Special Delivery: Autonomous Shipping Logistics

[EECS 149/249A Class Project]

Peter Letizia
petera88@berkeley.edu

Gabriel Fan
gfan95@berkeley.edu

Department of Electrical Engineering and Computer Science
University of California
Berkeley, CA

ABSTRACT

Achievements in technology have increased the burden on the shipping industry to maintain efficient logistics for door to door delivery. More small businesses require convenient delivery methods to its consumers, which requires driver coverage of a local area each day.

This increase in e-commerce can be helped if real time parcel information is collected in advance. A parcel has a weight, a volume, and a ship to/from address. Our project focuses on the volume/dimensions of a package, and autonomously feeding this information to a back end service.

1. INTRODUCTION

A straight forward approach to obtain dimensions of a box autonomously would be to use three time of flight sensors calibrated to a known distance. From this, you can quite trivially extract the three dimensions. However, this requires a setup physically large enough to fit the largest possible volume. We wanted a less invasive way to grab this information. Specifically, we wanted a single, overhead system, that could robustly compute the dimensions, without placing identifiers on the box itself to locate its corners. This requirement is the basis for our project.

Our novel approach is to use one range finder (aka TOF sensor) and one camera to extract the dimensions of the box, which involves finding the relation between TOF sensor readings and number of centimeters per pixel in the image.

1.1 Problem Statement

Our goal is to autonomously obtain dimensions of a box from a single reference frame. We also want to be able to do this in series so we can find the dimensions of multiple boxes in a row.

2. OVERVIEW

To achieve our goal we will need three values. Length, width, and height, each within a centimeter. A height can be measured and extrapolated directly from a calibrated time of flight reading. However, the length and width cannot be achieved so simply. What we will attempt to show, is how a centimeter/pixel length scale can accurately be achieved via the height reading, and then from an overhead image, the length and width can also be obtained, based on the coverage of the box within the image.

3. MODEL

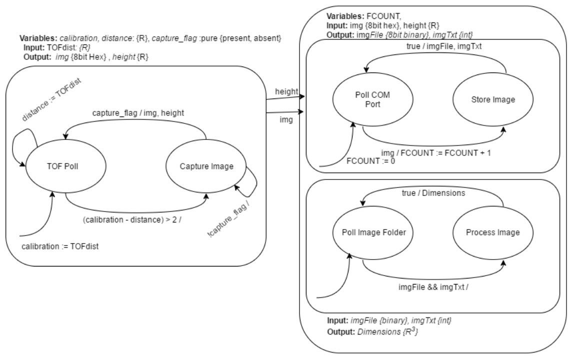
Variables: calibration, distance: {R}, capture_flag :pure {present, absent}
Input: TOFdist: {R}
Output: img {8bit Hex} , height {R}
distance := TOFdist
capture_flag / img, height
TOF Poll
Capture Image
(calibration - distance) > 2 /
!capture_flag /
calibration := TOFdist
height
img
Variables: FCOUNT,
Input: img {8bit hex}, height {R}
Output: imgFile {8bit binary}, imgTxt {int}
true / imgFile, imgTxt
Poll COM Port
Store Image
img / FCOUNT := FCOUNT + 1
FCOUNT := 0
true / Dimensions
Poll Image Folder
Process Image
imgFile && imgTxt /
Input: imgFile {binary}, imgTxt {int}
Output: Dimensions {R^3}

Figure 1: Cascade Composition of camera module with side by side synchronous python servers handling image processing.

Our model consists of two systems reacting independently, with a middle-ware handling the receiving and parsing of the image. The idea is to take the boxes in series. To do this, while avoiding "real-time" timing issues with human input, we needed the camera module which is snapping the photo, to be independent of the current state of the image processing. A middle-ware was added which will receive the image from the CMOS sensor, where after receiving all of the image data, stores it in a directory with shared access by the image processor. If the image processor detects a new file, it will read it in, and return the dimensions.

4. HARDWARE

(1) OV2640 CMOS Sensor.
(2) ArduCAM MicroController.
(3) Arduino Zero Micro Controller.
(4) HC-SR04 Ultra Sonic Sensor.
(5) Tripod

We went with an Arduino Zero despite the 32k limitation on its SRAM and dealing with image processing. Originally we were attempting to use an OV7670 CMOS sensor directly to the Arduino Zero, and reading in a single line of pixel information at a time. Our goal with this was to create a low cost module that didn't require a lot of resources for image

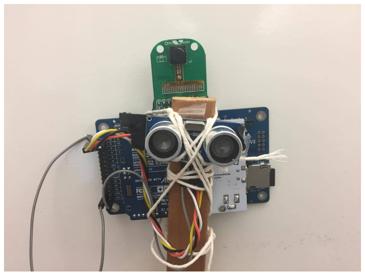
Figure 2: Picture of our hardware.

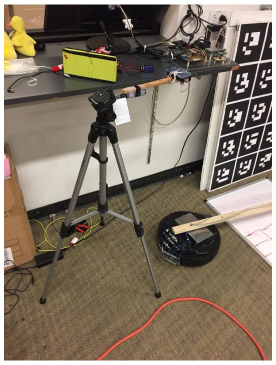
Figure 3: Picture of our setup.

processing. We were unsuccessful getting reliable CMOS sensor data to process one line at a time, so we attached a controller that includes a frame-buffer and a higher level api our group would be more comfortable working with. The next simplification we made in our model, was to purchase instead an OV2640 CMOS. This did a couple of things for us. For one, it was more compatible with the Arducam Controller we were now working above, and also the 2640 came equipped with a JPEG encoder. This allowed us to black-box our image on the frame buffer, and read out an already constructed image file in jpeg format. The final design included buffering the image from Arducam into the Arduino, and then outputting the data over USB interface.

5. IMAGE PROCESSING

We can easily obtain the height of the box using the range finder. However, to obtain the dimensions of the box along the other two dimensions, we need to apply image processing. Some assumptions we made were: (1) the background is uniform (no lines), (2) the box is vertically and horizontally aligned with the camera orientation, and (3) the box is centered (or close to center).

The two main parts of the algorithm is, first, to use the Sobel operator to find edges of the box and, second, to calibrate the algorithm to convert from pixel-to-cm lengths depending on the height of the box.

Fig. 4 shows our approximation of where the edges are. The brightness of each line represents the results we got after step 3 in our procedure found in the next section. The values are actually 1-D arrays, but we displayed them on a 2-D matrix for visualization purposes. As seen in the figure, the brightest four lines align with the edges of the box.

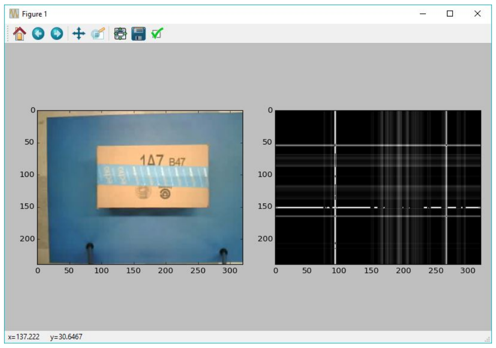

Figure 4: Edge approximations. The brightness of each line represents the length of edges along that row/column.

5.1 Algorithm Details

1. Use Sobel operator to obtain dx and dy values along the image vertically and horizontally.
2. Threshold the image so only parts of the image with high dx/dy values remain. (Also, ignore the outer 10% of the image since edges of our mat was seen in the image.)
3. Count the number of remaining values along each line (x- an y- directions done separately).
4. Find the locations of the longest edges in each half of the image (i.e. top/bottom, left/right)
5. Find the distance between edges to get vertical and horizontal lengths in pixels
6. Convert pixels to centimeters.

By following this procedure, we were generally able to accurately determine the dimensions of a box within a centimeter. The Sobel operator is an effective way of finding edges in the image. Then, thresholding the image allows us to isolate and find the lengths of the edges. Because we can assume a uniform background, the edges of the box must be the longest edges in the image. Finally, we just need to convert the distances between the edges of the box, vertically and horizontally, from number of pixels to centimeters.

5.2 Sobel Operator

The Sobel operator was introduced by Sobel and Feldman at the Stanford Artificial Intelligence Laboratory (SAIL) in 1968 [1]. What it does is it convolves a 3x3 matrix with the image matrix to produce dx/dy values. You do this in both x- and y- directions, separately. We have the two following matrices below, one for dx values and one for dy values, respectively.

$$S_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$S_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

In our code, this was done by using the ndimage library from SciPy (i.e. scipy.ndimage.sobel), which did this part for us. Fig. 5 shows the Sobel operator on a box we used.

5.3 Length Conversions (px to cm)

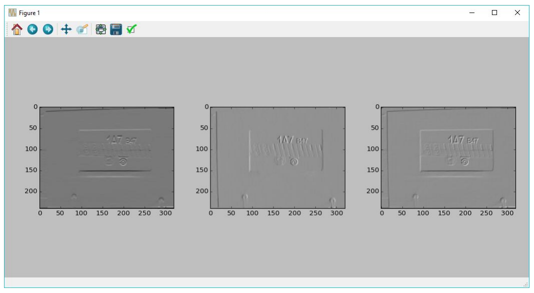

Figure 5: Screenshot of Sobel operator being used. Image 1 shows dy values, image 2 shows dx values, and image 3 is both dx and dy combined.

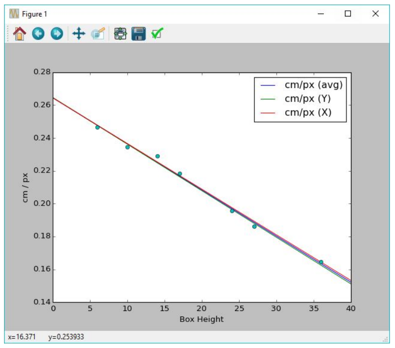

Figure 6: Graph of Box Height vs. cm/px conversion values

To tell the algorithm how to convert px to cm, we need to feed in data points of boxes of which we know the dimensions for already. To keep it simple, we took pictures of the same box but at different heights (by placing smaller boxes underneath). Intuitively, we figured the resulting line would be linear, such that as the height of the box increases, the cm/px conversion value becomes smaller. (As the box gets closer to the camera, more pixels are used to represent the box.) The data points confirm our intuition as shown in Fig. 6.

Then, when a new box comes in, we can use the height reading to determine the conversion rate using the linear equation we calculated from calibration, which was:

$$y = -0.002809x + 0.264550$$

where y is the cm/px rate,and x is the box height.

5.4 Limitations and Future Improvements

Although our algorithm tends to work fairly well and runs instantaneously (e.g. <1 second), there are some limitations to it. The main limitation is that the algorithm is not rotation invariant. In other words, the box cannot be rotated at an angle. In Fig. 7, we see that since the box is at an angle, our approximations for the edges has a greater range. One potential solution is to rotate the image until this range becomes thin, like in Fig. 4. In this case (without any rotations), our algorithm would simply choose the index that has the greatest value (which is generally somewhere in the middle).

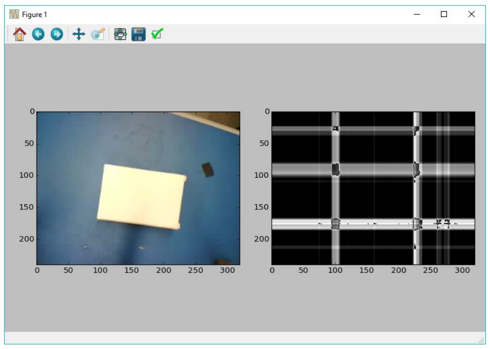

Figure 7: Edge Approximation on non-aligned box.

6. ANALYSIS

Overall, our project was able to successfully obtain the dimensions of a random box from a single height reading and a single image from above. The slowest part of the process was reading in the image through the CMOS sensor and sending it over to the computer. This required saving the image in a frame buffer and then sending the bytes over USB Serial one row at a time.

During the development phases with the Arduino, we ran into what seemed to be random behavior, and also missing data not showing up in our image processor. Since we wanted to write binary to a txt file on a windows machine we output binary data from the Arduino to our file receiver. This caused an issue with our delimiter to find new lines. Transferring binary gave us a 1/255 chance on every pixel, to appear as a ”new line.” This meant we were throwing away our pixel information. To fix this we converted it to Hexadecimal out from the Arduino using the Arduino print(str, HEX) function. However, we were still missing information across the wire. This turned out to be a loss in leading zeros during the conversion, which was also pixel information.

Once the hardware started working properly, and we were receiving quality photos for processing, we ran into a buffer overflow issue. Using Jpeg, we were able to snap photos that would take around 20k of SRAM to store. Our Arduino zero had 32k, about 3k of which was used for other data and the program itself, leaving us with about 29k for our stack. Our problem was with lighting and picture complexity. If the conditions were right, we would snap a photo that would blow out our stack. Since the Arduino allocates the rest of its remaining SRAM to the stack, we were overflowing into potentially other registers, or even the program itself. After overflowing the Arduino, it would come up in an ”unknown” state, and would not boot up properly. The only way to clear this, is to disconnect each component (Arducam, Rangefinder, Arduino), and let it sit for a couple of minutes. This was difficult to pin down, since the Arduino would hang at the time of taking the photo. This made it impossible to see the file size coming off of the frame-buffer, and determine in fact we were running out of SRAM.

Our image processing algorithm worked fairly robustly on new boxes. It even worked on a Raspberry Pi box which was small and had images and text on it. The dimensions our algorithm obtained was $7.57x12.01x3$ (cm) for length x width x height. The actual dimensions were $7.5x12x3$ (cm).

6.1 Key Concepts from Class

Real Time Behavior - The system has a human input factor. It is unpredictable when a box will be put under for measurement. We wanted to make sure we wouldn't miss any boxes, so to do this we separated the system into three modules. A sender, a receiver, and a processor. The processor and receiver work synchronously side by side. Under this architecture, data is properly received, even while another image is being read in and processed. This allows a box to be put under for measurement, even if the previous has not completed its processing yet.

Timing/Concurrency - Since we broke the receiving and processing up to allow real-time behavior, we had timing issues between the receiver and processor. If there is no task for the image processor currently running, then it may try to grab the image file before it has been completely written. To avoid this, if a new file is detected, we simply delay sufficient enough time for the initial image to be written, before reading it in for image processing.

Polling - We utilized polling for each module of the system. This allowed real time behavior based on a change of incoming value. We poll the range finder for a height reading above 2cm, the receiver polls the Arduino micro controller for a start flag showing the beginning of an image, and the image processor polls for a new file added to the directory.

7. MISSING PARTS

We were unable to successfully include the scale, to measure the weight of the box, and the Kobuki, to sort the boxes based on size, to our project.

The load sensors we had were generic load sensors, and we were unsure how to use them in the end. We were also unsure if our Arduino Zero would even be able to handle it since a large part of its memory was already being used for the image.

As for the Kobuki, we were able to successfully drive the Kobuki to three different locations based on which bumper sensor we hit (i.e. left bumper would tell the Kobuki to go to location 1, center bumper to location 2, and right bumper to location 3). However, we were unable to integrate Bluetooth from our computer to the myRIO. This would have allowed us to communicate to the Kobuki the size of the specified box from our computer after the image processing was done. The loading and unloading of the box from the Kobuki was left a mystery. The code is included in project submission for code.

8. CONCLUSION

Although we did not meet the distribution phase of our project, we did obtain dimensions of a box from a single reference frame. We achieved this without needing to add any identifiers such as a QR or April tag on top of the box to find the corners. To estimate a worst case execution time, we lowered the delays between our reads and writes to registers. The rate limiting factor is the hardware, and not the image processing. With this in mind, the worst case execution time is approximately 1.0 seconds per box. This is to allow enough time for the Arduino to buffer image data from the FIFO Frame Buffer on the Arducam, and then send it down a serial line for receiving.

Ideally we would like to achieve this process without a powerful machine behind receiving the image for processing. Since the Sobel edge detection algorithm we use requires a 3x3 matrix for each pixel, we are confident we can provide a method reading 3 lines of pixels at a time, and doing an on the fly processing to detect the edges. If this is the case, even the limited resources on the Arduino would be too much. An MC with enough stack to store the three lines should be sufficient, assuming it also has the proper peripherals.

9. ACKNOWLEDGMENTS

We would like to thank the GSIs and Professor Edward Lee for helping us through this project. Throughout the semester, Marten Lohstroh and Gil Lederman taught and helped us learn how to use embedded systems. We would like to give huge thanks to Professor Edward Lee for teaching this great class.

We also give thanks to our peers, all of which were open minded and helpful when asked. Several of the students provided clear explanations on some of the issues we were running into.

Finally Mark Potts, CTO of BridgePoint Systems Inc. here in Berkeley CA. He provided mentoring throughout the project from hardware design to implementation.

Our Arduino code ran on an Arduino Zero. Our server and image processing code are written in Python. The main libraries we used include: NumPy for computing with N-dimensional arrays, SciPy for computing Sobel edge detection, Matplotlib for plotting and displaying images, and Skimage for gray-scaling images for visualization.

10. REFERENCES

[1] Irwin Sobel. History and definition of the sobel operator. February 2014.

[2] The source code for the Arduino software in the Java environment is released under the GPL license and the C/C++ microcontroller libraries are under the LGPL license.

[3] ArduCAM source code created by "Ardu Lee". Copyright (C) 1991, 1999 Free Software Foundation, Inc. Released under the GNU LESSER GENERAL PUBLIC LICENSE.

[4] NumPy libraries started by Travis Oliphant. Copyright (c) 2005-2016, NumPy Developers. All rights reserved. Distributed under the BSD license.

[5] SciPy libraries created by Travis Oliphant, Pearu Peterson, Eric Jones. Copyright (c) 2003-2013 SciPy Developers. All rights reserved.

[6] Matplotlib libraries created by John Hunter. Copyright (c) 2012-2013 Matplotlib Development Team. All Rights Reserved. License based on Python Software Foundation (PSF) license.

[7] Scikit-image libraries created by Stefan van der Walt. Copyright (C) 2011, the scikit-image team. All rights reserved.

[8] Kobuki robot created by Yujin Robot. Code written by Daniel Stonier, Younghun Ju, Jorge Santos Simon, Marcus Liebhardt. Released under the BSD license.

[9] MyRIO created by National Instruments. Copyright (c) 2016 National Instruments. All rights reserved.

[10] Code for the HCSR04 Range Finder was taken and adapted from this tutorial by jsvester. URL: http://www.instructables.com/id/Simple-Arduino-and-HC-SR04-Example/?ALLSTEPS.

11. CODE

Link to source code on Github: https://github.com/gfan95/ee149-special-delivery.

12. DEMO VIDEO

Link to demo video on YouTube: https://www.youtube.com/watch?v=FSyB3gWVOjA.