Pi & Python Project Report: Roomba Home Defense System (R.H.D.S.)

Team name: Team South Hall

Jared Crow CSGD 2026, Matthew Killy ECMC 2026, Sam Kundt ECMC 2025

Letourneau University

COSC 3143-01: Pi and Python

Dr. Glyn Gowing

12/12/2024

Contents

Project Description

Intro

Our project is the Roomba Home Defense System - R.H.D.S. for short - which came from the idea of combining a Roomba, camera, and self-defense device. With the need for safety in our homes becoming more of a concern when the homeowner is absent, this project provides a solution to the problem of home security. The system we created can move autonomously for an extended period while searching for a face to target. When an intruding face is detected, the R.H.D.S. locks onto the intruder, tracks them, and takes a shot at them when in range before finally resuming its autonomous searching.

In a realistic situation, a more effective device for repelling the intruder such as pepper spray or Taser would be used, but for the sake of this project, a single shot NERF blaster was chosen for its simplicity and size.

Construction

Our solution to the problem contains several components, such as a Pi camera, a NERF N-Strike Jolt Blaster, two servos, the Raspberry Pi 5, the battery, and the tank chassis. Components were modeled in SolidWorks, which made it easier to design and 3D print brackets to hold everything in place. Not only do the brackets support all of the components, but they also allow the NERF blaster and camera to be rotated up or down by the servos. This is helpful to find the right height for the camera to see the intruder and be able to adjust the height to hit the intruder with the NERF blaster. Figure 1 shows the SolidWorks assembly. All the brackets were printed with an FDM 3D printer. Specifically, a Bambu Lab P1S printer was used.

The main brackets that attach to the chassis have two bearings for the NERF blaster to spin on. This makes it easy for the servo to rotate the NERF blaster brackets up and down. The camera is attached to the NERF blaster with another 3D printed bracket. It took a few iterations to get everything to fit and work together, but now it is a solid and robust system. To attach the brackets to the NERF blaster, I used two semi circles clamped around the circular grip of the NERF blaster. The reason for attaching it this way is to not have to drill and ruin the NERF blaster in anyway. The camera, servos, and the rest of the brackets all have mounting holes in them for M3 bolts.

The camera is connected by a ribbon cable directly to the Raspberry Pi, and is used to search for faces that the system can target. After finding out the right length needed for a cable, one challenge we encountered was when the first ribbon cable bought was mistakenly a monitor cable and did not work with a camera. Fortunately, the ribbon cables are cheap, and another one was quickly ordered. The connection to the servos is simple. The signal wire of the servo goes directly to the Raspberry Pi, while the GND and 5v go to the H-bridge. The H-bridge gets its power from the battery. At first, we tried to power the Raspberry Pi with the H-bridges 5v and GND pins but was not able to supply the current. Our solution was to use a buck converter to convert the battery voltage to 5v for the Raspberry Pi. It worked great. Even though the Raspberry Pi complained that it did not recognize the power supply, the servos and motors operated perfectly fine with this converter. The only downside is that the Pi would only last around 2-3 hours at a time on the battery. To conserve power, a Cana Kit charger was used to power the Pi while typing the code for extended durations of time before switching to the battery power source for running the program.

The program used to move the servos uses RPi.GPIO.PWM to create a PWM signal that would correspond to the position of the servo. The setup for it is very easy. First, you set the pin connected to the signal wire of the servo and the frequency of the PWM signal. Then you can set the starting PWM signal and change it whenever you want. One challenge that kept occurring was the servos were jittering when they were not moving. To fix this, the program only sends signals to the servos when they are supposed to move. This was done by changing the PWM signal to 0 after the servo has moved to the desired location.

In the future, we can add code to smoothly move the camera up and down as part of the scanning process. This will help make sure that the system is not missing an intruder. Right now, the servo's motions are pretty fast and rough, which made the system need to move in small increments in order to detect faces accurately. However, if we did some dynamics on the linkage system, we could slow it down and smooth it out in the future.

The Code (Imported Modules)

Three Python files are used for the functionality of the R.H.D.S. system. The main file CrowJ_PiProject_FaceDetection.py was created by Jared Crow, with some borrowed code from an open-source repository. It is the largest of the three files, runs the system when executed in the command line, and references the other two Python files during its runtime. The file SamK_PiProject_Chassis.py was programmed by Sam and contains simple methods for making the treads of the system move in different directions. The method left_turn() causes the system to turn left for a given number of seconds, the method right_turn() causes the system to turn right for a given number of seconds, etc. CrowJ_PiProject_FaceDetection.py calls the left and right turn methods extensively, but other movement options in Sam's code such as forward and reverse ended up being unused for reasons that are described in the following section.

The file MatthewK_PiProject_NERFShoot.py was created by Matthew, and contains a single, crucial method known as shootNERFgun(). This method is used to fire the system's NERF blaster given the pins for two servos and four angle values. The first servo adjusts the angle that the blaster is pointing at, and the second servo is used to pull the blaster's trigger. When called, method uses the first servo to point the change the angle of the blaster's firing position, then moves the second servo the pull the trigger, then resets the second servo to its default position, and finally resets the position of the blaster to default with the first servo. It should be mentioned that both the camera and blaster are attached to the came 3D-printed component, so when the blaster is adjusted, so too is the camera. Since the R.H.D.S. system operates from ground-level, its default position points both the camera and blaster upwards so the system can scan faces. When firing, the system points the blaster downwards to aim for a body shot instead of a head shot and then resets back to its default position to keep searching for faces. It should also be noted that one challenge due to size constraints was the system can't cock the NERF blaster on its own. However, that can still be done manually by the user before running the main program CrowJ_PiProject_FaceDetection.py.

In addition to the programs created by Sam and Matthew, the main program CrowJ_PiProject_FaceDetection.py also depends on other libraries to successfully search for faces to target. The OpenCV2 library gave the main program the ability to perform basic facial recognition by loading the 'haarcascade_frontalface_default.xml' cascade (Vpisarev). Also, OpenCV2 allows the program use the cv2.rectangle() method to dynamically draw a box around any face detected by the camera, and use cv2.imshow() to draw a window that displays what the system's camera sees. Speaking of which, the program also uses the Picamera2 module to be able to use the camera for face detection. Additional imports include numpy, threading, time, and RPi.GPIO.

Additionally, the program borrowed some open-source code created by GitHub user automaticdai to help with face tracking, which provided a very helpful starting point when creating the main program for this project. The program from the repository that code was borrowed from was face-detection.py, which can be found under the src/face-detection folder of the GitHub repository rpi-object-detection, which is linked at the end of this document in the References section (automaticdai). Any code in the main program that was borrowed from the open-source face-detection.py program is labeled as such in the Python file with comments.

The Code (Threading)

Essentially, the main program has multiple processes running at once through the use of threading, and always operates in one of two possible states: the patrolling state and the targeting state. Getting the main program's threads to work was challenging at first, but a solution was found through the use of global variables to make sure tasks only occur when they should. When the program starts, it is initially in the patrolling state, and two tasks are occurring: the main loop of the program, and a thread for the patrolling state. The main loop displays a window of what the Pi camera sees and continually scans for faces, drawing a blue box around any detected faces. Even though the main loop draws a window, seeing the window isn't necessary for the R.H.D.S. to function. The system can function autonomously without being plugged into a display device. The window is only there if the user wants to plug an HDMI into the Pi to see what the camera sees.

At the same time as the main loop, the patrolling thread causes the system's chassis to repeatedly cycle between gradually moving right for a handful of seconds, then left for a handful of seconds, and then back to moving right. The movement is handled by the methods called from Sam's program. Originally, we also wanted the system to move forward on its own in the patrolling state. However, doing so would have required more sensors to be added to the construction of the system, which would have likely made both the construction and code too ambitious for a single semester. Thanks to threading, these two tasks are able to occur simultaneously, allowing the system to search for faces and move in a looping cycle at the same time. These two tasks run continuously until the escape key is pressed by the user, which terminates all the system's tasks and ends the program. The only other time a task is halted is when the system changes to the tracking state.

As mentioned, the system searches for faces in the main loop while the patrolling state is running, but when a face is found, that face is added to a list of faces to potentially target. If there is only one face in the list of faces, that face is chosen to be targeted. Otherwise, if there is more than one face in the list of faces, the face with the largest box drawn around it is chosen to be targeted. On the display, the targeted face is outlined with a red box instead of a blue box. Once a face is chosen to be targeted, the system changes to the targeting state. While the targeting state is running, the patrolling state pauses and does nothing until the targeting state ends.

In the targeting state, a targeting thread is started. Instead of the system moving in a cycle, now the system is aligning itself to shoot the NERF blaster at its target. If the targeted face isn't in the center of the camera's view, the system moves right if the face is to the right of the center, or moves left if the targeted face is to the left of the center. On the display, the center range is depicted by a tall, yellow box. Like in the patrolling state, Sam's methods are used here to move the system's chassis left or right. Once the targeted face is within the center of the camera's view, the targeting thread calls the shootNERFgun() method from Matthew's code to fire the NERF blaster at its target. Once this occurs, the targeting thread terminates. Alternatively, targeting thread terminates if either the targeted face disappears from view for more than 1 second, or the targeting process takes 3 seconds or longer. The reason for this is because of a quirk with the chassis motors. The longer the motors are used by a thread, the slower they become, and ending the thread fixes the issue. Ending the thread this way doesn't cause issues because if the targeting thread ends while a face is still in the camera's sight, the targeting thread will instantly start up again, refreshing the speed of the motors. Once the targeting thread ends, so too does the targeting state ends, thus returning the patrolling thread to its normal function.

The Code (Variables and Organization)

Following the import statements at the top of the main program is the setup of all the GPIO pins needed for the servos and chassis motors, as well as the initialization of some global variables. Any variables created by Jared follow the naming convention of dataType_variableName. The variable b_debug mode equals False, but when set to True by a user, it auses debug statements to be printed to the command window while the program is running. The integer int_state controls what state the system is supposed to be in, and is referenced in the main loop, patrolling thread, and targeting thread. When int_state = 0, the R.H.D.S. is in the patrolling state, and if int_state = 1, the system is in the tracking state. Other globals include a tuple for the largest face, a float for the initial time the program starts, and a boolean that equals True until the patrolling thread starts for the first time.

After the variables and GPIO setup are the methods. The first method visualize_fps() is from the borrowed face-detection.py code, and is used to display the current FPS of the program's display. After that are the methods thread_patroling and thread_tracking, which were both coded by Jared and contain the functionality of the patrolling thread and tracking thread respectively. After referencing two global variables, thread_patroling contains a while loop that repeats continuously until the program ends. More globals, including int_state are referenced inside of the loop, causing them to update every iteration. Inside of the loop are references to Sam's methods that cause the system to autonomously move left or right, which are only called if int_state = 0. The system patrols right by default, but after 100 iterations of this loop, it changes direction. Also, the Boolean b_previouslyTracking also helps the system to maintain its last direction if the state switches from tracking back to patrolling.

Next in the method thread_tracking, global references are made to the pins used to fire the NERF blaster, in addition to int_state, and some local variable declarations. Unlike in thread_patrolling, int_state only needs to be referenced once at the beginning. Then there is a while loop that terminates if either of the previously mentioned time conditions are met, or breaks if the targeted face is in the center range. The while loop contains the if statements that allow the system to dynamically move left, right, or shoot, in addition to an if and else that controls the value of face_prevFace. If a face ever disappears from view, the red box for the face remains perfectly still as opposed to maintaining subtle movements. So, face_prevFace is used to determine if this is occuring. If the face is gone, that causes the red box to pe perfecly still, which causes the first timer to count down. The second timer counts on its own.

After the methods, comes some more borrowed code, mixed with added code, which both declare variables needed for the main loop. The distinction between borrowed variables and variables by Jared can be made with their different naming conventions. Then there is the main loop, which contains the last of the borrowed code at the beginning and end, with Jared's original code in the middle. Three lines of borrowed code at the beginning of the loop initialized variables that help the camera process any images in grayscale. Seven lines at the end of the loop display the window every iteration, and contains an if statement that ends the loop when the escape key is pressed. The original code appends a detected face to l_targets, and then finds the largest face in l_targets in an if statement. Once the largest face is found, a tracking thread is started only if b_notTracking is True. This was done to make sure only a maximum of one targeting thread is running at a time. If int_state = 1, b_notTracking = False. Else, b_notTracking = True.

Finally, at the end of the program are the lines of code for cleanup. The Pi camera object is closed, All GPIO pins are set to False, GPIO.cleanup is called, and cv2.destroyAllWindows() closes the program's display.

Motors

The motors are driven by a L298N dual H-Bridge circuit. The dual H-Bridge circuit takes HIGH or LOW inputs from the Raspberry Pi or PWM inputs. The H-Bridge then converts these inputs to the higher battery input voltage. We opted to forgo the option of PWM for motor control. And only drive the motors at max speed, this choice was made to simplify the coding process. Methods are declared for driving forward, turning and driving backward. This makes the main loop easier and cleaner. The code used included GPIO.cleanup function at the end of each method, which conflicted with the other methods being executed after movement.

Electronics

One of the biggest challenges of the project was figuring out a way to supply enough power to both the Pi 5, chassis motors, and servos. The system is powered by a 7.4V 2S lithium – ion battery. The 2S battery was chosen because of its compact form factor, relatively large capacity. Two standard geared DC motors were chosen to drive the chassis. To be able to supply enough current to the motors, an L298N dual H-Bridge circuit was chosen. The dual H-Bridge circuit can supply the components with the required power. The circuit was kept fairly small, to adapt to the small form factor of the device.

Battery

The battery is a lithium – ion battery with 2 cells. Providing 8.4V when fully charged. As mentioned above, this battery was chosen for the high capacity for its size. This is important, as the system is supposed to be able to move autonomously for an extended period. This comes at the tradeoff of having a slower discharge rate compared to equivalent lithium – polymer batteries. However, the low discharge rate is not a problem for our system as we are not powering equipment requiring a lot of current such as stepper or brushless motors.

Servos

Servos were the main challenge in our system, by far requiring the most amount of power to pull the trigger. The servos are used to aim the turret and fire the NERF gun, and the position of the servos is controlled using PWM signals from the Pi. However, the current required by the servo to engage the trigger is very high compared to the rest of the system, this results in a much shorter functional battery life, in which the robot is able to both fire and patrol. This can be fixed by either using a battery with a higher voltage, or increasing the capacity for the battery.

Conclusion and Future Improvements

The project is regarded to have been successful. An autonomous vehicle was built that could patrol left and right, track a target, and fire a shot. This serves as a working proof of concept for how to build such a system.

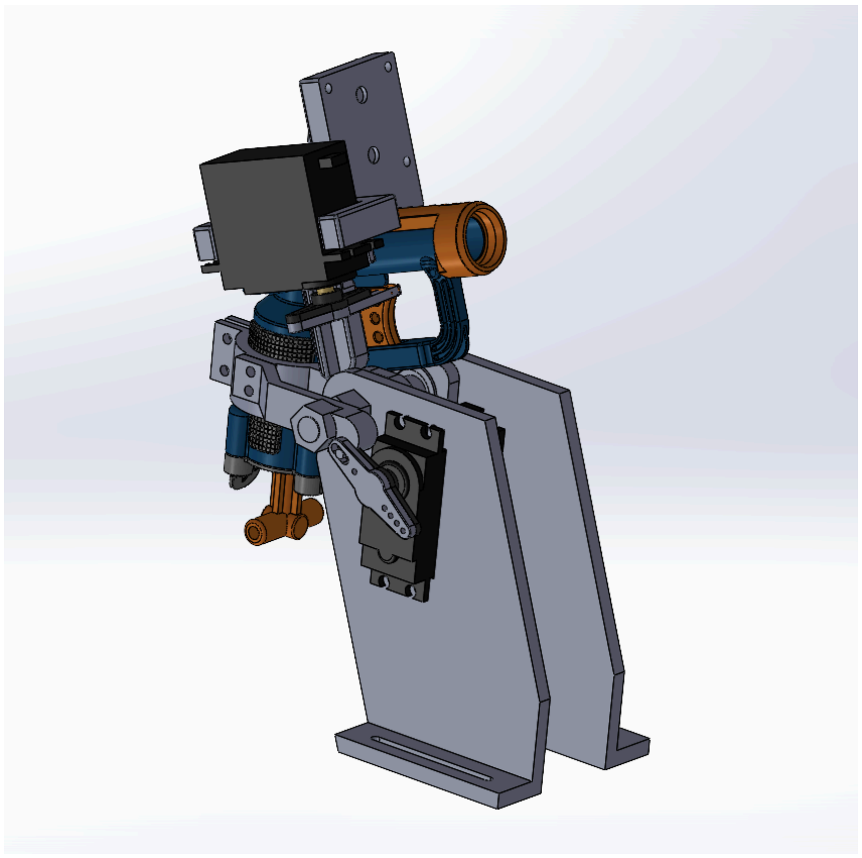
In the future, if improvements were to be made, a web application or a smartphone app could be developed to see a live feed of the camera, and even override directional control manually. The addition of positioning sensors, such as ultrasound, infrared, LIDAR, and GPS would be extremely useful for creating a map of a house so the R.H.D.S. can patrol around a room or building like a Roomba. In addition, these sensors would help gauge distance to a target and help moving into firing range. Finding a way for the system to dock itself when needing to charge would also greatly improve the system's viability for a longer trip or vacation. Still providing safety for longer than a day or two. Finally, OpenCV should be trained on some more sets of data, not just allowing for face tracking, but tracking human bodies in general, especially when not facing the machine. Safety features should also be implemented in the system, such as shutting down when the owners enter the house.

Not to mention, we all gained valuable new experience from this project. Jared learned how to work with OpenCV, a camera module, and face tracking library for the first time, in addition to becoming more comfortable with threading and modules in Python. Matthew learned to work with PWM to control servos and had to figure out how to power the system, in addition to learning about face tracking from Jared. Finally, Sam learned how to use H-Bridges for motor control, how to manage the power of the chassis, and how manage the confined space of components when constructing the system.

Back Matter

Figure 1

SolidWorks Assembly



Parts List

2Pcs MG995 55G Micro Servo Motor Metal Geared Motor Kit for RC Car Robot Helicopter

Cost:	$13.99
Brand:	Deegoo-FPV
Amazon listing:	https://a.co/d/3SOLdgq

2WD Tank Robot Chassis Platform with 2PCS DC Motors & Plastic Track

Cost:	$20.99
Brand:	Smaringrobot
Amazon listing:	https://a.co/d/gzPkbQN

7.4V Li-ion Battery 2000mAh 2S with JST Plug Rechargeable High Capacity RC Battery with 2 USB Charger Cable

Cost:	$24.48
Brand:	PCEONAMP
Amazon listing:	https://a.co/d/0EsX3MA

Arducam Lens Board OV5647 Sensor for Raspberry Pi Camera, Adjustable and Interchangeable Lens M12 Module

Cost:	$18.99
Brand:	Arducam
Amazon listing:	https://a.co/d/a1L4c8M

CSI FPC Flexible Cable For Raspberry Pi 5, 22Pin To 15Pin, 0.5 M Length, Suitable For CSI Camera Modules

Cost:	$7.97
Brand:	Wonrabai
Amazon listing:	https://a.co/d/h7Dm9E5

NERF N-Strike Jolt Blaster (blue)

Cost:	$14.49
Brand:	NERF
Amazon listing:	https://a.co/d/1Do93lz

Third-Party Modules

NumPy

How to install: https://numpy.org/install/

OpenCV cv2

How to install: https://opencv.org/get-started/

Picamera2

How to install: https://datasheets.raspberrypi.com/camera/picamera2-manual.pdf

RPi.GPIO

How to install: https://www.raspberrypi-spy.co.uk/2012/05/install-rpi-gpio-python-library/

References

automaticdai. (2019). *Raspberry Pi Real-Time Object Detection and Tracking*. GitHub. https://github.com/automaticdai/rpi-object-detection?

Vpisarev. (2013). *data/haarcascades/haarcascade_frontalface_default.xml*. GitHub. https://github.com/opencv/opencv/blob/4.x/data/haarcascades/haarcascade_frontalface_default.xml