AUTONOMOUS WALKING ROBOT

Isara Cholaseuk (isara@bu.edu)
Thanadol Sangprasert (tontun55@bu.edu)
GitHub repo link: https://github.com/isarachol/EC535_FINAL_PROJECT.git
Youtube link: https://youtu.be/4r_WLDysyvo

Abstract

Navigation tasks require both a robust locomotion system and perception. Legged robots are more robust at traversing complex environments, such as uneven terrain or steep inclines, because they can adapt their stable configuration to the environment. This makes them more suitable for navigation tasks. The goal of this project is to develop a legged robot that performs object-tracking tasks without colliding with obstacles. We designed a legged robot that can navigate to a predetermined location marked by a geometrically shaped, colored marker. We use a camera to perform object detection to detect the marker through a personal computer and send commands through a local network to the BeagleBone, which controls the motor. We also implement obstacle avoidance using an ultrasonic distance sensor to halt motion when an obstacle is detected. The final prototype is tested in a simplified environment with even terrain and white lights. The robot can detect predefined markers, walk to them, and stop before colliding. In the future, we intend to improve the robustness of object detection algorithms across different light conditions, enhance the mechanical design, and introduce a more sophisticated motor controller.

Introduction

Many autonomous systems rely on computer vision to navigate complex environments. Some systems utilize a fixed camera as a third observer to control and coordinate the robots to achieve the task. The benefit of using a fixed camera is that it helps filter out uncertainty. Since the camera does not need to move with the robots, it does not experience the vibrations encountered during robot movement. However, the critical disadvantage of the fixed-camera method is its limited operational space. The robot can only operate in a known or predefined workspace, resulting in degraded generalization. A potential approach to achieving true autonomy is the use of an onboard camera. The robot must be able to possess egocentric perception; the ability to see and interpret the world from its own perspective. This project focuses on developing an autonomous walking robot equipped with an onboard camera system. By integrating vision directly onto the robot's body, it offers a self-contained agent capable of detecting objects and making real-time navigational decisions to move forward, left, right, or stop.

There are three motivations behind the autonomous walking project. First, most of the environment and infrastructure are designed for bipedal animals, which are humans. Vertical objects such as stairs, curbs, and door thresholds are optimized for legs rather than wheels. As a result, traditional wheeled or tracked robots face challenges in traversing those environments. By implementing bipedal walking, the robot aligns its locomotion with the geometric logic of the built environment. The robot gains the mechanical capacity to step over debris, reach various heights, and navigate discontinuous terrain. This capability is critical for robots intended to assist humans in domestic environments, such as houses.

Second, an onboard camera gives operational independence and scalability. The fixed camera limits the scalability of a robotic swarm. The swarm of robots can face failure if a set of fixed cameras fails. The fixed camera system functions as a centralized system: the entire system goes down when the central hub malfunctions. On the other hand, an onboard system creates a decentralized framework. Each robot acts as an independent node, processing its own data. The decentralized structure ensures that the failure of one part of the system does not compromise the overall task. In addition, it allows the system to scale infinitely without requiring an enormous amount of external cameras to cover the operating area.

The last motivation is overcoming noise from the onboard camera. The main challenge of using an onboard camera is the rhythmic oscillation it suffers from. While the fixed camera suffers from the occlusion from the blind spot, the onboard camera suffers from oscillation and vibration caused by the walking gait. This is called active noise and can significantly degrade the robot's perception by introducing blur and jitter. Developing a robust imaging process enables the robot to maintain precise situational awareness. Solving this problem is crucial to transitioning from static to dynamic observation and increasing real-world interaction.

This project consists of two major parts: hardware and software. For hardware configuration, the physical platform is designed to balance the weight between the two legs. The robot uses 3D-printed parts for a bipedal chassis driven by two servos at the legs. Due to the item constraint, the robot needs a third servo to assist with steering. The robot uses a BeagleBone as its central embedded microcontroller to manage low-level motor control. A camera module is mounted on the upper chassis. The robot is also equipped with a distance sensor, preventing it from crashing into objects. For software configuration, the autonomy stack consists of three layers: perception, the bridge, and the low-level controller. The perception layer is run on the external computer, ensuring robust data computation. This layer performs object detection, (both color and shape) and calculates the error of the object's center from the center of the frame. Then it sends commands via TCP socket. The bridge or userspace connects to the server, receives commands, and passes them to kernel space. The low-level controller, or kernel space, parses the transmitted commands and controls three servo motors to accomplish the task.

The current achievement of this project is the ability to follow the predefined objects without crashing into them. The robot can track and move towards objects based on user input: a combination of color and shape. The robot can also turn left or right based on the position of the tracked object.

Design Flow

In this section, we provide a high-level overview of the logic, hardware, and software modules. We first determine how the robot would operate logically at the high level.

Logic

The goal is to develop a robot that walks toward a desired object without crashing into it. This means the robot needs the ability to detect objects with specific descriptions, process their relative positions, and actuate to move itself while measuring the distance to objects/obstacles in front of it. This led us to divide the robot into two subsystems: the object detection subsystem and the sensing-and-actuation subsystem. The flow chart in Figure 1 shows the high-level logic of the implementation.

For the object detection subsystem, we decided to use classical image processing techniques to detect objects of specific colors and shapes. Once the robot 'knows' where the object is relative to its position, it can determine the direction of motion that would bring the object to its center of frame.

For the sensing-and-actuation subsystem, we perform a periodic actuation. Depending on the direction determined by the object detection subsystem, the robot's legs would move by a set amount in each cycle. At the same time, the distance sensor would have to measure the distance in front of it every cycle, and if the distance is less than a certain threshold, the robot would have to stop to prevent a collision. After determining the high-level logic, we designed the hardware and software to support the logic.

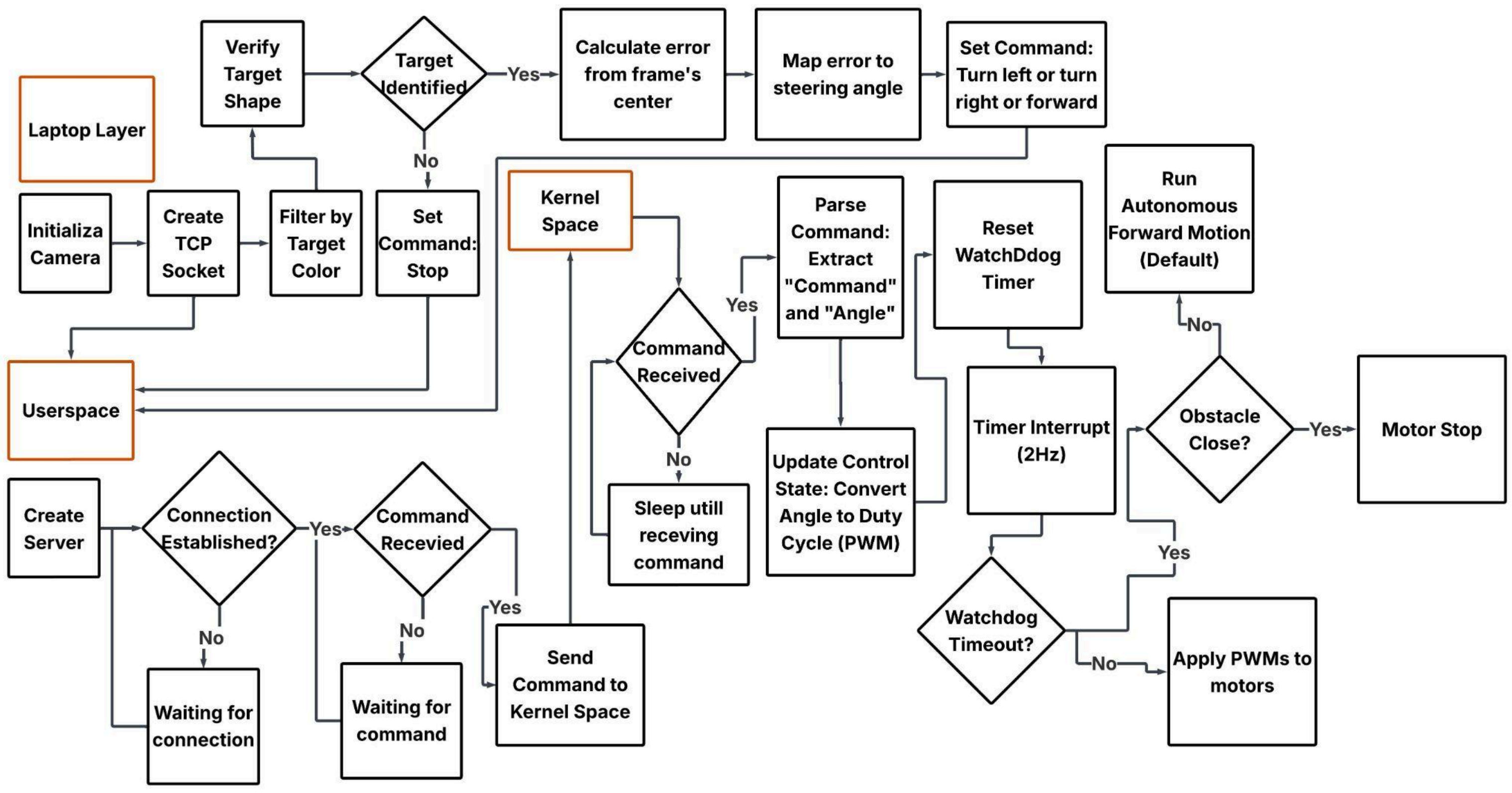

Figure 1: High Level Logic flow diagram

Hardware Design

The available components are listed in Table 1. The first subsystem includes a camera and a personal computer. Because the object-detection software we use, OpenCV, primarily supports the C++ API, we decided to run this subsystem on a personal computer. The second subsystem includes a distance sensor and three servo motors, which can be written in C and run on BeagleBone. Once the personal computer processes the image, it sends a command through an Ethernet cable to BeagleBone, which then controls the servo motors. The physical setup is illustrated in Figure 2.

We designed the mechanical mechanism to enable the robot to walk using 3 servo motors. The moving mechanism uses two side legs to push the robot forward. While turning, only one leg propels the robot, while the other is suspended in the air, and the rear wheels turn to help steer the robot in the desired direction.

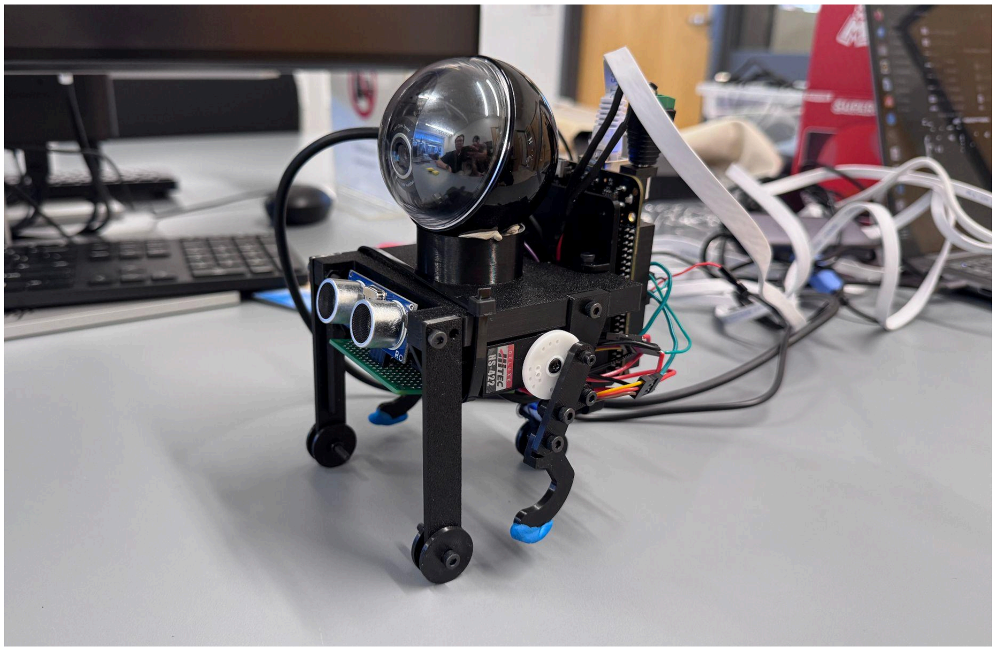

Figure 2: The physical robot hardware

Table 1: Bill of Materials

Item #	Name	Quantity
1	BeagleBone Black	1
2	Distance sensor (HC-SR04)	1
3	Camera (Logitech Orbit AF)	1
4	Servo motors	3
5	Ethernet cable	1
6	Personal computer	1

Software Design

Based on this logic, the object detection subsystem can operate independently, determining the direction of movement from the input image and sending it to the sensing-and-actuation subsystem. The direction output is sent as a formatted ASCII string (e.g., "FORWARD:90") through a local network using TCP protocol. The real-time responsiveness is critical for this project. The TCP_NODELAY socket option is implemented to disable Nagle's algorithm, forcing the immediate transmission of the commands rather than buffering them. This signal, however, cannot be passed directly to the kernel program on the BeagleBone that handles the sensing-and-actuation subsystem, so we created a userspace program that receives the signal and writes it to the kernel module on the BeagleBone.

Contribution

- Thanadol developed the object detection algorithm, the communication protocol between the personal computer and BeagleBone, including the userspace program.
- Isara developed the sensing-and-actuation kernel module, the mechanical design, and the circuit design of the robot.

Each team member contributed to 50% of the project.

Project Details

In this part, we discuss the detailed developments of each subsystem. The developments are divided into three sections: object-detection subsystem, sensing-and-actuation subsystem, and circuitry.

Object-Detection Subsystem and Userspace communication

The object detection subsystem serves as the robot's primary perceptual interface, running on an external laptop to leverage higher computational power for image processing. The software is implemented in C++ and utilizes the OpenCV library for image processing and the Linux Socket API for network communication [1].

For image pre-processing and HSV conversion, the system captures video frames from a standard USB camera. To ensure the object detector can detect under various lighting conditions, the raw BGR image is converted to the HSV color space. HSV enhances detection by separating color information (Hue) from light intensity (Value), making detection more robust [2]. We decided to use specific threshold ranges for our targets: Yellow: Lower (15, 100, 100) and Upper (35, 255, 255), and Blue: Lower (90, 60, 30) and Upper (140, 255, 255). However, there is one challenge happening during the experiment. The camera detects the noise following the initial masking process. Although the HSV thresholding successfully isolated the target colors, environmental factors produce Salt-and-Pepper Noise, scattered small white pixels within the binary mask. The noise leads to false positives which the contour detection algorithm might interpret a cluster of noise pixels as a valid target, causing the robot to steer towards a non-existent object. The solution is to implement noise filtering before the contour algorithm. First, an erosion operation is implemented with 1 iteration. The erosion operation removes the white layer, eliminating small, isolated noise. This operation also causes the targeted object to shrink, losing its white edges. To restore the target size, the Dilation operation is implemented with 3 iterations. The dilation operation reexpands the current valid regions, restoring target objects to approximately the same size and filling any internal holes [1]. This sequence effectively removes noise, ensuring that only legitimate targets are forwarded to the next algorithm.

After the color detection and noise reduction, the system performs the shape classification. The algorithm uses the Douglas-Peucker algorithm (approxPolyDP) to approximate the contour's geometry [3]. The selected target is a triangle and a square. After the geometry approximation, the algorithm counts the number of corners: 3 vertices are identified as a triangle, and 4 as a square. After all the processes, the system would be able to detect 4 combinations of the target object: a blue triangle and a square, and a yellow triangle and a square.

The Object-Detection subsystem also sends a navigation command to the BeagleBone. The navigation command relies on lateral error. The algorithm computes the distance between the centroid of the detected object and the center of the camera frame. To prevent robots from rapidly changing direction, a dead zone is implemented. The current defined dead zone is 50 pixels, determined through trial-and-error. If the distance is greater than 50 pixels, the turn-right command is sent. If the negative distance is less than -50 pixels, the turn left command will be sent. Otherwise, the forward command will be sent.

The userspace running on the BeagleBone serves as a bridge between the high-level vision system and the low-level kernel driver. The Linux Kernel cannot directly receive or handle standard TCP socket protocols [4]. Due to time constraints, the Ethernet port is used for communication rather than wireless. The userspace program configures the BeagleBone to act as the TCP server, binding to Port 5000. This makes the BeagleBone able to listen for incoming commands. Since the Ethernet port is used for a direct connection, the IP address needs to be manually assigned to the network interface. The BeagleBone's IP address **192.168.7.2** is configured as a static IP address. This ensures that the laptop acting as a client has a consistent destination address. There is one additional challenge beyond algorithm development. In reality, the program might need to be terminated to prevent unpredictable behaviour. Typically, a TCP server holds the port in **TIME_WAIT** for a specific period after the server terminates. As a result, it blocks any new program from binding to Port 5000. This structure can significantly delay the testing cycle. The **SO_REUSEADDR** socket option is implemented. This allows the userspace to bind to the port immediately after restart [4].

However, there is one significant issue. The latency of sending data from the laptop to the userspace and activating the kernel is significant. A standard TCP connection buffers small packets to improve bandwidth efficiency. This structure can introduce a delay that causes the robot to lag behind reality and to respond too slowly to real-time input. **The TCP_NODELAY** option is implemented in client code, which disables Nagle's algorithm, forcing the immediate transmission of every motor command [4]. The userspace includes a signal handler for **SIGINT** (Ctrl + C) and **SIGTERM, which are** used to close the network socket and release all file descriptors.

After the TCP server is created and the command is received, the userspace program writes the buffer to the character device file **/dev/walker**. As a result, the string commands are passed down into the kernel space for motion actuation.

Sensing-and-actuation Subsystem

As discussed before, this subsystem is implemented inside a kernel module and controls the distance sensor and the servos. In this subsystem, we developed a character device module that implements a timer callback function that executes every fixed period of time, which is where the main implementation takes place. Inside the callback function, the distance sensor is triggered, and if the distance is not too close, the PWM signals for the servo motors are configured.

Distance sensor

Every time the distance sensor is triggered, it records its time in nanoseconds from the moment it boots up (call this trig_time). This is done at every timer cycle. Once an ultrasonic wave reflects off an object and returns to the ultrasonic sensor, the echo pin triggers an interrupt on the GPIO pin, which is configured as an interrupt pin. This interrupt signal would trigger a handler function, which measures the time in nanoseconds from the moment it boots (call this echo_time). The difference between trig_timer and echo_time (call this elapsed_time) can be used to calculate the distance to the object the sound is bouncing off of, assuming the sound travels at 343 m/s.

There are two problems associated with this method. First, traditional jiffies count every 4000 microseconds, which is not fast enough for the microsecond resolution of the average trig-echo elapsed time (as a reference, a 10 cm object would be measured in 583 microseconds). So, we turned to *the ktimer* library, which provides an API to get nanoseconds from boot time (ktimer_get_ns()) in **<linux/ktimer.h>** [8]. Second, because the trigger mechanism works by setting a GPIO pin to HIGH, then LOW for a very brief moment (10 microseconds), we used the udelay() function to delay the operation. We attempted to time the elapsed_time with the beginning of the HIGH trigger and the rising edge of the echo interrupt. The result turned out to be highly inaccurate (calculated distance was either too high or 0). We then switched to using the elapsed_time between the end of the HIGH trigger and the falling edge of the echo interrupt. We were unable to find a conclusive reason for this difference. However, we hypothesize that there is a transient between the onset of the HIGH trig signal and the onset of the HIGH echo signal. In contrast, this behavior is minimized at the falling edge. This transient behavior can be caused by the diaphragm's physical properties or by the circuits.

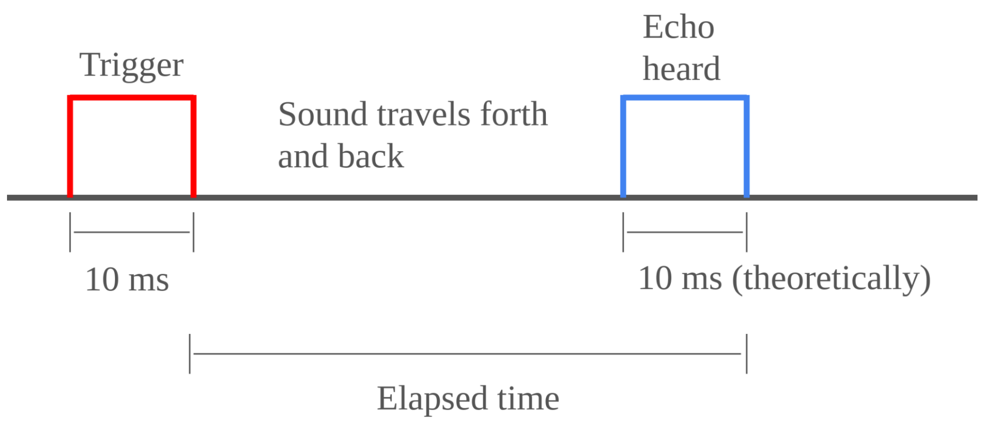

Figure 3: Visualization of trigger and echo mechanism for the distance sensor where the elapsed time is measured at the falling edge

Servos Control

The servo motors are controlled by PWM signals, with the duty cycle determining their rotational position. According to the datasheet [11, 12], the PWM signal period is 20,000 microseconds, and the on-time is in the range of 500-2500 microseconds. We tested the three servo motors with PWM signals in those ranges. We determined that on-times of 600-2400 microseconds can achieve near 180 degrees displacement, with 1500 microseconds resulting in the center position, as [11, 12] suggested. So, we use this range of PWM signals to manipulate the servo motors.

Since the legs only move back and forth, we use only two PWM states, with on-times of 600 and 2400 microseconds. For the servo motor attached to the robot's back wheel, there are three states for steering the robot left, right, and straight forward, which correspond to 1050, 1950, and 1500 microseconds of on-time for the PWM signal.

While implementing the PWM signals, we encountered two problems: configuring pins as EHRPWM pins and implementing PWM signals from kernel space. We first followed a tutorial from [9], which configured the PWM pins using bone_capemgr slots, but our file system does not support them. We then looked at [10] and found a precompiled overlay that configures our PWM pins in /lib/firmware on the BeagleBone. These files are BB-PWM0-00A0.dtbo, BB-PWM1-00A0.dtbo, and BB-PWM2-00A0.dtbo. Each of these overlays configures a specific PWM chip that enables only two EHRPWM pins. To apply six EHRPWM pins, we can add the following lines inside the file /boot/uEnv.txt:

```
uboot_overlay_addr1=/lib/firmware/BB-PWM0-00A0.dtbo
uboot_overlay_addr2=/lib/firmware/BB-PWM1-00A0.dtbo
uboot_overlay_addr3=/lib/firmware/BB-PWM2-00A0.dtbo
```

We can then configure its period or duty cycle through either the userspace SYSFS interface or through the kernel space. We first tested in the userspace to see which pins can be used. We use the following commands to check, enable, and configure the PWM period and duty cycle.

```
ls /sys/class/pwm/ # should show pwmchip1 pwmchip2 ...
echo 0 > /sys/class/pwm/pwmchip1/export # export dev number 0 in pwmchip1
echo 20000000 > /sys/class/pwm/pwmchip1/pwm-1\:0/period # set period in ns
echo 1500000 > /sys/class/pwm/pwmchip1/pwm-1\:0/duty_cycle # set on time
echo 1 > /sys/class/pwm/pwmchip1/pwm-1\:0/enable # send PWM signals
```

This allowed us to map the export number to pin numbers of the BeagleBone.

The second problem is implementing the PWM signals in the kernel space. There are two methods of implementing it in the kernel space. The first one is done by using or creating an overlay that configures specific PWM pins with specific names using Device Tree. After spending a significant amount of time on the device tree, we realized we needed the { compatible } key for the overlay we were using, which was pre-compiled. This was quite impossible as we could not go through thousands of files inside the stock image to find where the source overlay is, if there is even one. So, we decided to use legacy API commands provided in **<linux/pwm.h>** [7]. We followed a tutorial [6] to set up the code and performed trial-and-error to determine the ID numbers of the EHRPWM pins. (The IDs are representations of the pins, which can be looked up from the device table, which is set through the overlaying Device Tree.) We run the code with **pwm_request(ID, 'pin name')** using IDs ranging from 0 to 5. Table 2 summarizes our experimentation.

Table 2: PWM IDs and their corresponding physical pins (See section *Circuitry* for pin explanation)

PWM IDs	BeagleBone Pins
1	P9_22
2	P9_21
4	P9_14
5	P9_16

With these pins, we are able to request, enable, and configure the PWM signals for each pin using the legacy API.

Circuitry

The first consideration is power delivery. In the object-detection subsystem, the camera is a webcam designed to be powered by a personal computer, so we do not need to design additional circuits. For the other subsystem, while the BeagleBone operates at 3.3V logic, the BeagleBone itself, the distance sensor, and the servo motors require 5V input power. To fix this, we use an external 5V power supply with a common ground. Figure 4 shows the wiring diagram with pin numbers on the BeagleBone. The pin numbers on the left are labeled P9_##, while the pin numbers on the right are labeled P8_##, which are referenced from [5].

To control the distance sensor, we need one output pin to trigger an ultrasonic wave and one input pin to receive the echoed ultrasonic. We connect its trig pin to a GPIO pin on the BeagleBone and set it to output mode. The 3.3V signal is strong enough to trigger the ultrasonic signal. For the echo pin, however, we created a voltage divider circuit to reduce the original 5V to 3.3V before connecting to another GPIO pin, as shown on the right of Figure 4. The two pins can be any GPIO pins; we chose P8_15 (trig) and P8_17 (echo).

For the servo motors, the PWM signals can be implemented with 3.3V, so they are connected directly to the PWM pins. As discussed above, we selected P9_14, P9_16, and P9_22 pins, which are

Enhanced High-Resolution PWM pins dedicated to implementing PWM signals. The servo motors' connections are shown on the left of Figure 4.

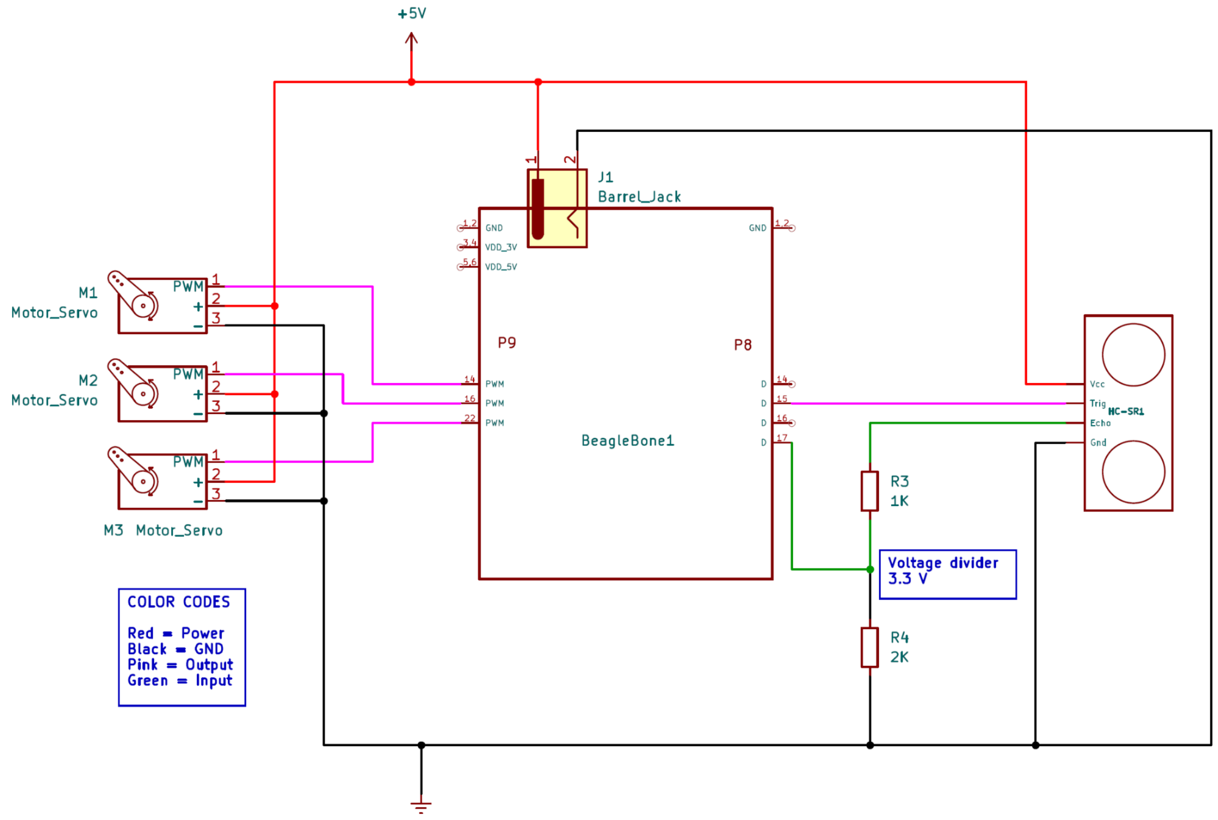

Figure 4: Circuit Diagram of the sensing and actuation subsystem

Summary

The project's main goal is to develop an autonomous walking robot. The robot can perform object detection to extract the target object from the background based on the user input. After performing the object detection, the robot can follow and move towards the target object within the predefined distance.

Based on the experiments, the robot is able to perform object detection based on both color and shape. Blue and yellow colors, along with triangle and square shapes, are selected as the main targets for the robot. The robot successfully detects all tested combinations and executes forward movement toward the selected target. The robot maintains a consistent heading and adjusts its path in real time to keep the target centered in the frame. Namely, the robot successfully executes the right and left commands using two servo motors at the leg and one servo motor at the back.

However, two main challenges remain in this project, regardless of the item constraint. The first challenge is robustness in object detection. Since the main experiments were conducted under consistent white light, color segmentation degrades significantly when ambient lighting changes. The second challenge is walking stability. The current setup heavily relies on wired connections: an Ethernet cable for the TCP socket server and a USB cable for the camera. While the robot maintains its stability, the motors lack sufficient power to move with all the cables attached. As a result, the robot cannot move forward, left, or right efficiently without human assistance to lift and manage all the cables.

References

[1] A. Kaehler and G. Bradski, *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. Sebastopol, CA: O'Reilly Media, 2016.

[2] R. C. Gonzalez and R. E. Woods, *Digital Image Processing*, 4th ed. New York, NY, USA: Pearson, 2018.

[3] D. H. Douglas and T. K. Peucker, "Algorithms for the reduction of the number of points required to represent a digitized line or its caricature," *The Canadian Cartographer*, vol. 10, no. 2, pp. 112–122, 1973.

[4] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming, Volume 1: The Sockets Networking API*, 3rd ed. Boston, MA: Addison-Wesley Professional, 2003.

[5] “Beaglebone Black Pin Map.” *MATLAB & Simulink*, www.mathworks.com/help/matlab/supportpkg/beaglebone-black-pin-map.html.

[6] Johannes4Linux. “Let’s Code a Linux Driver - 16: How to Write a Linux Device Driver for a Device Tree Devices.” *YouTube*, YouTube, www.youtube.com/watch?v=yLm4EDVNceo.

[7] “The Linux Kernel.” *Parallel Port Devices - The Linux Kernel Documentation*, www.kernel.org/doc/html/latest/driver-api/miscellaneous.html#c.pwm_get.

[8] “The Linux Kernel.” *Ktime Accessors - The Linux Kernel Documentation*, docs.kernel.org/core-api/timekeeping.html.

[9] Briancode. “Working with PWM on a Beaglebone Black.” *Adventures in Programing*, 7 Jan. 2015, briancode.wordpress.com/2015/01/06/working-with-pwm-on-a-beaglebone-black/.

[10] Cooper, Justin. “Introduction to the Beaglebone Black Device Tree.” *Adafruit Learning System*, learn.adafruit.com/introduction-to-the-beaglebone-black-device-tree/exporting-and-unexporting-an-overla y.

[11] *SERVO MOTOR SG90*, www.ee.ic.ac.uk/pcheung/teaching/DE1_EE/stores/sg90_datasheet.pdf.

[12] *Sparkfun*, cdn.sparkfun.com/datasheets/Robotics/hs422-31422S.pdf.