BioStrider: Design of a Six-Legged Autonomous Robot for Efficient Navigation in Sandy and Rocky Terrain

Jonathan Nez, Erica Thompson, Matthew Troyer

Pueblo Community College

Southwest Campus

Faculty advisor: Melissa Watters

April 14, 2025

Introduction

The BioStrider robot was developed to take part in the Colorado Space Grant Consortium Robotics Challenge. The challenge requires participants to design and build a robot analogous to a Mars rover type of robot. The BioStrider robot was designed to meet the constraints of that challenge, namely autonomous navigation in sandy and rocky terrain and limitations in weight (less than 4kg) and cost (less than $500). The project was also constrained by available time. The BioStrider team had 6 months to design, build, and test the rover.

The project goals unique to the BioStrider robot development team were to gain insight into insect biomechanics and locomotion and to use them to model mechanical, structural, and electronic systems using original and proven designs, including those from nature. The problems we intended to solve are navigation in varying environments to efficiently traverse complex environments and integrating those systems using sensors to support autonomous insect-like locomotion. The design could be scaled and utilized for various environments that humans and wheeled rovers may have difficulty crossing. In addition, the project design could be scaled for use in many other activities, such as disaster relief and rescue, military, surveillance, and wildlife management, and was inspired by some of the goals of the Elevate walking car concept under development by Hyundai Motors [1].

Throughout our research on insect locomotion, we observed how insects utilize their unique muscle physiology to walk and how we could use this same process of locomotion to aid us in our project. The standard explanation of insect walking is that they utilize a tripod gait, or three feet on the ground at any one time, to optimize stability [2], [3]. With the front and hind legs on one side of the body and the middle leg on the other side of the body on the ground, the other three are lifted, creating a triangular, or tripod, stance [4, Fig. 1]. The tripod is very stable.

It is hypothesized that the prevalence of the tripod gait in insect locomotion is related to its ability to traverse challenging terrain rapidly [4, p. 2]. In addition, this stability minimizes the central nervous system processing required concerning balance [2].

The prey chasing response to visual stimulus is examined in [5]. The mechanisms for chasing prey were considered similar to avoiding obstacles for the BioStrider robot. The "vision" for BioStrider was achieved through a time-of-flight sensor. The tiger beetle uses the measurement of the elevation angle of the prey to determine the distance, which can be determined by a compound eye [5]. This idea was used in the combination of a time-of-flight sensor and motion in both horizontal and vertical directions through two servos, as described in more detail in the following sections.

Methodology

Structural Design

The overall structure of the rover consists of a body section where the 6 legs are mounted and where all of the electronic components are housed. Leg segments consist of linkages to allow movement similar to how muscles would move an insect leg, and the rover could utilize various foot designs to address the varying terrain. An exploded diagram of the assembly of these components is shown in Fig. 1.

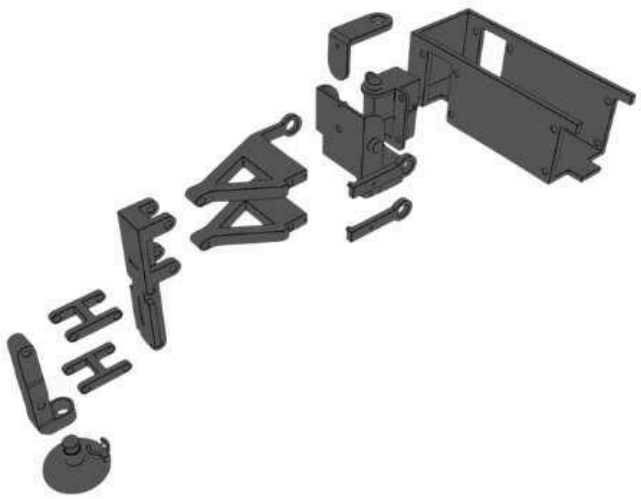

Fig. 1. Exploded view of leg assembly, servo housing, and body attachment

These components were modeled in Fusion 360 and then 3D printed using PLA (Polylactic Acid), PETG (Polyethylene Terephthalate Glycol), and TPU (Thermoplastic Polyurethane) filaments where appropriate.

A mixture of PETG and PLA, along with stainless steel hardware, was used for the rover's legs and body pieces, allowing for both structural stability and the ability to modularly replace pieces if necessary. The first iteration of the legs was unsuccessful as they were too large, and the motor connections were flimsy and quite weak, necessitating that they be connected in a Y shape instead of two separate pieces that didn't connect. Upon further examination, there were concerns about friction and precision with the 3D printed fasteners and how precise they had to be to work as intended. The second iteration of the legs was also unsuccessful as the legs were smaller, and the mechanical design for the lower half of the leg would not work the way it was intended due to excessive movement, as well as not being tall enough to allow for greater clearance from the ground. Issues pertaining to the motor housing, along with strength concerns with the connective braces, required that they be made thicker and printed with a stronger

material. The third iteration was successful after many components of the leg were modified to respond to what was learned from the previous iterations. From then on, we continued to make minor adjustments to this design as needed and added more metal fasteners at the joints to minimize friction and handle greater torque from overall movement.

Given the sand dunes' variations in temperature and terrain, PETG and TPU were used for the feet of the rover. PETG has a heat deflection temperature of 70℃ or 160℉, mimicking the Great Sand Dunes Tiger Beetle's ability to withstand the variations in temperature of its habitat [6]. TPU was used for the bottoms or "shoes" of several iterations of the feet, allowing for grip on harder, uneven surfaces like tires would for a wheeled vehicle.

The initial foot designs for the rover were inspired conceptually by science fiction media, particularly the AT-AT (All-Terrain Armored Transport) vehicles from the *Star Wars* franchise [7]. These fictional vehicles, although made real through miniature models used for the movies, exhibit wide, flattened, and articulated feet perfect for distributing weight across varied terrain. These design characteristics were implemented along with a textured bottom for grip on several iterations of the feet, allowing for better stability on the sand.

The use of TPU with 95A shore hardness, or the material's resistance to indentation, was a challenge in itself, proving to be difficult to print with and even harder to make flexible enough for a tire-like grip that was the desired result for the bottom of the feet.

Mechanical / Electrical Design

The rover has six legs with two degrees of freedom per leg. The legs are aligned in two linear rows, one on each side of the robot. While this design is simple, it presented several challenges for turning the rover and making room for each leg's horizontal movement. For any future designs, a hexagonal arrangement would be preferable.

The legs can be adjusted independently in the vertical and horizontal directions and independently of each other, utilizing two servos for each leg. The reason for using two degrees of freedom as opposed to three is the reduced number of servos. Three degrees of freedom would require eighteen servos, as opposed to the twelve required for two degrees of freedom. This reduces power consumption, weight, and complexity. Originally, the servo controlling the vertical movement of a leg was going to be placed in the knee to distribute the weight throughout the legs instead of condensing it all into the main body. This was problematic because it resulted in the knees being very wide, which severely limited the usable horizontal range of each leg. For this reason, the servos in the knees were moved to the hip, resulting in a more gimbal-like design.

Individually controllable legs are intended to give the robot tighter control and better footing in rough terrain, but this brings with it several issues. The legs need to be able to identify when they are grounded since they are meant to form to the ground. The robot also needs to be able to balance itself since the legs will often be at different elevations. The robot will also need to maintain a proper body elevation so that the legs are always able to reach the ground while also being high enough to step over objects.

To solve the issue of detecting the ground, each leg's lower section is divided into two parallel, vertical parts. They are attached by short linkages that allow a slight vertical variation between the segments while keeping them parallel. An IR sensor is then mounted to the inner segment, facing the outer segment. The foot of the leg is attached to the bottom of the outer segment. A strip of black electrical tape across from the IR sensor on the outer segment is positioned so that when the outer leg segment moves up or down, the sensor passes over the edge of the tape. Black tape was used since the IR sensor does not detect black but can detect the

material of the leg. This allows the sensor to detect the position of the lower segment by whether it senses the leg material or not. Tape was used to allow for fine-tuning the position that results in the leg being grounded. There is a rubber band attached in such a way that when the leg is not grounded, the lower segment of the leg will be pulled to its lowest position, resulting in the sensor being completely blocked by the tape and thus not sensing anything. When the leg is grounded, the weight of the robot overcomes the strength of the rubber band, and the lower segment of the leg moves to its uppermost position, resulting in the sensor detecting the material at the edge of the tape. This mechanism is shown in Fig. 2. This approach works regardless of the shape or size of the foot or what part of the foot is grounded. This also allows for simpler feet designs and modular feet.

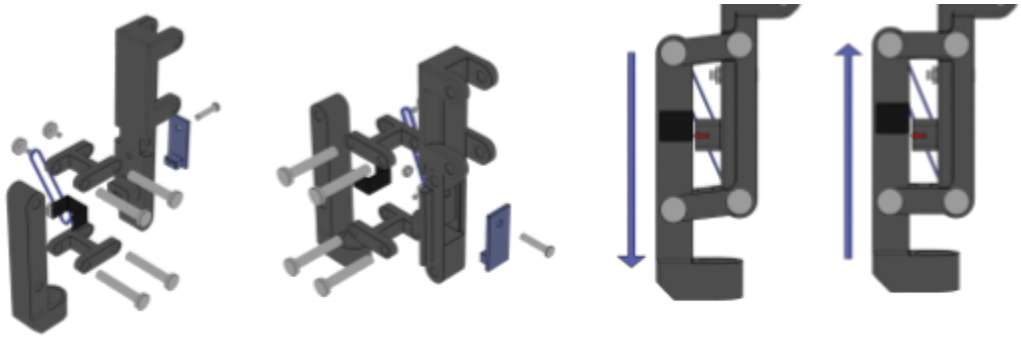

Fig. 2. Exploded view of lower leg and ground detection mechanism

For the robot to maintain its balance, it requires an IMU (Inertial Measurement Unit) to determine its orientation. With the help of the IMU, the robot can detect its roll and pitch and adjust its leg heights accordingly to compensate for any imbalance. The IMU uses I2C (Inter-Integrated Circuit) to communicate with the controller. I2C is a protocol used to communicate with multiple devices connected on the same bus, consisting of two wires. The IMU, however, only works if it is the only device on the I2C bus. Even including multiplexers, which are devices used to turn one I2C bus into several separate buses that can be selected individually to isolate different components. This is due to the way the IMU violates common

I2C rules to fit its own implementation. Because of this, the IMU is connected to a separate controller that then relays the IMU readings to the main controller via serial, another communication method.

The desired elevation of the robot is calculated based on the average vertical position of the legs. All the legs can be adjusted equally to change the average height. The average is usually kept around half of the legs' vertical range. Using this method, the robot body will always be positioned in a way that should allow each leg enough vertical range of motion. This should also prevent the body of the robot from moving too close to the ground.

The robot will detect objects to the front and front sides, using a time-of-flight (ToF) sensor or optional ultrasonic sensor mounted on a servo gimbal, allowing for vertical and horizontal adjustment. This allows for a wide range of view without the need to implement multiple sensors. A ToF sensor is used due to its speed, accuracy, small footprint, and ability to detect surfaces at a steeper angle than an ultrasonic sensor can. Support for an ultrasonic sensor was still included as a backup, however, since the ToF sensors may work poorly in some environments.

The servos are all controlled via a servo driver that supports up to 16 servos. The servo driver and ToF sensor are connected to the main Arduino using I2C and a multiplexer since they do not work well when on the same I2C bus. All the servos combined create a power-hungry robot, so it is operated off a 5200mAh 3S LiPo battery. This provides ample battery power for the rover to operate for several hours. An issue with using these batteries is that the voltage of the batteries is higher than the operating voltage of the servos, so a buck converter was used to decrease the voltage to the correct level.

Code

The code is structured using an object-oriented approach to simplify the structure and separate the different systems. This results in the code being more organized and manageable. Making iterations faster and changes easier. A UML diagram can be found in the Appendix.

A class is used for interacting with the IMU sensor. The IMU sensor itself is connected to a second Arduino so that it can have its own dedicated I2C bus. This second Arduino reads the IMU data and sends it over to the main Arduino via a Serial connection. The IMU class uses a custom data type (struct) for working with status, yaw, pitch, and roll data received from the IMU. An initialization method initializes the serial connection with the second Arduino and configures a pin to reset the second Arduino when needed.

Another method reads the data sent from the second Arduino and returns it as an instance of the struct mentioned earlier. The second Arduino sends the data as the status, yaw, pitch, and roll, each followed by an ‘s’, ‘y’, ‘p’, and ‘r’ respectively. The method obtains these values by reading data from the second Arduino until there is no more data to read. Each character read from the second Arduino is stored in a data value. When a letter is encountered, the data value is converted to a float and stored in the correct struct value, corresponding with the letter encountered. The data value is then reset to pick up new data. Once there is no more data to read, the latest readings have been acquired, and the method returns the struct containing the data.

The IMU sensor is rather unstable and can sometimes crash, causing it to send incorrect data and resulting in the rover losing the ability to level itself or correct its heading. To fix this, a method is used to reset the second Arduino when the IMU crashes. The IMU class controls a pin connecting a GPIO (general purpose input output) pin on the main Arduino to the reset pin on the second Arduino. During normal operation, the GPIO pin is turned on, resulting in the second

Arduino functioning as normal. When the main Arduino needs to reset the second Arduino, it turns the GPIO pin off for a moment and then turns it on again. This causes the reset pin to be triggered on the second Arduino, resulting in the IMU sensor rebooting and fixing itself.

The second Arduino is reset when garbage values are read from the IMU. The values that normally indicate a crash are 0 for the status, 0 for the yaw, -1 to 3 for the pitch, and 0 or a very large number for the roll. Whenever this pattern is detected, the method for reading data will reset the Arduino and recursively call itself until it gets a valid value. Once a valid value is read, it is returned. There is a chance that a valid value will match the garbage code, specifically if the status, yaw, and roll are all zero, but this is rare enough to not pose an issue with the intended application.

Whenever the IMU is reset, it sometimes results in the yaw measurements shifting. To account for this, each time there is a successful reading, it stores the yaw value measured. When a reset occurs, it adds the difference between the last yaw value read and the new yaw value to a yaw offset variable. Which can then be acquired by other classes to adjust the desired heading. To avoid multiple corrections for the same offset, the method that returns the yaw offset resets the offset value when it is called after it returns the yaw offset. This implementation is rather messy and has many potential bugs, but it was a quick solution that fit within the time constraints and worked well enough for the application.

Another class is used for handling and abstracting away the nuances of dealing with the servo driver. The driver can control the servos by being told to send a certain PWM (pulse width modulation) frequency to each servo to tell it what angle to adjust to. This class has fields that define the minimum, maximum, and offset frequencies for each servo. An initialization method is used to initialize the servo driver component.

Each servo's corresponding minimum and maximum frequencies are tuned to allow close to 180 degrees of rotation. The offset of each servo is to align it more precisely to the other servos. A method is used to set a servo's position, It takes two arguments, the servo number and the angle to set that servo to. This method converts the input angle to a frequency by getting the percentage that the input angle is of 180 degrees and taking that percentage of the specified servo's range of frequencies. The calculated frequency information is then sent to the driver to adjust the specified servo's angle. This allows for easily controlling the servo positions using angles instead of frequencies.

Another class handles interacting with the sensor mounted on the front of the rover. The sensor is mounted on a servo gimbal, allowing for vertical and horizontal rotation. This class defines two different struct types to simplify working with it. The vector3 struct stores the x, y, and z coordinates of a position, usually relative to the center of the servo gimbal. X represents horizontal position, y represents vertical, and z represents depth. The box struct contains two vector3s to define the minimum corner and the maximum corner of an area.

The scanner class has three modes, allowing it to work with one of two different ToF sensors or an ultrasonic sensor. This diversity is to allow for the rover to be adapted to different scenarios. The ToF sensors used do not work well in some environments, so ultrasonic support is included, though it is much slower. This class contains several methods for looking at a position, scanning an area, reading distance data from the sensor, and setting the gimbal's angles. An initialization method is used to initialize the distance sensor; it initializes the correct sensor based on a mode parameter passed to the method when it is called, and the mode is saved to a class field for later use. The area scanning method takes an area and checks if it contains an object or not by looking at points spread throughout the area and taking distance measurements. The

method that gets a distance measurement from the distance sensor uses the stored mode to interface with the correct sensor. Several readings are taken, and the average is returned to help filter out bad values.

The method for looking at a position uses trigonometry to find the correct angles to set the servos to, as well as find the expected distance to the inputted position. The angle of the horizontal servo is acquired using Equation (1).

$$\theta = atan(x / z) + 90° \quad (1)$$

Where θ is the target angle, and x and z are coordinates from the input position. Ninety degrees are added because the servos are centered at ninety degrees instead of zero. The vertical servo angle is determined using Equations (2) and (3).

$$\theta = atan(x / z) \quad (2)$$

$$\theta_2 = atan(y, z / cos(\theta)) + 90° \quad (3)$$

Where θ_2 is the target angle, and x, y, and z are coordinates from the input position. Again, ninety degrees are added for correct servo alignment. The expected distance is determined using Equations (2), (4), and (5).

$$\theta_2 = atan(y, z / cos(\theta)) \quad (4)$$

$$m = z / cos(\theta) / cos(\theta_2) \quad (5)$$

Where m is the expected distance, and x, y, and z are coordinates from the input position. After computing the target angles and expected distance, the method sets the servo angles and returns the expected distance. This distance can be used to detect an obstacle by taking a distance measurement after looking at a position. If the measurement is less than the expected distance, that position is blocked by an object.

Another class is used to represent a single leg in the code. Each leg is mounted onto a servo gimbal, allowing for horizontal and vertical movement. This class has methods to adjust, set, and get the gimbal angles and check if the leg is grounded. The methods used to set the gimbal angles set the leg's position by passing in a percentage that is interpreted as a position at that percentage of the leg's range of positions along the respective axis. The ground check method returns whether the leg is grounded or not. It checks the signal from the IR sensor mounted on the leg to determine if the leg is grounded or not.

The methods that set the gimbal angles use trigonometry to determine the correct positioning of the legs. This uses positions instead of angles to compensate for the change in travel per degree while rotating around the pivot point. The target horizontal angle is determined using Equations (6) and (7).

$$h = cos(90° - \theta) * h_2 \tag{6}$$

$$\theta_2 = asin(x / h) + 90° \tag{7}$$

Where θ_2 is the target angle, x is the target horizontal position, θ is the current vertical angle, and h_2 is the length of the linkage connecting the vertical servo to the 'knee' of the leg. θ is subtracted from ninety degrees to convert from the servo angle, centered around ninety degrees, to an angle centered around zero degrees. The vertical angle is determined using Equation (8).

$$\theta = asin(y / h) + 90° \tag{8}$$

Where θ is the target angle, y is the target vertical position, and h is the length of the linkage connecting the vertical servo to the 'knee' of the leg.

Another class is used to coordinate movements with multiple legs. It stores a list of legs and works with all of them simultaneously. It has all the same methods as a leg object for setting and adjusting a leg's position, just for all legs in the set, at the same time. It also has methods to get the minimum, maximum, and average horizontal and vertical positions, a method for grounding all its legs, and a method to check if all its legs are grounded or not. The grounding method moves down any non-grounded legs in the set until all the legs are grounded.

Another class is used to handle things like walk cycles, turn cycles, levelling, and course correction. An initialization method is used to initialize the leg positions and assign the initial heading for course correction.

The levelling method first reads data from the IMU to get the current pitch and roll values. If the pitch and roll are within a certain threshold, the method will end without executing any further logic. If the pitch or roll is not within a certain threshold, the code will go on to level the rover. First, the middle leg on each side of the robot is raised to allow for easier levelling. Then, while the pitch is out of the acceptable range, the pairs of legs at the front and back will be moved up or down to adjust the pitch. After the pitch is corrected, the same is done for the roll, using the left and right sets of legs. Once the levelling is complete, the front and back legs are adjusted so that the rover body is parallel to the ground. This is accomplished by taking the difference in minimum height between the back and front sets of legs and adjusting the back and front sets up or down until they have the same minimum height. This was added since the rover had difficulty climbing slopes and would just detect them as obstacles. After this is finished, the middle legs are returned to the ground, and the method ends. To be made more efficient, the section that levels the pitch could be removed since the pitch is adjusted later anyway to be parallel to the ground.

The method for correcting the heading begins by getting the yaw data from the IMU. If the current heading is beyond a certain angle from the initial heading, the rover will turn towards its initial heading until it is facing within a certain angle from the initial heading. Before each comparison with the initial heading, the heading variable is updated by getting the yaw offset from the IMU handler class. This ensures that any IMU crashes encountered when reading the current yaw value are accounted for.

An adjust height method is used to move the rover to a certain height, relative to the positions of its legs. This method takes in a percentage and adjusts all legs equally to where the average leg height is at the specified percentage of the legs' vertical range.

The step method uses a tripod gait so that there are always at least three legs on the ground to maintain balance. It can step forward or backward based on an input direction. The method first checks whether the last action taken was a step; if it was not, the legs are recentered, and the step phase is reset before taking a step. To take a step, the code works with two sets of legs, one set with the front and back legs on one side and the middle leg on the other side. The other set is the other three legs. Based on the phase that the step cycle is on, one set will be assigned as the up set, and the other as the down set. First, the up set is raised, then the down set is moved either forward or backward, based on the input direction. The up set is simultaneously moved in the opposite direction. Then the up set is set back on the ground, leaving the legs in the correct position for the next step phase, which is the same thing, just with the up and down sets switched.

The turn method takes a direction argument to specify either right or left. First, all legs are centered and grounded. Then, depending on the direction input, all legs on one side are moved back while all legs on the other side are moved forward. This method is rather inefficient;

however, due to the linear mounting of the legs, a tripod turning method does not work well. Given enough time, a better algorithm could have been developed, such as a tripod method that accounts for the nuances presented by the linear alignment.

The main navigation code ties all this together. A setup function is called by the Arduino when the program begins. In this method, it initializes the servo driver, scanner, IMU, and platform objects. In addition to the Serial and I2C connections. The rover is then leveled and adjusts its height to around half height.

There are three modes that the rover can be in: standard, blind, and test. Standard is the normal navigation mode. Blind is a fallback mode where the rover just walks forward and does not attempt to detect or avoid obstacles. Test mode varies depending on what is being tested but is mainly used so that the standard navigation code does not need to be modified to test anything. The mode is selected using a potentiometer before powering on the rover.

In blind mode, the rover will just walk forward. First, it takes a step forward, then levels itself if needed, then adjusts its height if needed, and this sequence is repeated. In standard mode, the robot will use more advanced features, such as obstacle avoidance, course correction, and getting itself out of corners in which it is “stuck”. First, the rover will check if it has turned left and right at least three times each, using left and right turn-counting variables. Since the turn count variables are reset each time a successful step is taken, if the rover has been turning left and then right repeatedly, it is most likely stuck. In this case, it will take several steps backward, then turn several times in the direction of the last turn. After each step or turn, the rover is leveled and adjusted to the correct height. The left and right turn counting variables are then reset to zero.

If the rover is not stuck, it will try to detect obstacles. First, the rover will check its left, and then its right. If either side detects an obstacle, the sensor stops, and the rover steps backward and turns away from the detected object. After each step and turn, the rover is leveled and adjusted to the correct height. The last turn direction and left/right turn counting variables are then updated accordingly. If both the left and right areas are clear, the rover will check the forward area. If there is an object detected, it will go through the same sequence as when an obstacle is detected on the left or right, except that the rover will turn in the same direction as the last turn. If no obstacles are detected, the rover will take two steps forward, levelling and adjusting height after each step. Then, the rover corrects its course if needed. Finally, the left and right turn counting variables are reset to zero, and the method repeats.

Results

A photo of the final design as tested in the robotics challenge courses is shown in Fig. 3.

Fig. 3. Front view of BioStrider robot while navigating a challenge course

Overall, the rover performed well, though it was a bit slow. Several minor adjustments were made in the code to account for issues encountered in the courses. The areas scanned by the rover were adjusted to better account for ground clearance, the distance measuring code was sped up by decreasing the readings taken for filtering, and the turning code was modified to get

more angular travel between obstacle detection cycles. The levelling code was modified to allow for the rover to adjust correctly to traversing steep inclines instead of just levelling its pitch. An obscure bug was corrected where specific legs would not correctly ground themselves in certain situations.

Ultimately, the rover met design constraints and goals. It weighed 3 kg, the components and printer filament totaled about $400, and the design was completed and tested in sufficient time to perform at the challenge. The unique goals of implementing insect-inspired locomotion and navigation were also realized by this design.

Recommendations

A hexagonal layout for the legs would have made turning easier to execute and would have left more legroom without needing to extend the body quite so much. The array of different sensors presented many challenges with system integration using just an Arduino, so a more powerful controller, such as a Raspberry Pi, could have made things easier. The added benefit of using a Pi over an Arduino is the ability to use Python instead of C++ for programming. Python is a high-level language with many quality-of-life features and is generally easier to work with than C++, which is a lower-level language requiring more manual implementation.

The modular feet performed well, but using granular feet similar to those developed by students at the Jacobs School of Engineering at UC San Diego [8] would have provided the rover with more traction on rougher surfaces and in tricky areas, such as when it was trapped inside one of the pits on course two of the challenge. Additionally, its turning time could be reduced and, in some cases, potentially eliminated with a secondary rear sensor, effectively converting the back of the rover into the front, with additional coding required.

References

[1] J. Stewart, "Hyundai's Walking Car Concept With Legs Reinvents the Wheel," Wired, Jan. 7, 2019. [Online]. Available: https://www.wired.com/story/hyundai-walking-car-elevate-concept-ces/

[2] "Insect Locomotion," Cronodon.com. [Online]. Available: https://cronodon.com/BioTech/Insect_locomotion.html. (Accessed: 24 January 2025).

[3] "Insect Locomotion," Department of Entomology, North Carolina State University. [Online]. Available: https://genent.cals.ncsu.edu/bug-bytes/thorax/locomotion/ (Accessed: 24 January 2025)

[4] P. Ramdya et al., "Climbing favours the tripod gait over alternative faster insect gaits," Nat. Commun., vol. 8, p. 14494, Feb. 2017, doi: 10.1038/ncomms14494.

[5] R. M. Noest, *Insect locomotion: Control, stability, and optimization*, Ph.D. dissertation, Graduate School, Cornell Univ., Ithaca, NY, USA, Aug. 2017.

[6] T. C. MacRae, "Colorado's Great Sand Dunes Tiger Beetle," *Beetles In The Bush*, Jan. 16, 2012. [Online]. Available: https://beetlesinthebush.com/2012/01/16/colorados-great-sand-dunes-tiger-beetle/ [Accessed: 13 Jan 2025]

[7] "AT-AT Walker," *Star Wars Databank*, Lucasfilm Ltd., [Online]. Available: https://www.starwars.com/databank/at-at-walker. [Accessed: 13 Jan 2025].

[8] E. Lathrop and M. T. Tolley, “These flexible feet help robots walk faster,” *Jacobs School of Engineering, UC San Diego*, Jun. 1, 2020. [Online]. Available: https://jacobsschool.ucsd.edu/news/release/3051?id=3051 [Accessed: 11 Mar 2025]

Appendix

Code UML Diagram