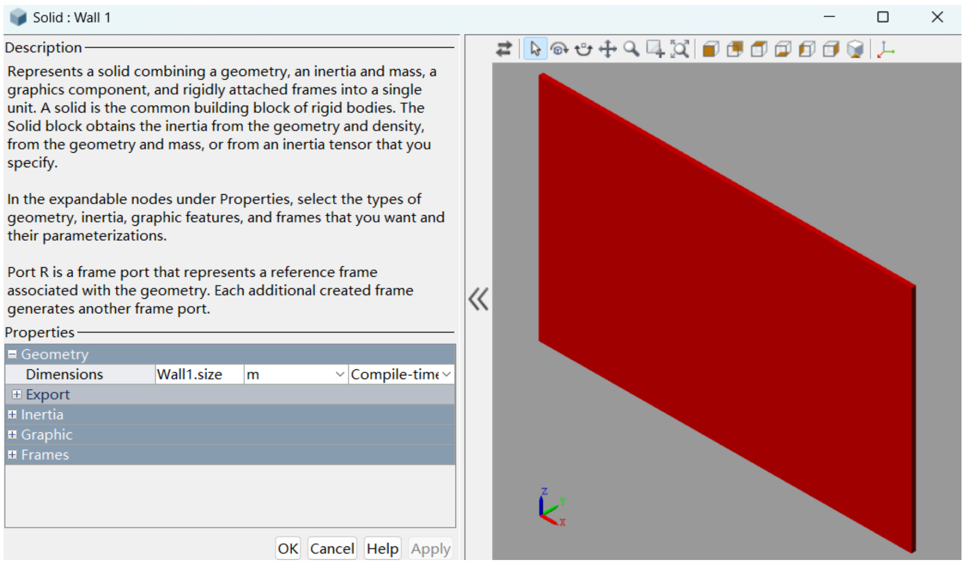
To modify the parameters easily, we set up a MATLAB script and defined all necessary data inside. We set the grid step to 0.2 and defined each wall data such as center, position and size. Wall 1 for example as the code below.

```
grid_step = 0.2;
x_grid_vector = 0:grid_step:10;
y_grid_vector = 0:grid_step:5;
z_heights = zeros(length(x_grid_vector), length(y_grid_vector));

% Wall Data
Wall_Height = 0.5;
Wall_Length = 0.01;
Wall_Width = 0.8;

% Wall #1
Wall1.index = [14,12];
```

```
Wall1.position = [x_grid_vector(Wall1.index(1)), y_grid_vector(Wall1.index(2)),
z_heights(Wall1.index(1), Wall1.index(2))];
Wall1.size = [Wall_Width, Wall_Length, Wall_Height];
```


Solid : Wall 1

Description

Represents a solid combining a geometry, an inertia and mass, a graphics component, and rigidly attached frames into a single unit. A solid is the common building block of rigid bodies. The Solid block obtains the inertia from the geometry and density, from the geometry and mass, or from an inertia tensor that you specify.

In the expandable nodes under Properties, select the types of geometry, inertia, graphic features, and frames that you want and their parameterizations.

Port R is a frame port that represents a reference frame associated with the geometry. Each additional created frame generates another frame port.

Properties

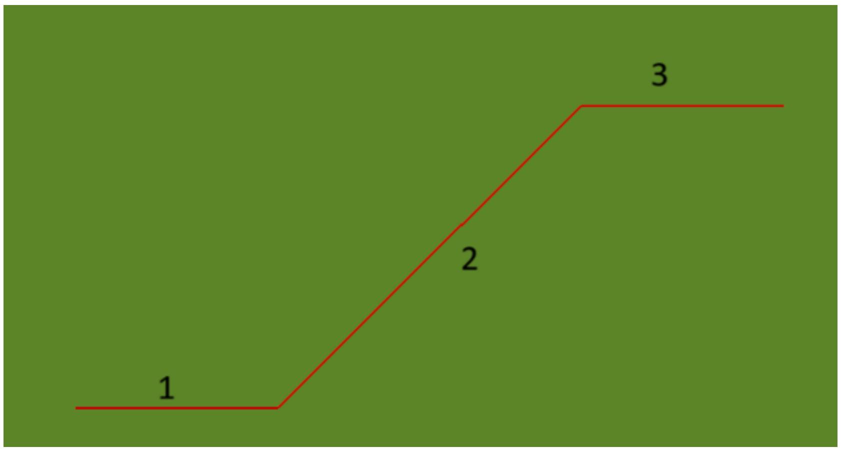
Geometry			
Dimensions	Wall1.size	m	Compile-time
Export			
Inertia			
Graphic			
Frames			

[Figure 08: Simulink “Solid: Wall” block.]

According to this approach, we just need to input the size we have defined into every brick solid then the obstacles will accurately appear in the environment.

It’s simple to place the position of the walls that align to the grid; however, we still have some walls that span across the grid. For those which don’t align to the grid, we need to calculate the differences in x and y direction between 2 parallel walls. After having these 2 data, we can thereby know how much angle the middle wall should turn with

$\theta = a\tan 2\,(\Delta y, \Delta x)$ by connecting the rigid transform block with the wall in the environment library block that needs to be turned. We take wall 1, 2, 3 for example as the code below.



[Figure 09: Visualization of the following implementation of the environment.]

```
% Wall #1
Wall1.index = [14,12];
Wall1.position = [x_grid_vector(Wall1.index(1)), y_grid_vector(Wall1.index(2)),
z_heights(Wall1.index(1), Wall1.index(2))];
Wall1.size = [Wall_Width, Wall_Length, Wall_Height];
% Wall #3
Wall3.index = [24, 18];
Wall3.position = [x_grid_vector(Wall3.index(1)), y_grid_vector(Wall3.index(2)),
z_heights(Wall3.index(1), Wall3.index(2))];
Wall3.size = [Wall_Width, Wall_Length, Wall_Height];
% Compute the angle between Wall 1 & 3
delta_x = (Wall3.position(1)-Wall3.size(1)/2) - (Wall1.position(1) +
Wall1.size(1)/2) ;
delta_y = y_grid_vector(Wall3.index(2)) - y_grid_vector(Wall1.index(2));
theta = atan2(delta_y, delta_x);
theta_val1 = theta;
% Wall #2
Wall2.index = [(Wall1.index(1)+Wall3.index(1))/2,
(Wall1.index(2)+Wall3.index(2))/2];
Wall2.position = [x_grid_vector(Wall2.index(1)), y_grid_vector(Wall2.index(2)),
z_heights(Wall2.index(1), Wall2.index(2))];
Wall2.size = [sqrt((delta_x)^2 + (delta_y)^2), Wall_Length, Wall_Height];
```

After settling all walls to their corresponding position, we need to assign their exact position to map the occupancy grid [7]. To do this we first make an occupancy grid to be a x and y dimension matrix. For the wall that aligns to the grid, we simply define how many grids are incorporated in this wall and mark the corresponding grids to be 1 as occupied.

```
occupancy = zeros(length(x_grid_vector), length(y_grid_vector));
occupancy(((Wall1.index(1) - ((Wall_Width / 2) / grid_step)):(Wall1.index(1) +
(Wall_Width / 2) / grid_step)), Wall1.index(2)) = 1;
```

Nevertheless, with the wall that does not align to the grid, we use the slope to define which grid nodes are occupied by this wall. For each grid that is passed by the wall then 4 nodes will be marked as occupied. We use this method by iterating through the y-axis grid, the code maps the wall to the occupancy grid, filling it layer by layer based on the wall's width and grid step size.

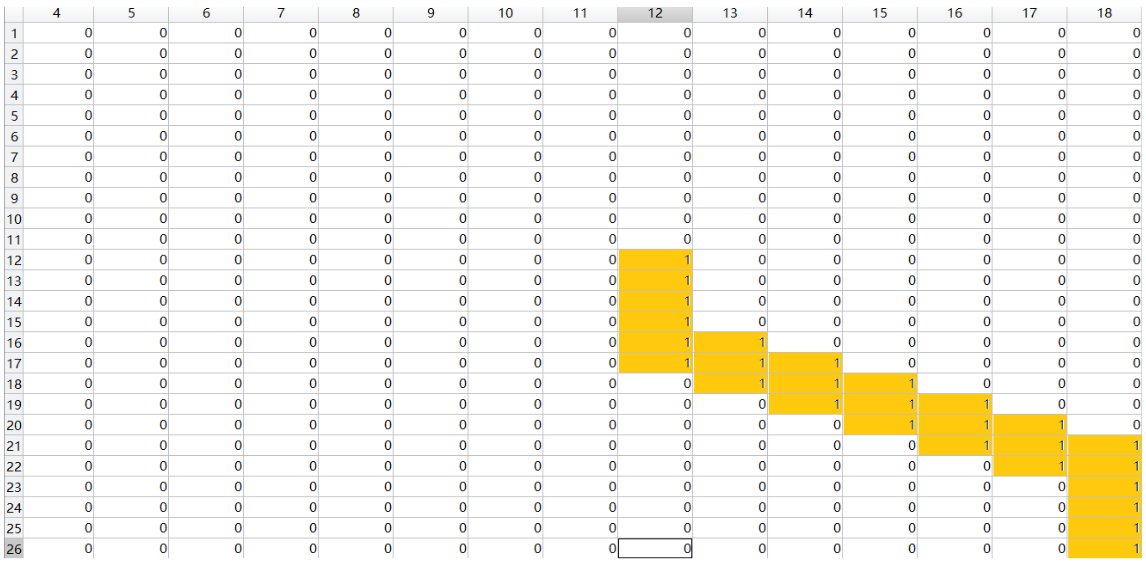
```
n_x = delta_x / grid_step;
n_y = delta_y / grid_step;
v = n_y / n_x;

% wall 2 transform to occupancy grid
for i = 0:n_y
```

```
occupancy(((Wall1.index(1) + ((Wall_Width / 2) / grid_step) + round((1/v) *
i)):(Wall1.index(1) + (Wall_Width / 2) / grid_step) + round((1/v) * (i + 1))),
Wall1.index(2) + i) = 1;
occupancy(((Wall1.index(1) + ((Wall_Width / 2) / grid_step) + round((1/v) *
i)):(Wall1.index(1) + (Wall_Width / 2) / grid_step) + round((1/v) * (i + 1))),
Wall1.index(2) + (i + 1)) = 1;
end
```

Results:

Based on the occupancy grid, we can determine the actual positions of obstacles as yellow part in the figure below, where 1 indicates occupied and 0 indicates unoccupied. Control team can thereby use this occupancy grid to design the path for the robot, enabling it to navigate and avoid obstacles effectively.



	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
12	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
13	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
14	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
15	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
16	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
17	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
18	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0
19	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0
20	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0
21	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1
22	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
26	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1

[Figure 10: Example of the occupancy grid matrix.]

6. Use occupancy grid to find path / control

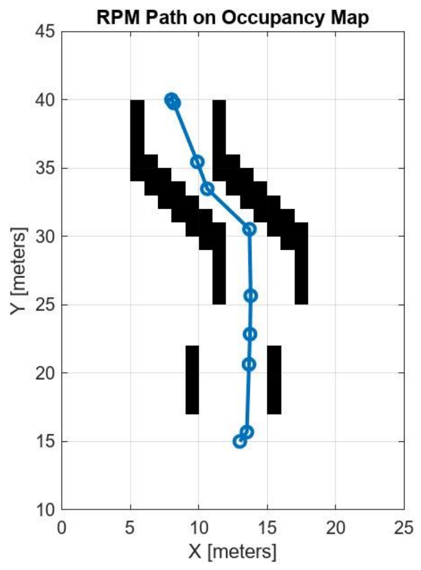
Context:

The next step to building the digital twin of the robot was to provide it with commands that helped it navigate any environment and obstacles. The robot had to obey any given command that was decided through a set of Simulink blocks containing user defined functions.

Implementation:

The first of these functions uses the occupancy map to calculate a two-dimensional array that represents the x and y coordinates of the path that the robot needs to take in the given environment. This array is obtained by using MATLAB's Navigation Toolbox [8]. A few functions from this toolbox find the most optimal path, which in our case is the shortest, obstacle-free one. Every point of this path represents the position of the robot, as can be seen in *Figure x* below.

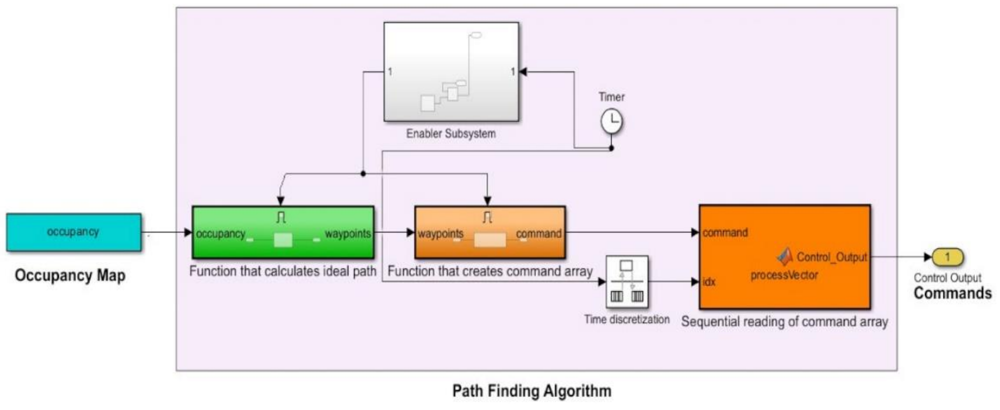
The second function uses the array obtained introduced in the previous paragraph. For simplicity, we called it *Waypoints*. Waypoints are input into a for-loop which determines the relative difference between the current and the previous position of the robot, and decides the next appropriate action, which corresponds to a numerical command that the gait team needs. For instance, if the difference in angle between two points is larger than a given tolerance, say 10°, and the difference in x values in positive, then the command is 7*,* which is equivalent to turning right. The result is yet another array, this time one-dimensional, the elements of which correspond to the sequential actions that the robot takes as it goes through the calculated ideal path.



[Figure 11: Visualization of the ideal path of the robot.]

Finally, the last function is one that uses time as an input, as well as the one-dimensional array obtained previously to use each of its elements as a separate input for the rest of the simulation.

The block diagram that connects the aforementioned functions is shown in the figure below.



[Figure 12: Simulink implementation of the Path Finding Algorithm.]

Results:

The robot receives the correct command given by the binary occupancy grid. The shortest path through the environment is found and it is successfully converted into numerical commands. If, for instance, in the path, there is a right turn, the robot receives a 6; if there is no turn to be made, the robot receives a 2 and keeps trotting.

The current issue that the “Control” team is facing is to make the robot turn exactly by as many degrees as specified in the ideal path. For now, the robot can only turn approximately 10 degrees, no matter the context. A possible solution to this is to manipulate the amount of time that a command is executed in the Control block diagram. Another solution is to create a feedback loop which will take the angle by which the robot has turned in small increments of time and only execute the turn until the predicted and actual angles have matched.

7. Robot reacts on changes in the environment / balancing

Context

The robot was able to walk correctly on flat terrain but on inclined terrain it was clear than the center of gravity was not aligned with the centroid of the feet. This was a problem because:

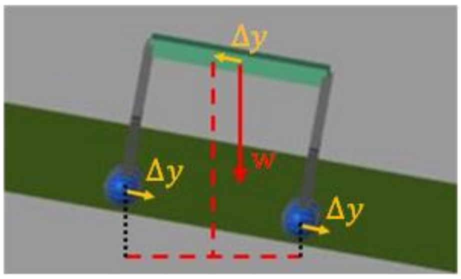
- Loss of balance: If the forces acting on the robot are not balanced, it may tip over or fall.
- Reduced stability: Reduce its stability and make it more difficult to move smoothly and efficiently.
- Increased wear and tear: Additional stress on the robot's joints and motors, leading to increased wear and tear and a shorter lifespan.
- Reduced performance: In terms of speed, agility, and payload capacity.

To control this imbalance, it was necessary to know the exact contact force between every foot and the floor surface, this was possible enabling the “Normal Force Magnitude” sensor of the “Spatial Contact Force” blocks.

Implementation

Having the forces for every foot, the idea is to make all of them similar, effectively making the center of gravity of the robot coincide with the centroid of the feet position. The forces had to be summed in groups for different purposes, the total sum is important to know precisely the total weight of the robot, the sum of the forces of the two feet on the left side is important for the lateral balancing, and the sum of forces on the front feet is important for the longitudinal balancing. It is also important to mention that, even though what we would like to move is the torso of the robot that carries most of the weight, we cannot move directly the torso of the robot, but the position of the control points that define the movement of the feet, we modify all the points by the same value.

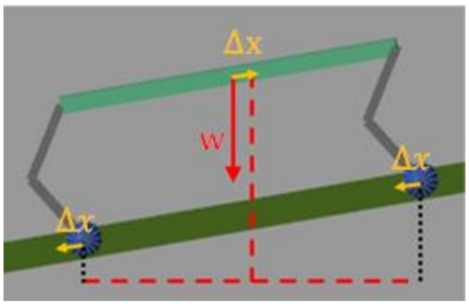
In the following figure you can see the scheme of the lateral balancing and the equation of the required displacement, there is an extra factor of 0.2 that was handpicked to slowdown the correction to prevent instabilities. The center of gravity of the robot is not aligned with the geometric centroid of the feet positions creating the imbalance. To correct this imbalance the torso of the robot (that concentrates most of the should be displaced, as the torso itself is not directly controlled, it is necessary to move the support points of the feet by means of changing the transversal coordinates of the Bezier curves control points.



[Figure 13: Robot standing on inclined terrain in lateral direction. The center of gravity and the accordingly needed displacements to balance out are highlighted.]

$$\Delta y = (widthTorso) \cdot \frac{totalForce/2\ -\ forceLeftFeet}{totalForce} \cdot 0.2$$

A similar scheme and equation are showed for the longitudinal balancing, the longitudinal coordinates of the control points for the Bezier curves that describe the feet movement is modified to make the center of gravity coincide with the centroid of the feet



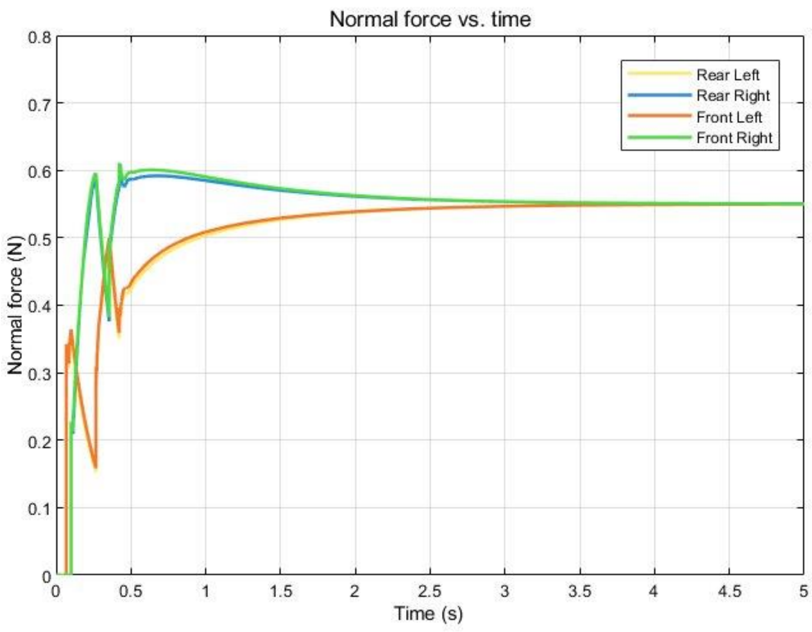
[Figure 14: Robot standing on inclined terrain in longitudinal direction. The center of gravity and the accordingly needed displacements to balance out are highlighted.]

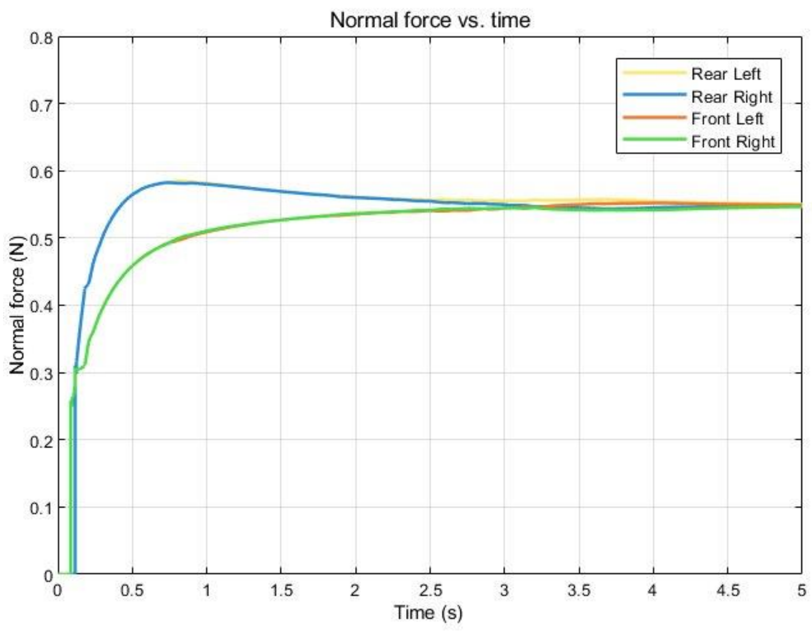
$$\Delta x = (widthTorso) \cdot \frac{totalForce/2\ -\ forceFrontFeet}{totalForce} \cdot 0.2$$

The equations were implemented using the "MATLAB Function" from Simulink, and the position of the Bezier curve control points were both inputs and outputs of the function, "Transfer Function with Initial Outputs" where used to prevent the algebraic loops resulting of having the same variables as inputs and outputs.

Results

After the implementation of these balancing functions, the robot was able to adapt to inclined terrains. Figures of the forces on the four feet are presented to 2 scenarios i.e. Standing still on a slope in the lateral direction and on the longitudinal direction:





[Figure 15: The normal forces of each foot over time. The first plot shows the case of lateral balancing, the second one shows longitudinal balancing.]

It should be mentioned that the magnitude of the balancing displacement is limited by the size of the limbs, so not every point is possible to reach and thus after some amount of inclination the robot cannot stay balanced.

8. Interactive control with Xbox controller

Context:

Section 5 details the automatic path-finding functionality of the robot used when it must traverse from one point to another in the fastest, most efficient way possible. However, certain scenarios during its mission might demand taking a specific route. Thus, it was established that a manual, interactive control method should be implemented for the robot, such that the operator can directly guide it as the situation requires.

Implementation:

Different approaches were considered for taking manual input from the user and translating it into movement in the robot, including input from the keyboard of the computer running the model and then mapping certain keys to each gait, so that the robot executes the gait for as long as the corresponding key is pressed.

Writing a function from scratch that can read continuous keyboard input in MATLAB proved to be impractical, so more straightforward options were explored within the existing blocks in Simulink, including graphical interactive elements in the Dashboard library. At the end, the Joystick block from the 3D Animation Library [9] was selected, which proved to be ideal for reading external input from a controller. Using this block, a simple videogame console controller is connected into the computer and the actions from its buttons and sticks are read as input. An Xbox One controller was chosen to assure compatibility.

In the case of buttons, the output given by the Joystick block is a 1D array with as many elements as buttons in the controller and two values per element: 0 and 1, which correspond to the states pressed or not pressed. After this block, a simple MATLAB function maps each element of the vector to a constant from 0 to 7, which is then passed on to the gait selector. The table below shows this relationship. When no button is pressed, the robot stands still.

Button	Gait
-	0 – Still
Y	1 – Walk forwards
A	2 – Walk backwards
Start	3 – Trot
X	4 – Walk left
B	5 – Walk right
Left bumper	6 – Turn left
Right bumper	7 – Turn right

[Table 1: Mapping of input and output using buttons on the controller]

A second control mechanism was developed using the axes of one of the joysticks and the two triggers, which may be more intuitive to some users. In this regard, the circular zone corresponding to the movement of the joystick was divided into four 90° quadrants centered on the horizontal and vertical axes (in the shape of an X). Every point within each quadrant has the same magnitude, which means that no matter how far or close to the centre the joystick is pushed, the movement speed is the same. The following table shows this relationship.

	Axis	Gait
	-	0 – Still
Joystick	Front quadrant	1 – Walk forwards
	Back quadrant	2 – Walk backwards
	Left quadrant	6 – Turn left
	Right quadrant	7 – Turn right
Trigger	Left	4 – Walk left
	Right	5 – Walk right

[Table 2: Mapping of input and output using axes on the controller]

9. Conclusions and Future Work

In conclusion, Simulink and Simscape Multibody proved to be excellent tools for the development of the digital twin of this walking quadruped robot. This project successfully integrated physics-based modeling, control system design and virtual testing environments to replicate the behavior of the physical robot in a digital domain. This allows for quick modelling and testing of robotic prototypes during the design phase without having to physically build them, which is a costly and time-consuming process.

Regarding the control of the robot using the gaits, our current implementation only allows for one gait to be active at any given time. This is based on the definition of the Bezier curves, which were implemented in two dimensions. A further improvement that could lead to a smoother movement of the robot as well as more dexterity when navigating more complex terrain would be the ability to combine the current gaits, such that the robot could move forwards and turn to a side to follow a curved path, for example. This would involve modelling the curves for the trajectories of the joints in 3D.

Closely related to this is the path-following algorithm, which currently has a functioning but slightly clumsy sequence. Once a certain threshold in the difference in angle between two waypoints is reached, the robot will change to the turning gait for a set amount of time/angular displacement, for example. This means that any given trajectory will be constructed out of several straight lines. A much better behavior would involve following curved paths.

Of course, additional gaits would be necessary to traverse even more complex terrain. Currently, our model can only traverse straight or inclined, relatively flat terrain. For the complex mission proposed at the beginning, gaits to go up and down stairs, to crouch through particularly narrow sections, or even swim or jump over obstacles should be implemented.

Finally, a more advanced balancing system would be necessary to react to dynamic changes in the environment, which could include unstable, “broken” or granular terrain, such as debris from a collapsed building or gravel.

Bibliography

[1] MathWorks, "Simulink," *MathWorks*, Available: https://de.mathworks.com/help/simulink/index.html. [Accessed: Jan. 27,2025].

[2] MathWorks, "Simscape," *MathWorks*, Available: https://de.mathworks.com/help/simscape/index.html. [Accessed: Jan. 27,2025].

[3] MathWorks, "Simscape Multibody," *MathWorks*, Available: https://de.mathworks.com/help/sm/index.html. [Accessed: Jan. 27,2025].

[4] L. Zhou, H. Qian, Y. Xu, and W. Liu, "Gait design and comparison study of a quadruped robot," *2017 IEEE International Conference on Information and Automation, ICIA 2017*, 2017, pp. 80-85, doi: 9781538631546.

[5] C. Hahn, "Team Automatons builds a quadruped robot for the ABU Robocon 2019 task," *Student Lounge - MATLAB & Simulink*, Aug. 14, 2019. [Online]. Available: https://blogs.mathworks.com/student-lounge/2019/08/14/quadruped-robot-for-robocon-2019/. [Accessed: Jan. 27, 2025].

[6] MathWorks, "Grid surface," [Online]. Available: https://de.mathworks.com/help/sm/ref/gridsurface.html. [Accessed: Feb. 5, 2025].

[7] MathWorks, "Occupancy grids," [Online]. Available: https://de.mathworks.com/help/nav/ug/occupancy-grids.html. [Accessed: Feb. 5, 2025].

[8] MathWorks, "Navigation Toolbox," *MathWorks*, Available: https://de.mathworks.com/help/nav/index.html. [Accessed: Jan. 27,2025].

[9] MathWorks, "Simulink 3D Animation," *MathWorks*, Available: https://de.mathworks.com/help/sl3d/index.html. [Accessed: Jan. 27,2025].

[10] MathWorks, " Joystick Input", *MathWorks*, Available: https://www.mathworks.com/help/sl3d/joystickinput.html. [Accessed: Jan. 27,2025].