Development of Autonomous Octopod Programming

Matthew Troyer

Pueblo Community College

Southwest Campus

Faculty advisor: Melissa Watters

April 13, 2026

Abstract

The goal is to develop a program capable of performing autonomous navigation and environmental adaptation without human intervention for an eight-legged robot, or octopod. The robot should be able to move continuously to maximize efficient distance coverage. To meet the goal of autonomous navigation, the program procures distance readings and orientation data through various sensors, and controls eight independent legs for locomotion. The program makes use of inverse kinematics to simplify leg movement and sensor orientation. Using a procedural walk cycle enables the octopod to adapt to various positions without adhering to a rigid sequential walk cycle, enabling simpler control methods and greater reliability. Parallel processing is used to separate the locomotion system from the obstacle detection system which removes interference between the two systems, allowing for continuous motion and obstacle detection. An equivalent version of the program was developed in Unity alongside a simulation of the physical robot, to test various software elements without risking physical components.

Introduction

The octopod in question was developed for participation in the Colorado Space Grant Consortium Robotics Challenge by team Arachnid. For the challenge, the robot must operate autonomously to navigate sandy and rocky terrain, not dissimilar to that of Mars. The robot was limited to a maximum weight of 5kg and built with a budget of $500. The goal was to improve upon a previous hexapod design by improving navigation speed and implementing more degrees of freedom for smoother and more adaptable movement [1].

This paper focuses on the software component of the robot design. First is a section on how the software was tested, then a section on drivers, configuration, and communication between software components. Moving on, there is an explanation of the different objects and individual programs that make up the software. Lastly, there is an in-depth explanation of the robot's somewhat unique locomotion system, including the procedural walk cycle, inspired by procedural animation in video games.

Simulation

To test walk cycles and other algorithms without endangering expensive electronics, a simulation was developed in the Unity game engine. The simulation functionality is nearly identical to the robot firmware with the main difference being that it was programed in C# instead of Python. Using a simulation accelerated the testing and iteration of procedural walk cycle algorithms and inverse kinematics implementations.

Custom Drivers

32 Servo Driver

The robot uses 26 servos in total, and to control these servos, two PCA9685 16-channel servo drivers are used. The servo driver 32 class provides a simple interface for other classes to set servo angles based on an index in the range 0-31 and an angle in degrees. This driver handles individual servo offsets, PWM ranges, and angle limits, ensuring uniform servo operation and collision avoidance.

Inertial Measurement Unit

The Inertial Measurement Unit (IMU) driver is used by other classes to get readings from the IMU. Values obtained from the IMU are modified to account for yaw, pitch, and roll offsets. This driver also supports automatically resetting the IMU as needed.

File System

Flag System

The flag system is used for communication between the walk system and the sensor system. Any flag can be set, cleared, and read with just a name. Setting a flag creates an empty file in the flags directory with the specified flag name. Clearing a flag deletes the file with the specified flag name, if it exists. And reading a flag simply checks if a file with the specified flag name exists.

Configuration File

There are many user-defined values used for operation that must be calibrated and adjusted for testing and different operating scenarios. Instead of having these variables declared across all the different systems in the software code, they are defined in a configuration file that systems read from at runtime. Configuration entries are in the form “<name>: <value>”, and each entry is on its own line. Blank lines and commented lines containing “//” are also allowed in the file and are ignored by programs.

The function to read a configuration value takes in the name of the desired value and then iterates over each line in the configuration file. If a line is blank or contains “//” it is skipped,

otherwise it is split around the sequence ":" the first part is then stored as the name and the second as the value. If the name matches the specified name, the value is returned. If no line is found with a matching name, an exception is raised.

Operational Data

Operation data is additional data that must be communicated between systems at runtime, where flags are insufficient. Operation data follows the same format as the configuration file, but there is no need for blank lines or comments since it is handled entirely by the software. Reading operation data works the same as reading configuration data.

The function for writing operation data takes in the name of the data and the value to be written. The method then iterates over every line of the operation data file. If a line does not have the specified name, it is added to a list of read lines. If a line's name matches the specified name, the line is modified to represent the new value and then added to the list of lines. If all lines have been read and no match has been found, a new line is added to the end of the lines list with the specified name and value. Once all lines have been put in the lines list, the operation data file is overwritten with the modified line data from the lines list.

Servo Offsets

The 32-servo driver program uses a file to read PWM offset values for individual servos. Each servo offset is an integer on its own line, and line numbers correspond to servo indices in the driver. Each servo is offset slightly by an inconsistent amount due to mounting methods, using a file to store the correction offsets makes calibration much faster and easier than if it were

hardcoded into the driver. Using PWM offsets allows for very precise corrections, down to fractions of a degree, resulting in consistent servo positioning.

Objects

Leg

The leg object is used to represent a single leg of the octopod. The leg object has methods for converting three-dimensional coordinates between the global space of the robot and the local space of the individual leg. It also contains methods to read the grounded status of the leg and set the leg's position using inverse kinematics to position the foot at a target position.

Sensor

The sensor object is used to control the sensor assembly on the front of the robot. It contains methods to have the sensor look at a certain position and get a distance measurement. The "look at" method sets the roll and pitch of the sensor assembly so that the sensor is looking at a specified position. It then calculates the distance to that position and returns it. The "measure distance" method takes a user-defined number of distance readings and stores them in a list. The list is then sorted, and the median is returned as the measurement. Using the median of multiple measurements helps to filter out bad readings.

Programs

Shutdown Listener

The shutdown listener program runs in the background. When the power button is pressed and held for greater than half a second, the shutdown listener is triggered and powers down the processor cleanly and safely.

Walk System

The walk system program handles navigation logic. It uses a simple state machine for walking forward, backing up, and turning as shown in Fig. 1.

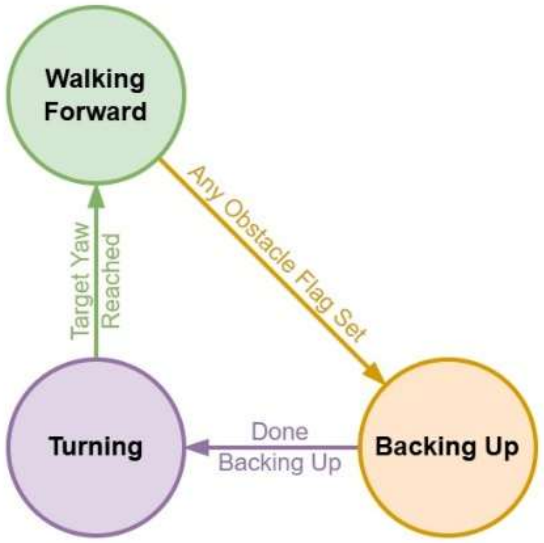

Fig. 1. Walk System State Machine

The default state is walking forward, and whenever an obstacle flag is set, a target yaw is generated by taking the current yaw of the robot and either adding or subtracting 30 degrees based on whether the left or right flag is set, respectively. The state is then set to the backing up

state, where the robot takes a set number of steps backwards and then transitions to the turning state. In the turning state, the robot rotates in place towards the target yaw, once the target yaw is reached it transitions back to the walk forward state and clears all obstacle flags.

Sensor System

The sensor system program handles obstacle detection. This program uses a loop system where during each loop a measurement is taken and evaluated and stores the position of the last measurement so that future loops can pick up where the current loop left off. The loop starts by reading the angles to the left and right frontmost legs and the height of the robot from operation data. It then uses the right and left angle data to generate rightmost and leftmost scan positions, to ensure that the sensor does not detect the frontmost legs as obstacles. The sensor works by sweeping from left to right and then right to left, and depending on the phase it will choose either the left or right position as the end target position (once the end target position is reached, the phase is switched so that the sensor will sweep in the opposite direction). A target position is generated by moving the last position towards the end target position by a user-defined sweep step. The sensor then looks at the target position, stores the calculated distance to that position, and takes a measurement. If the measured distance is less than the calculated distance, the current target position is blocked by an obstacle, and an obstacle flag is set. When a flag is set, the left flag is set if the sensor is looking left, the right flag is set if the sensor is looking right, and the forward flag is set if the sensor is not looking significantly left or right. The side angles and object detection zones are further detailed in Fig. 2.

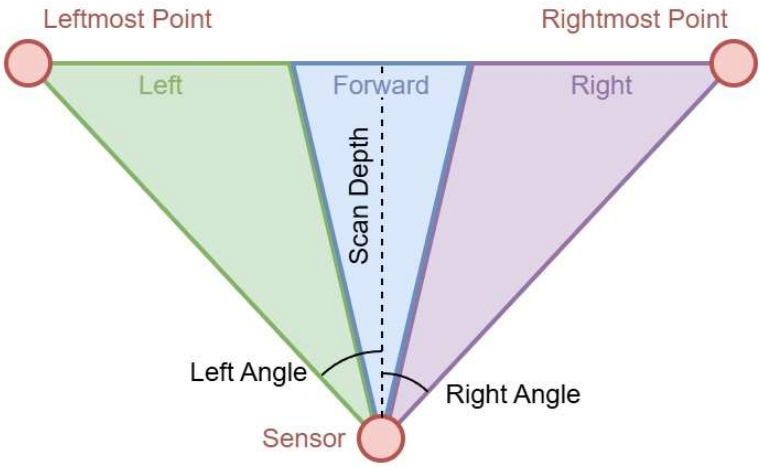

Fig. 2. Sensor Scan Field

Main

The main program runs on startup and is used to start and stop navigation, as well as switch navigation modes. The program listens for a start/stop button press and a mode switch button press. When the start/stop button is pressed, the main program will start both the walk system and sensor system programs if they are not currently running, otherwise it will terminate said programs. Running the programs in parallel allows them to run continuously without interfering with each other. When the mode switch button is pressed, if the robot is not navigating, the operation mode will be toggled between blind and navigation mode. Blind mode ignores all obstacles, and navigation makes use of obstacle avoidance. State machines for the aforementioned systems are present in Fig. 3.

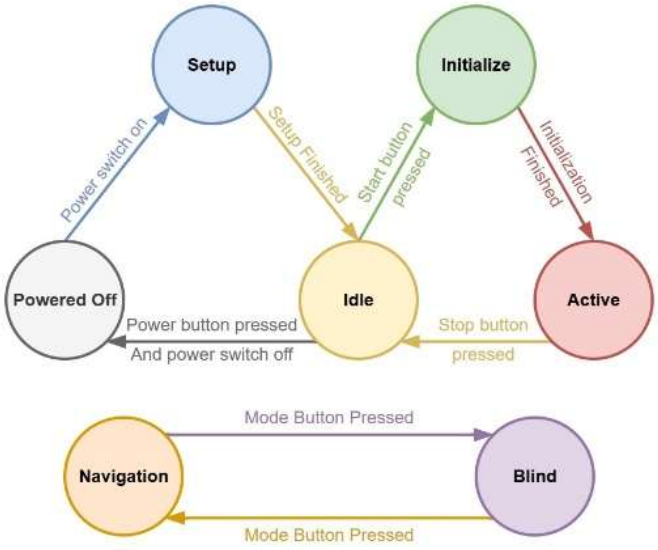

Fig. 3. Main System State Machine

Locomotion

Movement Pass System

The locomotion system is made up of several primary functions: a procedural walk cycle, a balancing method to keep the robot level, a method to keep the robot at the correct height, and a method to ground any legs that should be grounded and are not (due to the robot moving over uneven terrain). Each method is called and additively modifies a set of target positions for each individual leg, and once all methods have been applied, each leg is simultaneously moved to its respective target position. Each method is called many times per second based on a user-defined call frequency, so each function call only modifies the target position a little. A high call frequency results in smoother movements.

The balancing method makes use of the IMU to identify the robot's current orientation. It then iterates over each leg that is meant to be grounded and calculates its final target position. To do so it first checks whether the pitch is within a certain tolerance of the target pitch, and if not, it stores a position which is the result of rotating the current leg's current position around a point beneath the center of the rover at ground height. The same is then done for roll. The target position for the current leg is then linearly interpolated towards the calculated position by the movement step, which is calculated by multiplying the movement speed of the robot by a time step based on call frequency.

The height correction method first averages the height of all grounded legs and compares the average to a user-defined target height. It calculates the difference and interpolates the target position of each grounded leg by the movement step either up or down, to compensate for the difference.

The method for grounding all feet that should be grounded, first checks the state of each airborne leg. If a leg is supposed to be grounded, based on its state, its target position is moved downwards by the movement step.

Procedural Walk Cycle

In most applications, a fixed walk cycle can be used, however, due to the unknown ground and obstacle heights a procedural walk cycle is more suitable. A procedural walk cycle allows the robot to figure out how and where to place its feet to move to where it needs to go.

For this procedural walk cycle, the legs are grouped in pairs. Each leg works in sync with the leg it is paired with. There are four states that a leg can be in: waiting, ascending, descending,

and pushing, as shown in Fig. 4. When waiting, the leg does nothing until there is no other leg in another pair that is lifting or descending, at which point it transitions to the ascending state. When ascending, the leg will leave the ground and move towards its frontmost position, relative to the target direction, in an arc. Once at its frontmost position it will transition to the descending phase, where it simply moves straight down until it is grounded, at which point it will transition to the pushing phase. When pushing, the leg stays grounded and moves backward, relative to the target direction, and once it reaches the backmost position, it transitions back to the waiting state.

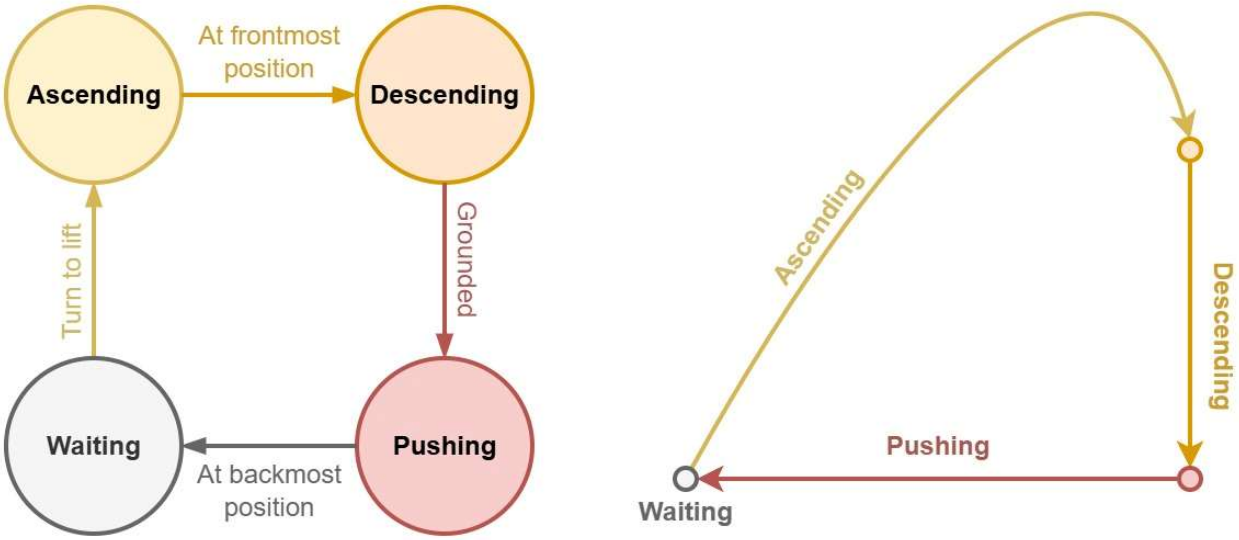

Fig. 4. Leg State Machine

If any leg is waiting while another is pushing, the pushing leg will do nothing until there is no leg waiting, this prevents the robot from dragging legs and allows the step sequence to form. Since there are eight legs in total, there are four pairs. To allow the legs to fall into step based on these rules, the ascending and descending phases last for only a third of the time that the pushing state lasts. This way, three pairs are grounded and pushing, while one pair is airborne. And by the time the airborne pair is grounded and ready to push, one other pair is ready

to ascend. When the pairs are out of phase, the waiting functionality allows them to become correctly synced again. The ideal walk cycle phases are shown in Fig. 5.

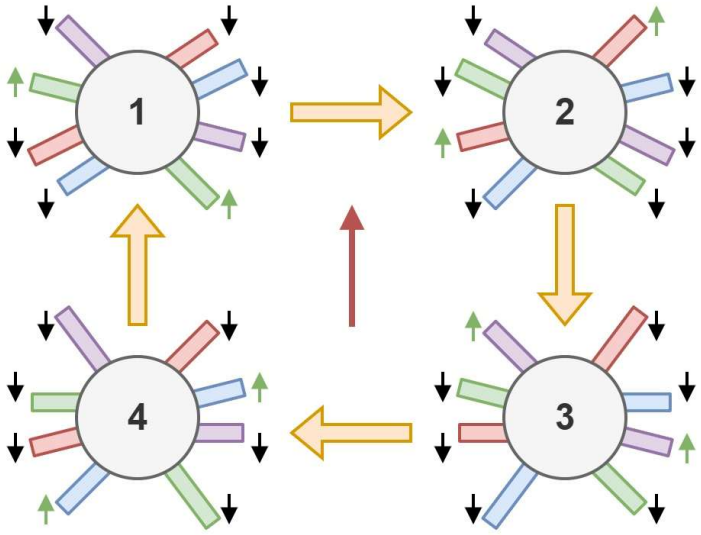

Fig. 5. Walk Phases

Rotation is very similar to moving linearly, except that legs move in arcs either clockwise or counterclockwise instead of straight lines as shown in Fig. 6.

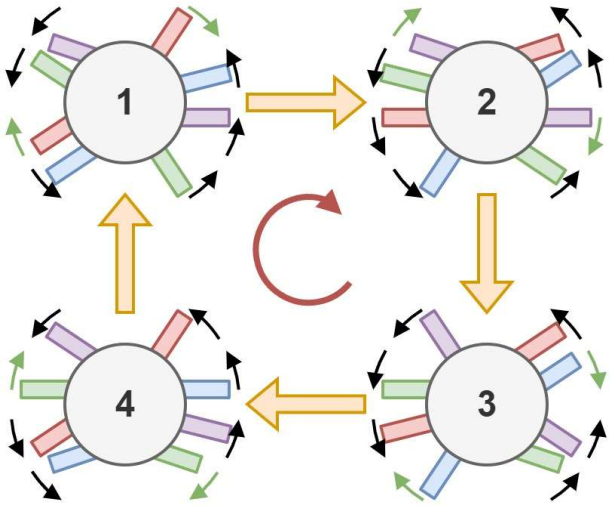

Fig. 6. Rotation Phases

There are two main functions that the rover uses for the procedural walk cycle, one to move the body linearly based on a two-dimensional vector that represents the target direction. And another for rotating the body, either clockwise or counterclockwise.

Several important user-defined variables used in the move function are the step length, the step height, the move speed, and the frequency at which the function is called. The step length and height affect step frequency and obstacle clearance. Both are balanced to maximize obstacle clearance and minimize step frequency, without compromising speed. The move speed affects how fast the robot moves and thus, how fast the sensor on the robot needs to scan for obstacles. Move speed is balanced to be as fast as possible without outpacing the sensor. The frequency at which the function is called affects the smoothness of robot movements. The more times the function is called per second, the less the function moves the robot per call, resulting in a smoother appearance, the frequency is balanced to allow for maximum smoothness while not being slowed down by how fast the processor can execute a single call.

The move function begins by calculating the step length and direction, this is done by normalizing the target direction and multiplying it by half the step length. It then calculates how much of a step to execute by multiplying the move speed by a time step, based on call frequency. Next, the function iterates over each pair of legs and sets two booleans, one to indicate whether any other leg is waiting, and one to indicate whether any leg in any other pair is ascending or descending. Then, in a nested loop, it iterates over each leg in the pair.

If a leg is pushing, it checks to see if its paired leg is waiting, if so, the current leg's state is also set to waiting and it continues to the next leg (to keep paired legs synced). It then checks to see if any other leg is waiting, if not, the leg's target position is moved opposite to the target direction. It then checks to see if the leg is at its backmost position, if so, it sets the state of the

current leg to waiting. If a leg is waiting, it does nothing, unless no other leg in another pair is ascending or descending, in which case it sets the state of the current leg to ascending. If a leg is ascending, its target position is moved towards its frontmost position by interpolating along a curve, based on the legs current position along the curve. It then checks to see if the leg is at its frontmost position, if so, the state of the current leg is set to descending. If a leg is descending, the leg's target position is moved downward, it then checks to see if the leg is grounded, and if so, the state of the current leg is set to pushing.

The turning function uses a lot of the same user-defined variables as the move function, but instead of step length, it uses a rotate angle, which serves a similar purpose.

The turning function begins by calculating the step angle and direction (clockwise or counterclockwise), this is done by multiplying the direction (-1 or 1) by half the rotate angle. It then continues similarly to the turning function.

The logic for the leg state machine is the same as in the turning function, except the pushing state. Instead of moving linearly in the opposite direction of the target direction, it is rotated along an arc opposite the target rotation.

Conclusion

Overall, the software met the goals of continuous and adaptable movement with autonomous navigation. The choice to use parallel processing greatly simplified the coordination of processes and allowed for a more modular design. One improvement to be made for future programs is to implement a more advanced and reliable obstacle detection and avoidance system, like SLAM (Simultaneous Localization and Mapping) or course mapping and pathfinding. The

usage of only a single sensor for obstacle detection simplified the design to a degree but ended up being the bottleneck when it came to how fast or reliably the robot could navigate.

References

[1] J. Nez, E. Thompson, and M. Troyer, *"BioStrider: Design of a Six-Legged Autonomous Robot for Efficient Navigation in Sandy and Rocky Terrain,"* Pueblo Community College, Southwest Campus, Apr. 14, 2025. Accessed: Apr. 13, 2026. [Online]. Available: https://www.colorado.edu/center/spacegrant/media/1104