Project Final Outline

Dmitri Demler & Leo Megliola
Physics 124

Motivation & Overview

There is a recent shift from use of notebooks to tablets for taking notes and writing problem sets. Tablets offer many benefits to students such as organized storage and a connection to the internet. However, there are also some drawbacks. One such drawback is glare caused by overhead lights in the screen of a tablet that is placed flat on a surface. This is of particular relevance in well-lit scapes such as libraries. Such spaces were designed to help reading books or writing in notebooks, but did not consider glare in horizontal screens. The result: library tables that are incompatible for students using both tablets and conventional paper notebooks simultaneously. The goal: only illuminate conventional notebooks, while leaving tablets alone.

To achieve this goal, we want to design and produce a robotic lamp that uses computer vision to selectively illuminate paper notebooks. This lamp is not just static; it actively searches tables for objects to illuminate. When such an object is found, the lamp will turn on and wait for the object to be removed before continuing to search the table. When the lamp detects a tablet, it understands the need to prevent glare and remains off, thereby avoiding unnecessary reflections on the screen.

Our hope is to improve the library lighting for everyone by removing the need to manually turn on their lamp.

Functional Definition

The primary objective of the light is to pinpoint and brighten conventional paper notebooks, ensuring that it skillfully maneuvers around obstacles and notably refrains from illuminating tablets. To meet this requirement, the lamp must exhibit several core functionalities:

1. **Mobility:** Positioned atop a robot chassis with wheels, the light possesses the ability to traverse the length of a table or desk. It maintains a step-like motion where it moves a small step then takes pictures while ensuring it remains atop the table or desk and does not fall off during its operation.
2. **Collecting Images:** The light will be equipped with a camera, allowing it to take snapshots of the area that would be illuminated if switched on. In order to distinguish between notebooks and other objects, the light will first need to collect random images from sample tables. We can make different datasets in the case we run into issues with the model seeing the notebooks. We can make one dataset with spiral notebooks.
3. **Training:** The sample images will be labeled to serve as a training set for a neural network. The training could occur offline, using a more powerful computer system than the onboard microcontroller. However, we will ensure that the network is small enough to run on the raspberry pi once trained.
4. **Recognizing Images:** Once trained, the network will then be deployed to recognize notebooks. The resulting set of network weights and biases should allow for real-time image recognition, since a trained network can perform feed-forward calculations quickly.
5. **Avoiding Obstacles**. The robotic foundation of the lamp, which grants it mobility, is enhanced with dual IR proximity sensors located at each end. With one sensor oriented forward and the other downwards, this configuration safeguards the lamp, ensuring it remains on the table and steers clear from potential obstructions.

Some features we include as options in the case that time permits:

1. **Camera Arm:** At its most basic, the camera arm remains static.
2. **Onboard Processor:** In its foundational design, it is adept at controlling a camera, DC motors, and proximity sensors while executing image recognition via a pre-trained neural network. Expanding on this, the advanced design envisages an onboard processor that undergoes native training, refining its performance during active usage.
3. **Camera Functionality:** At the minimum, the camera captures images at regular intervals, providing time-independent inputs to the model. With this approach we can use the simpler convolutional neural nets for notebook classification. In a more sophisticated approach, it offers continuous image capture, necessitating time-dependent data processing through models like RNNs, GNNs, or Vision Transformers. However this requires analysis on the capabilities of a raspberry pi with such an intensive network.
4. **Robot Chassis:** The basic model utilizes a readily available chassis designed for the Raspberry Pi. Contrarily, the enhanced design boasts a custom 3D-printed chassis tailored explicitly for the light's unique requirements. The custom chassis can allow for a better located center of mass and higher moment of inertia to prevent the lamp from falling.

The following table summarizes these ideas:

Sensors

Camera Sensor

In our design, the primary input device utilized is a camera. Its purpose is to capture snapshots of the potential area that the onboard light would illuminate. We opted for the Raspberry Pi Camera Module V2, a camera that boasts an 8 Megapixel resolution and is capable of recording at 1080p. This camera was preferred over a USB-connected alternative for several reasons:

- **Integration**: The Raspberry Pi Camera Module seamlessly integrates with Raspberry Pi setups. This tight integration ensures a smoother data flow and more direct control, enabling better real-time responses.
- **Compactness**: The Raspberry Pi Camera Module's compact design means it won't add unnecessary bulk to our system
- **Power Efficiency:** The Raspberry Pi Camera Module consumes less power compared to USB alternatives, which ensures longer operation times and less strain on the power supply.

Here is some more information about the camera itself:

- This sensor is actually a Sony IMX219 sensor which captures images with a high level of detail, crucial for our application where discerning between different objects on a table is vital.
- The sensor has a relatively wide field of view allowing it to capture a bigger portion of the table in a single shot. This is not necessarily an advantage because the lamp might detect the notebook at the wrong angle and it will stop early.
- The camera is able to adjust to different lighting conditions which helps with its adaptability in different environments.
- Images captured by the camera are directly fed into the Raspberry Pi system. This direct connection minimizes lag and ensures that the AI model receives image data promptly for swift processing and decision-making.
- One potential problem is that the recommended minimum distance for the camera is 50cm which is a little tall for the light mechanism and might compromise the balance and stability of the device. We suspect this will not be an issue because we will likely lower the resolution of images anyway, so the image quality is not critical. Interestingly, as long as the images are self-consistent and sufficiently detailed, the network should be able to train regardless of the images looking out of focus to a person.

The Camera module uses a ribbon cable to attach to the raspberry pi. We will need to get longer ribbon cables so that they can go the length of the lamp. To connect the camera to the raspberry pi we plug the cable to the Camera Serial Interface port on the raspberry pi.

Proximity sensors

We plan to use the infrared Sharp GP2Y0D805Z0F digital distance sensor. We will use it to ensure that the lamp system can operate autonomously, without running into obstacles or inadvertently falling off the table's edge. When the device senses the end of the table (for the downward facing sensors) or an obstacle (for the side facing sensors) the lamp needs to stop and reverse its direction to not fall or collide.

Note: We decided to use this model because it's available already in the lab. However, if this is no longer the case and other proximity sensors are available we can use them instead.

Specifications:

- The sensor detects obstacles in a range of 0.5cm to 5cm. This tight range ensures that the lamp can make rapid decisions without being halted frequently by distant objects that aren't immediate concerns.
- The sensor offers a direct digital output, allowing for a simplified interfacing process with our system's Raspberry Pi, eliminating the need for analog-to-digital conversion.
- By emitting and receiving infrared light, the sensor can identify the proximity of objects or the absence of a surface beneath the lamp. This mechanism is less prone to errors from ambient light or other external disturbances.

Regardless of which sensors we employ, we will need to experiment with position and orientation. For the downward-looking sensors, we need to extend the mounting points to cover the possibility that the robot approaches the edge of a table at a very shallow angle. The sensor needs to extend far enough away from the corner of the robot to look down and report a change in distance. That event must happen soon enough to allow time to respond, using an interrupt, that will immediately turn the motor off. If we struggle to get adequate feedback, we could increase the number of downward looking sensors from two to four. Given the shape of most tables and the initial orientation of the robot, that change will probably not be necessary.

For the forward looking sensors, the priority is to ensure a clear path for the robot’s wheels. Some small obstacles will go undetected, so again we need to experiment to determine what configuration provides a reasonable chance of good performance. We could try orienting the forward looking sensors at an angle outward from the edge of the robot (front and back), but depending on how the sensors operate, that might cause the signal emitted to scatter rather than reflect. Again, there is a chance that we might need to add additional sensors.

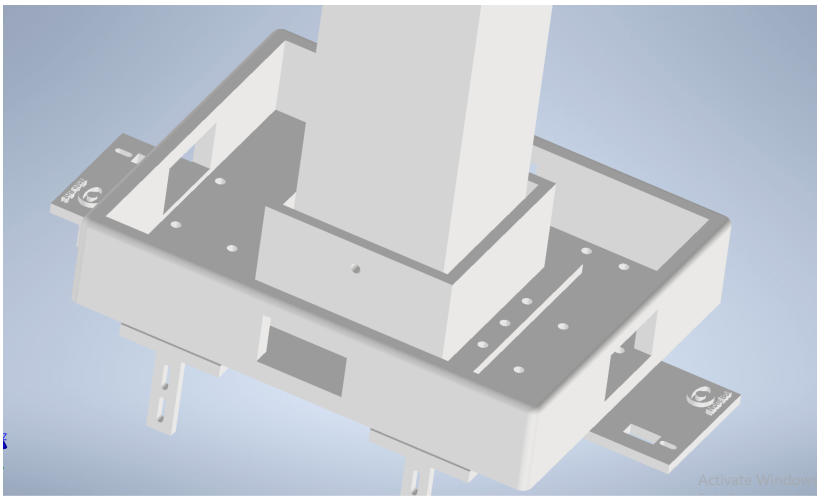
Mechanical Considerations

1. The robot is built on a basic 3-d printed chassis that allows it to drive around on a flat surface.
2. The forward-facing sensor must be mounted and oriented in a way to minimize interference, while detecting an object as small as a pen or pencil.

3. The downward-facing sensor must be mounted in a way that ensures detection of edges, even when approached at a shallow angle.
4. We need to make sure that the center of mass is at the wheels of the device. Otherwise, the device could tilt and fall. Furthermore we need to ensure that it is stable enough to not fall when it's accelerating.

Here is the prototype version of the device modeled using Autodesk Inventor. The base that holds the wheels, raspberry pi, and battery will be a finger jointed simple box that will be made from laser cut wood. In the prototype model it is shown as a simple box for now. The main “tower” of the robot will be a hollow box as well but the top part will be 3D printed due to the complexity. The tower has two parallel slider pieces that will go against the top horizontal bar to ensure the robot is balanced. The tower will be connected with screws and superglue on the bottom to the base box. The robot on the top has 3 circular slides to allow the lamp to bend without sacrificing stability and still allowing it to stay fixed. This will allow us to find the perfect angle for the camera. The horizontal part of the lamp that holds the counterweight and camera+light systems will be 3D printed since it is a small enough size and contains the sliders. The counter weight will ensure that the robot will not tilt under the weight of the camera system. We will most likely use a wooden block with dimensions depending on the weight of the other side. The camera+light part is a placeholder for the actual light that we will use and the camera. We will still use a circular slider to allow rotation. Lastly, the sensors will be connected to sides of the base looking outside and we will use wheels that we can find already in the classroom.

[Full version of the robot]



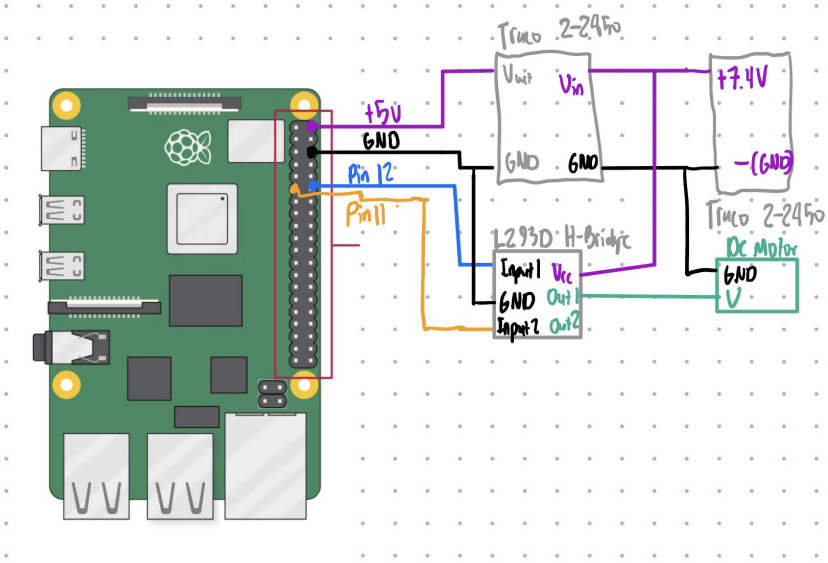
[Base of Tower]

Here is a photo of the working prototype:

Electrical Considerations

1. To supply power, we use a standard external power delivery to the Raspberry Pi. The entire device runs at 5V. The H-bridge decouples high current elements (DC motors) from logic components.

2. In order to operate the DC motor in both directions, it is necessary to reverse the polarity of the current applied across the motor. To accomplish this, we will use an H-bridge wired to two GPIO pins. That will permit directional control in software.

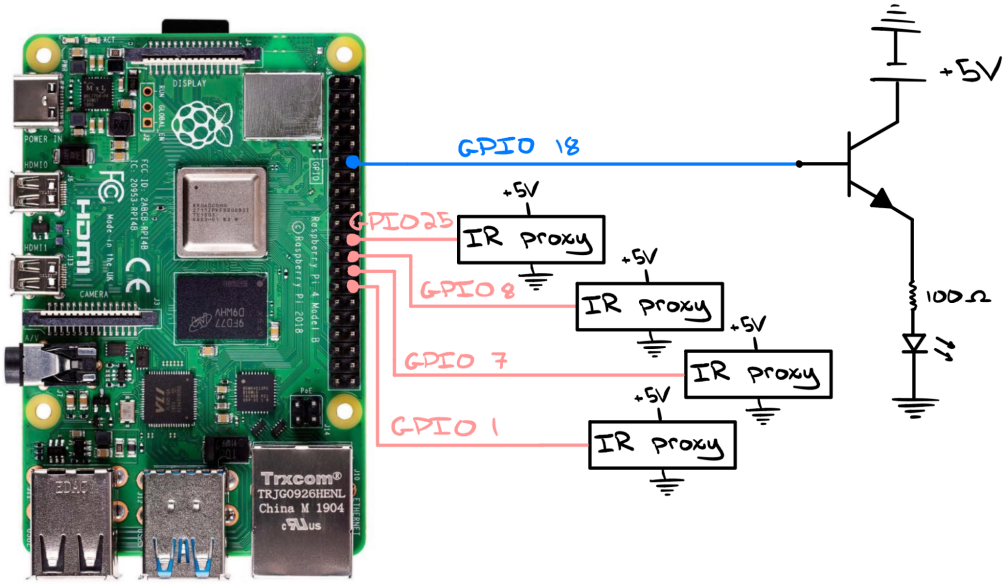


[Diagram showing wiring for points 1&2]

3. To modulate the speed of the DC motors, we use a high duty cycle and run the motors for short bursts, moving the robot forward incrementally.

4. The LED lamp may draw more current than is available from a Raspberry Pi GPIO pin; it is driven from the other side of the H-bridge to allow for greater current.

5. To monitor the proximity sensors, direct connection to GPIO pins is adequate.



[Diagram showing wiring for points 4&5]

Interfaces

1. The Raspberry Pi can communicate with proximity sensors using GPIO pins.
2. To control DC motors, the Raspberry Pi connects to an H-bridge that subsequently controls the motor.
3. To acquire images, the Raspberry Pi connects to a camera using a standard ribbon cable.
4. For software interfaces, see below in “Programming”.

Software

Programming

Our robot will employ **Raspbian**, a Unix-based operating system, tailored for Raspberry Pi devices. Raspbian has many handy libraries for image acquisition, neural net training, and, of course, the fundamental hardware control. This ensures our robot can run smoothly, efficiently, and leverages the best tools available for its operations.

We will primarily use the programming language **Python** due to its prominence in machine learning, simplicity, and integration with the raspberry pi. Python can access the Raspberry Pi’s GPIO pins through gpiozero. We will make python scripts that will govern how our robot interacts with sensors and DC motors. The primary objective here is to ensure that the robot can traverse its environment, be it a desk or a table, without colliding with objects or risking a fall from edges. This should be fairly straightforward, as we are limited to 1D motion.

To methodically explore a flat surface, our robot will deploy a **search algorithm**. We will employ the simplest possible search procedure: proceed in a direction until an obstacle is encountered, then reverse direction. This will sweep the maximum table area back and forth. The sensors will be digital interrupts, giving them the highest priority, ensuring the robot does not hit an object or fall off the edge of a table.

We will use OpenCV, an open-source library for computer vision for capturing still images of the desk. These images serve a dual purpose: creating a robust training dataset and aiding in real-time notebook detection during operations. A crucial part of this process is the hyperparameter that determines the rate at which images are captured and fed to the Raspberry Pi. By optimizing this parameter, we can strike a balance between system efficiency and accuracy in object recognition. Essentially, we will tune the rate at which images are acquired against the distance traveled between images. When done properly, this will ensure smooth, fast, and accurate operation.

To train the neural network we will use TensorFlow, a popular machine learning library. We will train remotely, on a desktop computer, GPU, or in the cloud. When the model is trained, we then convert it to Tensorflow Lite which is a lightweight option designed for mobile and edge devices such as a

raspberry pi. We can then run the model using opencv and thus deploy on the robot. This prevents processing limitations from getting in the way of finding ideal model parameters.

We will now go into more specifics about the machine learning aspect of the project. The baseline dataset as previously mentioned will be sets of images either with the notebook or without and labeled as such. Images will be labeled by hand from a set of snapshots captured by the raspberry pi. There will be many obscure considerations, such as how to label images where part of a notebook is visible. This will require substantial experimentation.

We will use dense layers with convolutional layers in a fairly standard way for computer vision. To keep computation fast we will use ReLu activation functions except for the output layer where we will use sigmoid. More specifications about the network will be seen in the following sections.

We noticed that most notebooks have blue lines and we can take advantage of this. We will try out options to make masks that show the blue line data individually and add them to the input. This could help train the network to notice the notebook lines as characteristic of notebooks. The reason we want to use a GPU is not due to the complexity of a single training, we want to try out various widths, heights, activation functions, and input data. To do this we need to train multiple networks in parallel and overnight to ensure the highest chance of success.

Tricky Parts

Machine learning is a potential bottleneck, so we need to consider strategies to increase our chance of success. Firstly we need to ensure that the model is reliable. The model needs to demonstrate a reliable capability to detect notebooks. It's not just about detecting any notebook, but specifically when the environmental conditions, such as the same notebook or table, remain unchanged. This will serve as our baseline performance. With this foundation established, we will escalate the complexity, training our model to recognize a broader range of notebooks under varying conditions.

Regardless of the level of complexity, the data assembly and preprocessing steps are also quite tricky; crafting an extensive and appropriately labeled dataset is undoubtedly challenging. However, the true test lies in ensuring the data facilitates accurate training. To bolster this effort, image preprocessing becomes crucial. Techniques such as grayscaling and resolution reduction can dramatically change the computational intensity of training and increase the accuracy of results. We also need to ensure that the data is diverse enough to prevent overfitting. To combat this we plan on running L2 (or L1) normalization, weight decay, and using separate validation and testing datasets.

We also need to figure out how to account for partial notebook views. It's improbable that the camera will always capture a complete view of the notebook. There will be instances where only sections of the notebook are visible. We need a strategy to label these images appropriately. We suspect that partial images should be labeled as if they contain a notebook, so as to not confuse the network into ignoring features of notebooks that are visible. However, this and all assumptions about the inner workings of neural nets often do not correspond to intuition. It must be tested.

Another challenge is to make sure the model is small enough for the raspberry pi to run fast enough. If the model takes too long to run then the lamp won't realize what it is seeing is a notebook until it has already passed it. We don't anticipate a problem as we will be running in feed-forward mode, with no backpropagation or iterative training steps.

Code and Code analysis

We developed a pipeline of processes to acquire, preprocess, and train from images. This has no natural place within the rest of this proposal document, so we made this section because this was a rather major component of the project.

Acquisition

We acquired images using the burst picture feature on an iPhone. That allowed us to capture images far more quickly than using the Raspberry Pi. However, it also required that the neural network succeed in generalizing its recognition of images, since the images processed in real time came from a fundamentally different source than the training set. We believe that requirement has a positive overall effect because generalization (instead of over-fitting) is the primary goal of training.

Preprocessing

To get the images preprocessed we automate the following steps in linux. Images on the iphone were normally in **.HEIC** file type and so we first changed them to jpg filetype, and then reduced the dimensions. To do this, make sure you have the files in a different folder, then run:

```
for file in *.heic; do sips -s format jpeg "$file" --out
"${file%.heic}.jpg"; done
rm -rf *.heic
for file in *.JPG; do sips -Z 800 "$file" --out
"../Signal_validation_2/${file}"; done
```

This code first changes the file type to jpg and then deletes the old filetype images and changes their dimensions to 800x600 and then moves them to the folder. The previous full dimension images are still in the original folder.

Note: For each code extensive code comments can be seen in the file itself.

After this is run the python file called **datamaker_finalreport.py** is run to preprocess the images to eventual training data. It changes the images to black and white and makes copies that are flipped horizontally. It also further decreases the dimensions of the images.

Training

Following this, the neural network training can begin. One of the jupyter notebooks is included in the code folder and is called **final_report_neural_network.ipynb** which sets up the data to work with

tensorflow, sets up the parameters of the neural network and then trains it and compares it with a separate validation dataset. For a description of the network architecture, see the source code.

Since the data is still relatively small compared to other popular datasets, we try to overcome overfitting in the following ways. We implement layer dropout, where 50% of the weights in each epoch of training are ignored so that each specific weight doesn't just memorize each “pixel” of the training data. Furthermore we use weight decay as well, where the weights loss function has an added component to have better regularization. This helps discourage the model from overfitting.

We chose some other model specifications as well from experience with other computer vision tasks. We use the Adam gradient-based optimizer function which uses momentum and Root Mean Square Propagation. We tried a mix of activation functions but decided to use the simple ReLU function since it is computationally the most efficient and does not compromise the accuracy as much. The final model had the following layer architecture:

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 238, 318, 32)      320

 max_pooling2d_3 (MaxPoolin  (None, 119, 159, 32)      0
 g2D)

 conv2d_4 (Conv2D)           (None, 117, 157, 64)      18496

 max_pooling2d_4 (MaxPoolin  (None, 58, 78, 64)        0
 g2D)

 conv2d_5 (Conv2D)           (None, 56, 76, 128)       73856

 max_pooling2d_5 (MaxPoolin  (None, 28, 38, 128)       0
 g2D)

 flatten_1 (Flatten)         (None, 136192)            0

 dense_6 (Dense)             (None, 128)               17432704

 dropout_5 (Dropout)         (None, 128)               0

...
Total params: 17591553 (67.11 MB)
Trainable params: 17591553 (67.11 MB)
Non-trainable params: 0 (0.00 Byte)
```

We have 3 convolutional layers followed by 5 fully connected layers with the width of 128. This means that there are 17 million trainable parameters which means a model around 70 MB. This means that this model is still small enough to run on the raspberry pi while still having enough parameters to make a good model.

Robot Control

Finally we have attached the control software that runs the mobility, collision detection, and image processes on the robot. This is called **final_notebot.py**. For details, see comments in the code. We originally intended to use multi-threading, but did not have time to write and test additional control software. See below in lessons learned for details.

The code to run this needs to control the IR sensors, the DC motors, and trained tensorflowLite model. Once the initialization is complete the lamp starts the operational phase. The pi takes a motion "step" phase where it moves for a set time (0.2 seconds) all the while checking for obstacles or table edges at the beginning, middle, and end of the motion. Then while being in a "stop" state it takes a picture and processes it through the network. We have a probability matrix that takes the last 3 results of the picture processing and if the sum of the probability (that the network thinks the pictures are a notebook) and if its larger than 0.4*3=1.2 and then the pi stops, and turns on the light.

Development Path

The step most likely to fail is training the neural network, while the step that takes the most time is the design and construction of the mechanical robot. Therefore, the development path should immediately allow testing of the neural network, while also allowing these efforts to proceed in parallel. First thing we will do is to make a makeshift arm that will be at the approximate height that the lamp will be at so we can start making the training dataset as early as possible.

First steps:

1. Create a simple 3d model of the critical dimensions of the robot primarily to get an idea about camera placement.
2. Position the camera at the height and angle it will have in production. Using the camera and OpenCV to collect snapshots, generate the labeled image set necessary for training our neural network.

We will then be in a position to work in parallel, avoiding bottlenecks from wait times for 3d printing or ordering parts:

Track 1: AI Model

1. Collect labeled data sets by taking images of tablets and notebooks in the context of a desk using the camera that will wind up on the robot.
2. Design and train a neural network on these datasets.
3. Test for accuracy by presenting the trained network with new images.
4. Repeat from step 2 if necessary.

Track 2: Mechanical Design and Construction

1. Finish designing and modeling the robot chassis.
2. Fabricate (though primarily 3D printing) the chassis and populate it with electronics such as motors, raspberry pi, light, camera, etc.
3. Perform basic control tests (turn motors on and off, take snapshots, read values from proximity sensors).

Once the mechanical robot is complete and the AI model is trained:

1. Develop and test a basic rover algorithm, allowing the robot to explore a tabletop.
2. Add image capture, stopping the robot and illuminating the light when a notebook is found.

Testing

Testing of the main subsystems is independent, so can be done in a parallel fashion:

Linear Upright Chassis (LUC).

1. Motor control. Confirm that the Raspberry Pi can control the DC motor, in both directions, by wiring the H-bridge and running a simple Python test script.
2. Torque test. Confirm that the motor can produce sufficient torque to move the chassis at constant, slow speed while substantially geared down.
3. Staying upright. To test the robot chassis' ability to maintain linear motion while staying upright, we will perform test runs on a tabletop, including:
    3.1. running toward the edge at twice the estimated final speed.
    3.2. approaching the edge at a variety of angles, including very shallow angles.
    3.3. running into combinations of obstacles, including narrow objects.
    3.4. examining potential sources of interference for the sensors.
4. Alignment. Go straight(ish). Align and tune the wheels to cause the robot to go in a reasonably straight line. Check that changing direction (repeatedly) does not have a substantial effect on heading.

Notebook-Sensing Neural Network (NoSe).

1. Training. While training the network, we will test a variety of:
    1.1. Preprocessing pipelines. We will examine the effect of image preprocessing (resolution reduction, grayscaling, color masks) to determine what combination produces good accuracy with reasonable computational overhead.
    1.2. Network configurations. We will examine the effect of various network configurations (node counts, layers, activation functions). We will perform tests to find a workable model for our neural network.
2. Software Deployment. We will confirm that we are able to deploy the trained network onto the Raspberry Pi (which has limited memory and processing power).
3. Hardware Deployment. We will use the camera in a realistic setting to capture images, then process those images on the Raspberry Pi, to test for accuracy.

Integration Test.

1. Interference. Using a fully assembled robot, we will test that the onboard systems do not interfere with each other.

2. Final Testing. We will deploy the robot on a table at the library and test its effectiveness.

Safety

A potential concern to be addressed here is if we are taking images in a public place, we should be careful not to depict people without their permission.

Parts and Reusability

Hardware Stack			**Software Stack**
Component	On hand?	Reusable?	
1 x Raspberry Pi 4 1 x H-bridge 4 x DC motor 4 x GP2Y0D805Z0F 1 x RPi camera 4 x wheels 1 x robot chassis 1 x set of 4M fasteners 1 x ribbon cable	yes no yes yes yes yes no yes no	no no no yes yes no no no no	Raspbian OS (based on Debian Linux) Python OpenCV Raspberry Pi Python libraries (gpiozero, etc) TensorFlow / Keras

[Everything marked not reusable was purchased or constructed by us during the project.]

Shopping list

1. Camera (amazon), $24.71

Total: $24.71

Lessons Learned

Everything we built broke once (or more than once) before working properly. Each time this happened was a learning opportunity. To stay organized, we will discuss some of the highlights in two sections. First, hardware challenges, then after we will discuss software challenges.

Raspberry Pi Camera

The first issue we encountered was connecting the camera to the Raspberry Pi. When we managed to display the video feed on the screen we noticed that the field of view was far more narrow than we expected, almost as if the camera was at two times zoom. We referred to the documentation and found out that the resolution of the camera directly controls the FOV. By default it is not at full resolution, and so the video is indeed zoomed in. This seems innocuous, but will turn out to be important when we want to process the images on the pi before feeding them to the neural

network—the camera needs to see enough of the table to distinguish notebooks, without the tower being too tall.

Designing parts that work in the real world

The next thing we learned is that it can be tricky to match dimensions in CAD software to that of parts that already exist. This is especially apparent for small parts with holes for small fasteners. This effect is compounded with the imprecision of 3D printers. As we came to understand, 3D printers prefer to print holes that are vertical, meaning the direction of gravity does not cause the part to sag. This is especially important for holes with important tolerances. These issues are the primary reason we had to make many prototypes of each part before getting proper fit.

IR proximity sensors

Another issue we encountered was a lack of consistency with the IR proximity sensors. We found that some work more reliably, or at different distances than others. On each side of the chassis we had two sensors, one downward facing and the other looking forward. We had to try many different sensors before we found four suitable for their role. This is because the downward sensors must be constantly triggered, and the forward sensors must only trigger when there is an object nearby. We were having an issue where the sensor outputs would momentarily flicker on a very short time scale. To account for this, we disabled the hardware interrupts in the code, opting instead to check the sensor values are three predetermined locations in the movement routine. This greatly reduced the number of false readings as it was highly improbable the sensor values were checked during a flicker event.

Axles and center-mounted motors

We made a late, and somewhat major design change when we decided to abandon the idea of center mounting our motors and connecting them to the wheels with axles. This was a difficult decison because of the substantial amount of time spent prototyping axles, but we decided it would be best to connect 4 motors directly to wheels. This was a good choice as the new motor mounts worked almost immediately, allowing us to begin testing the final control software instead of continuing to prototype parts.

Talking to the H-bridge

We had some substantial difficulty controlling the H-bridge using PWM pins on the Raspberry Pi. The H-bridge can be powered in various ways, and our cheap controller board did not have good documentation. We managed to wire and program the H-bridge in ways that almost worked, or sort of worked, or worked sometimes. This made the issue hard to debug, because we could not be sure if we were using the bord correctly. We eventually figured out that we had been powering the position called 5V in, which however becomes a 5V output when the 5V jumper pin is bridged. So we were in fact sending power to an output which was powering the board enough to make it look like it was working some of the time.

Once we had figured out how to power the board we found that it would still occasionally fail for seemingly no reason. After much more time spent trying various changes and following many dead ends,

we figured out that the pi and the H-bridge were not sharing a ground. We had originally assumed that the two would be implicitly connected to a common ground because they were both connected to the same power supply. We fixed this by tying a ground pin from the pi to the ground of the H-bridge. After these changes, we had a stable system and did not encounter any more instability.

Acquiring data set images

We originally attempted to gather data for our data set by using the pi camera. We had a program set up where the monitor was the video output from the camera and different keystrokes would take an image and put it in an appropriate folder (ether signal or background). Despite our efforts at automation, it took over an hour to gather fewer than 400 images. We decided this process—while reflective of the images that the pi would take in production—was too slow. We switched to using burst mode on an iPhone, and relied on the neural network to generalize.

Single v. multi-threaded control processes

We originally intended to write multi-threaded control software, allowing the pi to acquire and process images, move the robot, and respond to proximity sensors at the same time. The original design was based on two control routines: an image sensing loop that set a global boolean called “stop”, and hardware interrupts allowing the sensors to stop the motors. This design is simple because the image processing routine is responsible only for maintaining a single variable—it is unaware of the complexity of the robot. Likewise, the interrupts are straightforward because those were unrelated to the image processing. The robot’s requirements lend themselves naturally to multi-threading.

In practice, the noisy response from the sensors made this approach difficult to implement within the time available. We would have preferred to use timers to double-check the sensor readings, but given our deadline, we chose the simpler (to implement) approach of running all processes in a single loop. We also slowed the robot motion and used discrete steps to accommodate the processing time required to recognize images.

Given more time, we could have implemented multi-threading and had a smoother motion and better software design. However, overall, the step-based movement wound up looking rather intentional and so did not detract from the project.

Appendix

- **datamaker_finalreport.py**: This is the image preprocessing code that takes in a number of folders of jpg images and converts them into dataset images by making the grayscale and flipping them.
- **final_report_neural_Network.ipynb:** This is the jupyter notebook that does the actual training.
- **FINAL_PROGRAM.py:** This is the code that the raspberry pi lamp runs during operation.
- **wholeassembly.stl** This is the final CAD model.