ENGR-E321

Advanced Cyber-Physical Systems

Section: 19882

Final Project: Vision Tracking Robot

Due: December 13, 2021

by AJ Funari, Blake Dunaway

Table of Contents

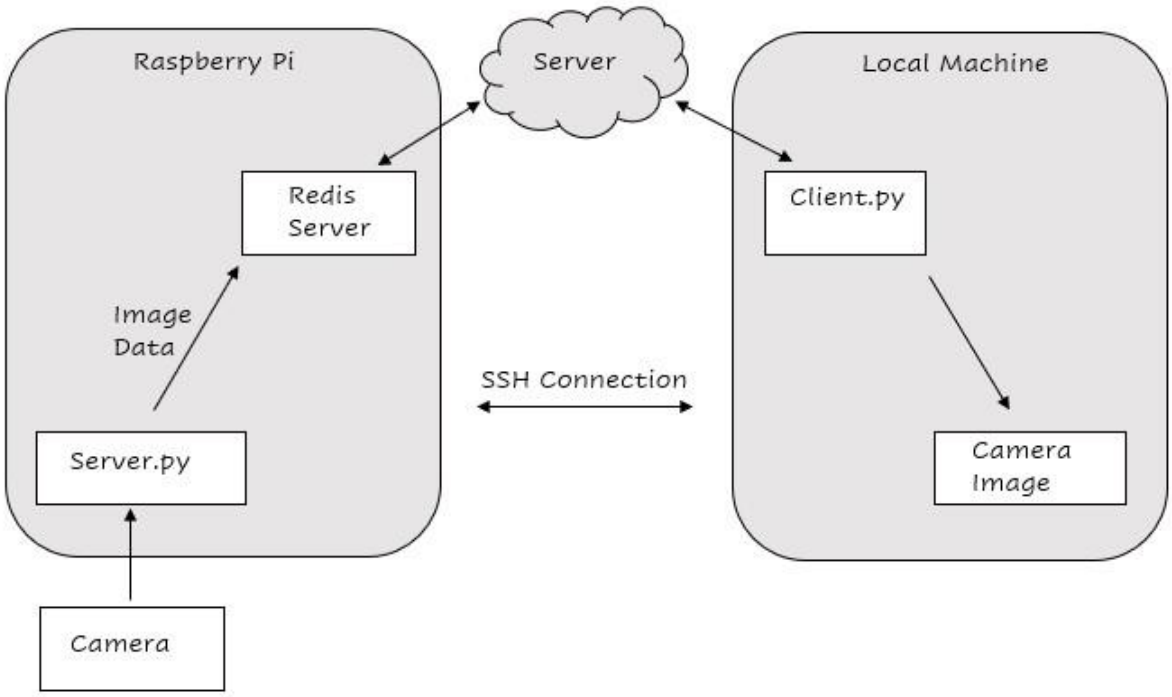
System Description

The main hardware components of the system are the raspberry pi, two chassis boards, and a camera. When fully constructed, the resulting robot is called the AlphaBot2. On the Alphabot2-Base, the bottom chassis board, there is a battery holder that supports two 14500 rechargeable batteries. At the front of the bottom chassis board are two reflective infrared photoelectric sensors for obstacle avoidance. A FC-20P 8cm cable connects the AlphaBot2 control interface between the top and bottom chassis boards. As for the top chassis board, a Raspberry Pi Model B+, using a 32GB micro SD card for memory, is connected to the Raspberry Pi interface. Also connected to the Raspberry Pi is the camera which is connected into the CSI (Camera Serial Interface). The camera is secured to the top of the chassis board with a 3D printed camera mount.

Installed on the micro SD card is the Raspberry Pi Operating System 32 bit, which is the software component to the system. Python is installed on the operating system which enables us to install necessary packages and run python scripts. Also installed on the operating system is Redis. Redis is a remote dictionary server used to send data through a server. For our system, we sent image data through Redis instead of an SSH connection, bypassing X11, resulting in a faster computation time. Displaying the image data through Redis resulted in a smoother image than if the image data was sent through the SSH connection. The mechanical aspects of the system are two N20 micro gear motors that are attached to the bottom chassis board. The motors are connected to two rubber wheels that measure 42mm in diameter and 19mm in width.

To connect to the Raspberry Pi, we used SSH or Secure Shell which is a network communication protocol that enables two computers to communicate and share data. The operating system running on my partner and myself's computers is Windows, so to SSH to the Raspberry Pi, we used an application called Putty. Another method used to SSH to the pi was performed using Visual Studio Code. Visual Studio Code is a source-code editor that allows users to install extensions for software development purposes. We installed an extension called "Remote - SSH" that gave us the ability to SSH to the pi using Visual Studio Code.

Software Design



The first component of the software in our system is the AlphaBot2.py class. This class is used to set up the GPIO pins of the robot and contains functions to control the speed of the motors. The next software component is the curses program. Curses allows users to display text in a window screen and incorporate keyboard input to control a program. Our curses program has two main functions: enables the use of arrow keys to control the movement of the robot in manual mode, and checks for keyboard input from specific keys. (See code to the above.)

```
def manual(self):
    self.key = self.stdscr.getch()
    curses.noecho()

    if self.key == curses.KEY_UP:
        self.stdscr.addstr(7, 1, "                ")
        self.stdscr.addstr(7, 1, "Key Pressed: Up")
        self.Ab.forward()
        time.sleep(0.075)
        self.Ab.stop()
    if self.key == curses.KEY_LEFT:
        self.stdscr.addstr(7, 1, "                ")
        self.stdscr.addstr(7, 1, "Key Pressed: Left")
        self.Ab.left()
        time.sleep(0.075)
        self.Ab.stop()
    if self.key == curses.KEY_RIGHT:
        self.stdscr.addstr(7, 1, "                ")
        self.stdscr.addstr(7, 1, "Key Pressed: Right")
        self.Ab.right()
        time.sleep(0.075)
        self.Ab.stop()
    if self.key == curses.KEY_DOWN:
        self.stdscr.addstr(7, 1, "                ")
        self.stdscr.addstr(7, 1, "Key Pressed: Down")
        self.Ab.backward()
        time.sleep(0.075)
        self.Ab.stop()
```

```
def checking_keys(self):
    self.key = self.stdscr.getch()
    curses.noecho()

    if self.key == ord('m'):
        self.key = 'm'
        return self.key
    elif self.key == ord('a'):
        self.key = 'a'
        return self.key
    elif self.key == ord('q'):
        self.key = 'q'
        return self.key
    elif self.key == ord('t'):
        self.key = 't'
        return self.key
```

The next component of software in our system is the main program script, server.py. This script is where all software components are combined and work together to create a fully functioning robot. The program begins and its first task is to connect to the Redis server running on the pi. After connecting to the server, the program enters the main program. In this loop, the curses program is constantly checking for using input keys to switch between manual mode and autonomous mode. When the program is in manual mode, the user has the ability to control the robot using arrow keys or exit manual mode. When the program is in autonomous mode, there are three main functions working. First, the curses program is checking for user input to exit autonomous mode. Second, the program is running a collision avoidance function that uses the front infrared sensors to prevent collisions. The last function of autonomous mode is the PID loop which controls the bulk of the autonomous vision tracking. The value used for the PID loop is the arUco marker's position error from the center of the camera frame. Using this position value, we obtain kp, ki, and kd, which are the parameters for the PID loop. The summation of kp, ki, and kd is stored in the variable output, which controls the motor speeds. (See code to the right.) If output is less than 0, the arUco marker is detected on the left side of the camera frame, so we decrease the speed of the left motor to force the robot to turn left. If the output is greater than 0, the arUco marker is detected on the right side of the camera frame, so we decrease the speed of the right motor to force the robot to turn right.

```
if (output < 0):
    if flag == True:
        left = (maximum + output) / 4
        right = maximum / 4
        Curses.Ab.setMotor(-right, -left)
        flag = False
    else:
        left = (maximum + output)
        right = maximum
        Curses.Ab.setMotor(-right, -left)
        flag = False
else:  # output > 0
    if flag == True:
        left = maximum / 4
        right = (maximum - output) / 4
        Curses.Ab.setMotor(-right, -left)
        flag = False
    else:
        left = maximum
        right = maximum - output
        Curses.Ab.setMotor(-right, -left)
        flag = False
```

The last component of software incorporated into this system design is the python script that reads the images from the Redis server and displays an image on our local machine. To achieve this task, we SSH to the Raspberry Pi using Putty and enable X11 forwarding which allows for the graphical images to be displayed using Opencv. To show the images being sent through the Redis server, we run the python script called client.py. This script connects to the Redis server running on the Raspberry Pi and displays the incoming images. The purpose of this script is to display a smooth image and show if an arUco marker is being detected by the camera.

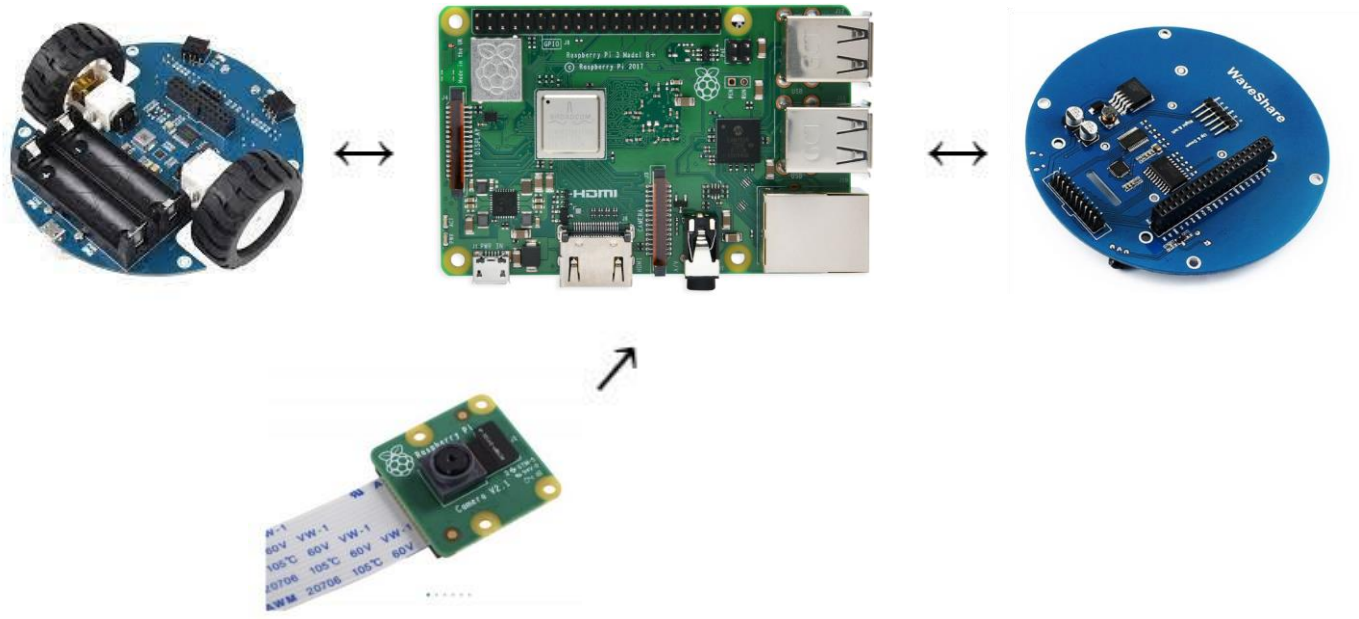
Control System Design

The manual control system incorporates the curses program, keyboard input, and functions from the AlphaBot2 class to control the robot; however, the control system for the autonomous vision tracking is more complex. The task is to autonomously track an arUco marker that is attached to the back of an remote control car. When switching into autonomous mode, the program is checking if an arUco marker is in the camera frame. If no arUco marker is detected, the motors

are programmed to stop. If an arUco marker is detected, the program begins vision tracking . Our control system is structured around the positioning of the detected arUco marker with respect to the camera frame. The fixed camera frame is 320 pixels in the x-axis direction, 240 pixels in the y-axis, and the top left coordinate of the frame is (0, 0). When detected, the program finds the four corners of the arUco marker and divides the x values by 4 to find the center position of the arUco marker. When the arUco marker is perfectly aligned in the center of the camera frame, the computed center position is 160. This makes sense because the half the camera frame length on the x-axis is 160. Our control system takes the returned center position value and subtracts 160 to compute the error. We subtract 160 from the position value because when the arUco marker is centered in the frame, we want the error to be 0. We then use the error value to compute kp, ki, and kd values for our PID loop. If the error value is negative, the arUco marker is positioned on the left side of the camera frame. When this occurs, the left motor speed is adjusted by a negative value. This results in the left wheel rotating at a slower speed than the right wheel forcing the robot to turn left. If the error value is positive, the arUco marker is positioned on the right side of the camera frame. When this occurs, the right motor speed is adjusted by a negative value. This results in the right wheel rotating at a slower speed than the left wheel forcing the robot to turn right. This technique solved the vision tracking problem; however, we used a different approach to implement collision avoidance.

The collision avoidance for our control system is based on camera vision and infrared photoelectric sensors on the front of the robot. The camera vision is used to slow down the speed of the wheels if the camera gets too close to the arUco marker. The infrared sensors are used to force the robot to a complete stop when an object is detected. When an arUco marker is detected, we find the difference between the bottom left corner y value and top left corner y value. For example, if the detected arUco marker is far away from the camera, the computed y difference will be small, and if the arUco marker is close to the camera, the computed y difference will be large. Then we check if the computed y difference is greater than or equal to 17% of the camera frame’s y-axis length. If this is true, we divide the current speed of both motors by 4. This translates to if the camera comes within a certain distance of the arUco marker, slow down. Lastly, if the infrared sensors are detected, the robot will come to a complete stop to prevent collision with the remote control car.

Hardware Design

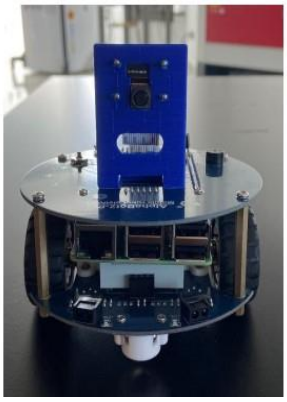
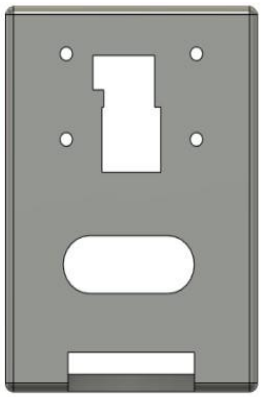


The electronic hardware consists of the Raspberry Pi, two chassis boards, and the camera. The camera communicates with the Raspberry Pi through the CSI (Camera Serial Interface) port which is where the Pi Camera module is attached using a 15-pin ribbon cable. The bottom chassis board and top chassis board are connected with a FC-20P 8cm cable. The cable is plugged into the AlphaBot2 control interface on each side of the chassis boards. The Raspberry Pi is connected to the top chassis board through GPIO pins, which is referred to as the Raspberry Pi interface. Each GPIO pin is associated with a specific piece of hardware on the Raspberry Pi. To communicate with the hardware on the robot, for example the infrared photoelectric sensors, we tell the software what GPIO pins the sensors are associated with and initialize them in the AlphaBot2.py class.

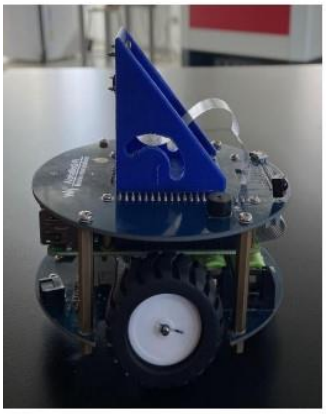
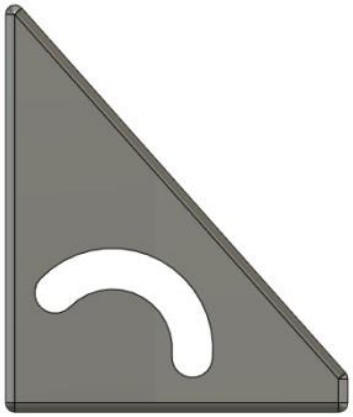
Camera Mount Design

The design we produced for the final camera mount was primarily focused around fixing the stability issues from the factory mount. The factory mount was flimsy and had many components holding it together. Our vision tracking system was purely dependent on the camera's image and the position of the arUco marker. An unstable camera would result in poor performance and inaccurate metadata needed for the PID loop. To fix this complication, we created one structurally sound design that was free of secondary components. This would prevent any arbitrary movement between the camera and the mount creating a stable structure. When designing the camera mount, three factors were considered, the top of the robot for mounting specifications, the length of the camera ribbon cable, and the camera's field of view. We

designed the bottom of the mount with a hole in the front to avoid soldered points allowing the mount to sit even on the robot. The rest of the design includes screw holes to attach the camera to the mount and the mount to the robot. The positioning of the camera was based on the length of the ribbon cable and where we felt the camera would have the best field of view for vision tracking.



Issues

The system overall performs well; however, there are some issues that need to be addressed. One issue is when the batteries lose charge, we need to remove the batteries from the robot to recharge them. The batteries take time to recharge and we found that the system does not perform well if the batteries are not fully charged. We would have to recharge the batteries after about five minutes of power usage. Another issue remaining in the system is the response time for manual driving mode. When switching to manual mode, the user has the option to control the robot using the arrow keys on the keyboard. There is a noticeable delay between the moment an arrow key is pressed and the resulting movement from the robot.

One aspect of the system that could be improved is the autonomous vision tracking. Our current PID loop has one internal conditional statement for collision avoidance. If the difference

between the top and bottom corners on the y-axis of the detected arUco marker is greater than or equal to
17% of the camera frame's y-axis length, the speed of the motors is divided by 4. An improvement to the autonomos vision tracking would be for the robot to increase speed if the detected arUco tag is a certain distance away from the camera. The solution to this problem would be very similar to the collision avoidance design. First, determine the maximum distance that the camera can detect the arUco marker and compute the difference between the y-axis corners. Next, add a conditional statement in the PID loop that checks if the computed y difference is near the maximum field of view distance. Finally, if this is true, increase the motor speed by a certain percentage.