Date of completeion April 27, 2022.

Digital Object Identifier

Image Processing: Line Recognition and Mo the Robot

KAITLYN COLAW[1], MASON LEE[2,3], CAMRIN PALMER[2], AND M. CATHERINE YOPP[2]
[1]Department of Applied Aviation Sciences,Embry Riddle Aeronautical University, Daytona Beach, FL 32114 USA
[2]Department of Mathematics, Embry Riddle Aeronautical University, Daytona Beach, FL 32114 USA
[3]Department of Physical Sciences, Embry Riddle Aeronautical University, Daytona Beach, FL 32114 USA

ABSTRACT We describe digital image processing as a process in which computer algorithms are used to gather useful information from inputted images. This paper details the effective application of image processing that is used to create a digital line following robot. The topics of computer vision and pattern recognition are explored in order to develop an algorithm to recognize and follow lines. The elements of code included in this report identify that the created digital line following robot is capable of successfully tracing a given path with high accuracy. With the increasing use of image processing in today's world with things such as self-driving cars and facial recognition, this research displays just one application of image processing that exemplifies the goal of automating tasks that humans would normally do.

I. INTRODUCTION

DIGITAL image processing has become a major part of public technology over the past few decades. It forms the base for several complex functions, like facial recognition in smartphones, lane detection in self-driving cars, and pathfinding in the Starship robots on campus. But these complex functions can be broken down into processes of identifying the elements of a picture. Understanding and replicating this process allows for an advantage in the growing technology industry of the twenty-first century.

A. APPLICATIONS OF IMAGE PROCESSING

Image processing is how computers handle images. This includes any kind of photo manipulation you can think of using a computer or electronic device; from viewing images to providing highly detailed analyses of them. Image processing is already widely applied today. For example, in digital photo editing, image processing is used to resize, crop, recolor, and join images. It can also be used to detect image features, which makes it useful for facial recognition in smartphones, population databases, and forensic science. Medical technology uses image processing to detect fractures in X-rays, abnormalities in MRI or CT scans, and other medical imaging procedures. Smartphones also use image processing to scan documents.

One useful application of image processing is Computer Vision (CV). CV has been used to develop techniques to help computers identify and process the content of digital images and videos, with the goal of automating tasks that the human vision can do. So, a computer should be able to recognize objects in an image such as someone's face or traffic lights. A real-life example of this application is seen in self-driving cars. A computer will take in live footage of the roadway to analyze; CV helps in detecting obstacles, recognizing paths, and understanding environments. Specific to its application in Python coding, OpenCV is a very popular library that can be used for computer vision and image processing tasks. It is used to perform several tasks like face detection, objection tracking, and landmark detection to name a few.

B. APPLICATIONS OF LINE FOLLOWING ROBOT

As stated in the introduction of this report, image processing has many applications in modern technology. Specifically, this line-following robot can be used in actual robots. Whether they be cleaning robots or not, the same process applies. The robot's computer can analyze a surface (like a dirty countertop) and use the contrast between the color of the countertop and the color of the stain to detect a dirty spot. Once the dirty spot is detected, the robot can implement a procedure to clean instead of one to follow, as ours does. Or take for instance a delivery robot. On a campus, like Riddle, or in a place like a shopping mall, robots like Starship can deliver food to students or shoppers, respectively. The robot can follow the sidewalk, or a path may be painted in a brightly colored line which can be marked out ahead of time.

On a more technical level, line-following robots can be used in manufacturing. If a machine can project or draw a line to be cut on an object, a line-following robot or robotic

part can identify the line and cut where indicated. This could shave time off processes that require manual or machine-drawn lines instead. Not to mention it creates a variety of shapes and lines that could be projected on the material to be cut.

The applications for this technology vary across different fields of technology. Whether the line-following program is used in medical, manufacturing, or entertainment, it can be useful.

II. THEORY

Digital image processing is done in three stages:

1) inputting an image
2) analyzing or editing an image
3) outputting either another image or an analysis of the input

Once an image is input, there are many things a program can do to the image to affect the output. Since an image is a two-dimensional array of the x- and y-coordinates of the picture elements (pixels), an image can be displayed as an array of these elements. Input pictures can be analyzed in monochrome or in full color. The type of output from a digital processing process determines the type of program used. If the input and output is an image, the program is an image processing program. If an image is input and an analysis is output, the program is a computer vision program. If a description is input and an image is output, it is a computer graphics program. If a description is both the input and output, the function provides an artificial intelligence program. The focus of this project will include both an image processing program and a computer vision program specifically utilizing pattern recognition.

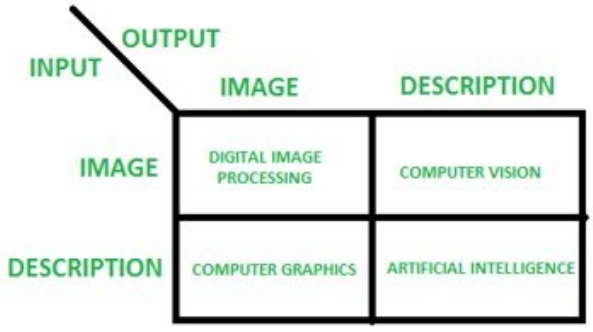

FIGURE 1. Diagram of Image Processing Applications. This diagram illustrates the applications of digital image processing based on the inputs and outputs.

A. EXPLORING OPENCV

Creating a code that could detect a line involved using a Python library equipped to read and edit images. To achieve this, an effort was made to learn OpenCV. This Python library has the capability to import images, change them to gray scale, recognize red, green, blue (RGB) values of pixels, resize and crop images, and analyze features of images including contrast and brightness. The primary features of OpenCV used in this project are importing an image and an inbuilt function called cv2.matchTemplate. The cv2.matchTemplate function works via a combination of the above princicples. It breaks an image down into its individual pixels, arranged into arrays of RGB values. By understanding the basics of OpenCV and image processing in Python, the line-following code was able to implement these simple commands into the complex final emulator.

One of the methods demonstrated in this paper is antialiasing. Aliasing is the reconstruction of a signal after it has been rendered to the point where it is no longer recognizable as its former self [1]. The goal of antialiasing is to decrease the distortion of the final signal, or image in this case. In digital processing, antialiasing can be used to convert analog information (photographs, television images, paintings, etc.) to digital information (a matrix of elements and pixels) [2]. For example, the resolution of a screen cannot capture all the details in a large, fine line painting so, antialiasing is used to minimize the possible distortion caused by attempting to represent such a high-resolution piece of work on a comparatively low-resolution screen. Antialiasing is also used in video games. When there are moving graphics, antialiasing each frame can be computationally costly. Changing the level and type of the antialiasing filter allows users to trade resolution with performance [3].

Another method that is applied is Otsu thresholding, which is a type of automatic thresholding. Thresholding is a technique where every pixel intensity is compared to a specified threshold value then put into groups of pixels under and over the specified value [5]. Otsu thresholding determines a threshold value by computing an image histogram, separating it into two clusters, and minimizing the weighted variance of the clusters [6]. The input image is then binarized based on this threshold value.

III. SYSTEM DESIGN

A. LINE FINDING

A function to extract outlines from the tops of complex images was also written using cv2 functions. Otsu thresholding and edge detection algorithms are applied. An input image is converted to grayscale, then PIL's edge detection function is used to extract all edges from the image. The PIL edge detection function had better results with the test images than the cv2 edge detection function; there were fewer edges. Otsu thresholding is then used to binarize the image and remove noise. The image is then transposed so that the columns can be iterated through as rows. Each row is looped through until a nonzero value is detected. The value is saved to an array of the same dimensions as the original image at the same location it was detected in. The end result is an image containing the top outline of the image.

```python
def linefinder(path):
    # read image and find edges
    img = Image.open(r'{}'.format(path))
    img_gray = img.convert('L') # L = luminance
    img_edg = img_gray.filter(ImageFilter.FIND_EDGES)
    img_edg.save(r"img_edges.jpg")

    # read new image and use Otsu Threshold
    img_edg1 = cv2.imread('img_edges.jpg')
    img_edg1_gray = cv2.cvtColor(img_edg1, cv2.COLOR_BGR2GRAY)
    thresh, img_otsu = cv2.threshold(img_edg1_gray, 0, 255, cv2.THRESH_OTSU)
    cv2.imwrite('img_otsu.jpg', img_otsu)

    # transpose new image
    otsuT = img_otsu.T
    cv2.imwrite('transposed_img_otsu.jpg', otsuT)

    # get image dimensions and find line
    img_size = img_otsu.shape
    img_line = np.zeros((img_size[0], img_size[1]))
    for row, i in enumerate(otsuT):
        i1 = i[3:] # buffer, edge image includes
        for col, j in enumerate(i1):
            if j == 255:
                img_line[col][row] = j
                break

    cv2.imwrite('img_line.jpg', img_line)
```

FIGURE 2. Linefinder Function. Shows the code for the linefinder function.

FIGURE 3. Original Test Image and its Edges. Shows the test image before and after applying PIL's edge detection function and Otsu thresholding.

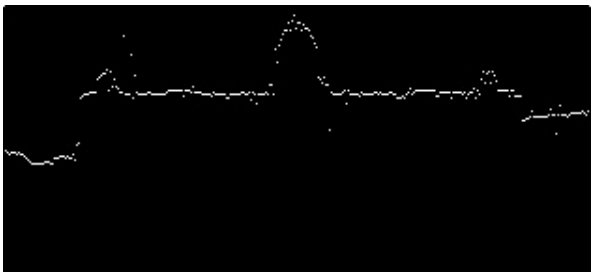

FIGURE 4. Output Line. Shows the top outline extracted from the image using linefinder.

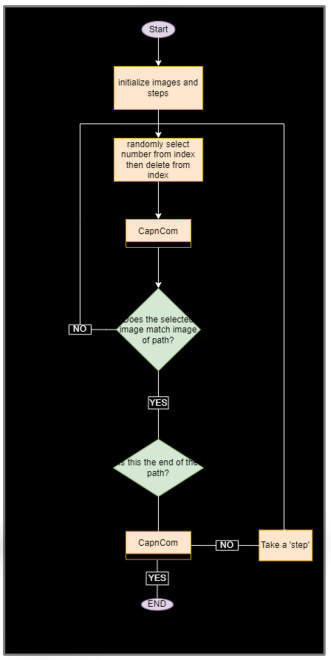

FIGURE 5. The flow chart describing the structure of the script *mo.py* which includes the implementation of *CapnCom*.

B. MO THE ROBOT

One of the simplest uses of pattern recognition is through a line-following-robot. They operate by having a dataset of the line that it is supposed to follow pre-loaded into its computer. For each movement that is made, it captures an image of what is in-front of it and compares it to the pre-loaded data set. The robot then turns itself to match the images that it captures to the dataset before it continues on the path. This project, however, does not implement a physical robot and so one was emulated–name Mo–through three Python scripts *CapnCom.py*, *mo.py*, and *mo_brain.py* (Fig. 5 & 6).

Emulation of a physical robot

The first two python scripts use a set of 20 images to emulate a robot 'capturing' a picture of the path in front of it. The 19 images include 18 digitally generated photos of a black line on a white background going from 0° to 180° in 10° increments (Appendix A). The 19^{th} image is a blank image which is how a physical line-following robot would be signaled that it has reached the end of its path.

Within the python scripts, each of these images is numbered and indexed. The first script, *Capncom*, includes the emulated pre-loaded dataset of the path that the robot is to follow. For this project, the hand drawn path in Fig. 7, was translated into angles to be coded into the script *CapnCom* by assigning each segment of the path to the number of the image that would best represent the turn necessary to draw the path (Fig. 8). Each segment is known as a 'step' in the code and for each step, the script *mo* runs an iterative loop randomly selecting a number from the indexed images to be compared to the appropriate path image that Mo is supposed to be 'looking' at during that step. This is to emulate the capture and turn correction of a physical robot. The selected number is then passed to the *CapnCom* script. If the image that was randomly does not match the correct image for that

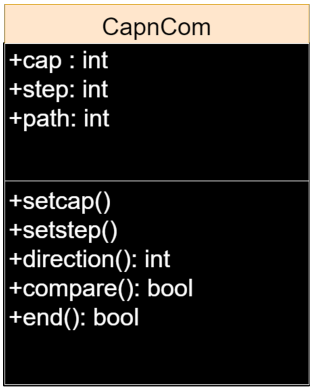

FIGURE 6. The internal functions and variables within the module *CapnCom.py* that allows Mo to compare and contrast it's 'captured' image to the path image and dictates the continuation of the integrated scripts.

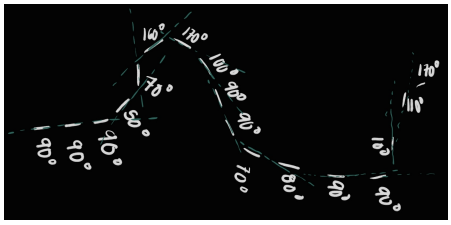

FIGURE 7. The hand drawn path the Mo is supposed to follow. The markings indicate the suggested angles to translate the path into code.

step, the loop continues. To prevent an excessive run-time, the incorrect image is removed from the index of available images until the next step.

C. PATTERN RECOGNITION

While the *CapnCom* script contains the function that determines what the pre-loaded path is and when it ends, it is the *mo_brain* script that implements the OpenCV library to conduct the pattern recognition through a function called *matchTemplate*. This function takes the RGB values of each

```
def setpath(self):

    if  self.step == 5:
        self.path = 1
    elif self.step == 10:
        self.path = 2
    elif self.step == 15:
        self.path = 3
    elif self.step == 25:
        self.path = 14
    elif self.step == 30:
        self.path = 13
    elif self.step == 35:
        self.path = 10
    elif self.step == 45:
        self.path = 2
    elif self.step == 55:
        self.path = 1
    elif self.step == 75:
        self.path = 3
    elif self.step == 85:
        self.path = 1
    elif self.step == 95:
        self.path = 4
    elif self.step == 105:
        self.path = 19
    else:
        self.path = 9
```

FIGURE 8. The pre-loaded path that Mo follows translated from the hand-drawn path in Fig. 7.

pixel in a template (the captured image) and compares is to a source image (the path image). To do this, both images are discretized into an array of RGB values by each of it's pixels. The template is then 'dragged' along the source image pixel by pixel. At each change in location, the Template's pixels are compared to the pixels of the source through Eq. 1 where x and y are the coordinates of the source image's pixels, x' and y' are the template image's pixels, and R is the resulting value at the source images pixel coordinates.

$$R(x,y) = \frac{\Sigma_{x',y'} T'(x',y') * I'(x+x', y+y')}{\sqrt{\Sigma_{x',y'} T'(x',y')^2 * I'(x+x', y+y')^2}} \quad (1)$$

It is then determined at which points on the source image R was the highest which can therefore estimate where in the source image the template image can be found. Because we are comparing two images, the values should be returned at approximately 1 for a match and 0 for a non-match. Although the correct captured image and the path image are the same, due to floating point arithmetic, the tolerance value for a match was set to 0.9. This value is returned within *CapnCom* and outputted in the *mo* script to dictate whether Mo should take a 'step' or 'turn' and 'look' again.

IV. RESULTS & DISCUSSION

A. UNDERSTANDING OPENCV

Implementing OpenCV in the line-following emulator allowed for the computer to analyze the image through much simpler processes. Using the OpenCV library, images can be converted to grayscale (see Fig. 9). Fortunately, the utilized path dataset was already in black and white. If this had not been the case, the cv2.cvtCOLOR function would have been utilized to convert the image to grayscale for the line-following code to read more easily.

```
img = cv2.imread("Resources/lena.png")

#Define a kernel for numpy
kernel = np.ones((5,5),np.uint8)

#CONVERTING TO GRAYSCALE
imgGray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

cv2.imshow("Gray Image",imgGray)
#cv2.waitKey(0)
```

FIGURE 9. Snapshot of code to change an image from RGB to grayscale. Python reads the image and converts the RGB values using the cv2.cvtCOLOR function.

In Fig. 10, the use of the cv2.cvtCOLOR function to change an image from color to grayscale is demonstrated. It's a simple technique, yet valuable to image processing because it enables easier line-detecting and anti-aliasing.

B. MO THE ROBOT

Seen in Fig. 11, Mo was able to trace the hand drawn path with high accuracy without straying or being interrupted, which is important in applications where high precision and accuracy are necessary.

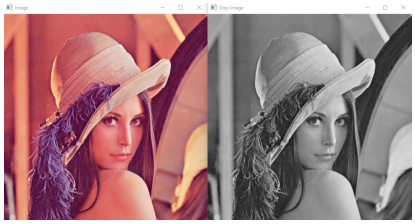

FIGURE 10. Demonstration of the conversion of a sample image from RGB pixel values to grayscale using the cv2.cvtCOLOR function.

FIGURE 11. The path completed by the emulated robot, Mo.

Completion Time

The design of the Mo the robot originally only had two scripts with the pattern recognition included in *CapnCom* but this method was computationally costly. This is because for every call of the function *compare()* from *CapnCom*, the images had to be imported and discretized for every iteration of the 'while' loop in *mo*. By placing the 24 lines of pattern recognition code into a module of its own, the images were only imported and discretized one time upon importation of *mo_brain* into *CapnCom* and then *CapnCom* into *mo*. This decreased the computational time by a factor of 10 since the only function that had to be repetitively used in terms of pattern recognition is *cv2.matchTemplate*.

REFERENCES

[1] Kesten, V. (2017). Evaluating Different Spatial Anti Aliasing Techniques. [bachelor's in computer science, KTH School of Computer Science and Communication]. KTH. https://kth.diva-portal.org/smash/get/diva2:1106244/FULLTEXT01.pdf

[2] Analog image processing vs Digital Image Processing - Javatpoint. www.javatpoint.com. (n.d.). Retrieved March 21, 2022, from https://www.javatpoint.com/analog-image-processing-vs-digital-image-processing.

[3] Cabading, Z. (2019, August 28). Anti-aliasing: Everything you need to know: HP® Tech takes. Anti-Aliasing: Everything You Need to Know | HP® Tech Takes. Retrieved March 21, 2022, from https://www.hp.com/us-en/shop/tech-takes/what-is-anti-aliasing.

[4] Kulhary, R. (2021, August 5). OpenCV - overview. GeeksforGeeks. Retrieved February 9, 2022, from https://www.geeksforgeeks.org/opencv-overview/.

[5] Dwivedi, P. (2018, January 15). OpenCV: Segmentation using Thresholding. GeeksforGeeks. Retrieved April 26, 2022, from https://www.geeksforgeeks.org/opencv-segmentation-using-thresholding/.

[6] Murzova, A. (2020, August 5). Otsu's Thresholding with OpenCV. LearnOpenCV. Retrieved April 26, 2022, from https://learnopencv.com/otsu-thresholding-with-opencv/.

APPENDIX A IMAGES TO BE COMPARED