Detection of Parking Occupancy Using Image Matching

A project by

Moshe Coleen N. Adique
Aaron Jimson L. Mandap
Victor Lorenzo B. Tirona

Submitted to

Luisito Agustin
Instructor, ELC 152.1

In Partial Fulfillment of the Requirements for the Course
ELC 152.1: Digital Signal Processing, Lecture

Department of Electronics, Computer, and Communications Engineering
School of Science and Engineering
Loyola Schools
Ateneo de Manila University
Quezon City, Philippines

December 2019

Abstract

The project aimed to create a parking occupancy detection application in C++ which detects individual parking slots and their corresponding occupancy. There are two image inputs to the application, the reference image or the parking lot without cars and the input image or the parking lot with cars. These two images are pre-processed through image alignment to increase their similarity and grayscaling and binarization to separate the foreground of cars and the background of the ground of the parking lot. The project was limited to using pictures of a setup made up of black illustration board, white tape, and yellow-white toy cars taken from a birds' eye view. The project successfully tested the application's effectivity only up to 12 parking spaces due to a limitation of time and illustration board. With this, the goal of parking occupancy detection was achieved. However, the researchers recommended that a more color-sensitive occupancy detection technique would be used in the future. The researchers recognize that there is much to be improved such as improving the morphological image operations for better alignment, improving the pre-processing to accommodate more parking setups, and developing the project to become real-time.

Acknowledgements

The researchers would like to thank the following for their contributions to the project. For without these people's guidance and support, the project would not have been as successful:

To Mr. Luisito L. Agustin, for pushing us to schedule consultations and providing constructive criticism which ultimately improved the project.

To our fellow digital signal processing classmates who we shared our struggles with in the creation of our projects.

To our dearest family and friends for being a source of motivation and peace during the struggles encountered in the creation of this project.

Table of Contents

1. Introduction

1.1. Significance of the Study

Current solutions in detecting parking occupancy in malls and parking infrastructures make use of sonars. As the scale of these infrastructures increases, the more sensors the user needs to buy. Cameras, on the other hand, are abundant in some areas as they are used for security. These cameras have the potential to replace the need for individual sensors for each parking slot. Through digital signal processing, instead a singular camera could be used to detect the occupancy of multiple parking slots.

1.2. Objectives

The objectives of this study are:

1. To locate and distinguish individual parking spaces.
2. To distinguish an empty parking space from an occupied parking space.
3. To develop a Graphic User Interface which represents the location and number of parking spaces and their corresponding occupancy states.

1.3. Scope and Limitations

This study is limited to developing a parking occupancy detection system using wxDev C++ and will not delve into comparing the system to existing parking systems using sonars.

Due to limitations in the quantity, variation, and the angle of the photos that could be taken from places accessible to students, this study will make use of a recreation of a parking lot made out of illustration board and tape. This parking lot will be made up of two rows with a common line in the middle with varying numbers of rows. The study is also limited to a bird's eye view perspective so that none of the features of the parking lot will be covered.

1.3.1. Image Alignment

The image alignment algorithm can only accomodate for small translational shifts. This is because of the nature of the algorithm; it checks it for every possible translation which increases quadratically as the maximum shift increases linearly.

1.3.2. Parking Slot Setup and GUI Output

The parking setup is limited only to have two rows of parked cars with varying number of columns. This is because a general code that would account for all parking slots is difficult to achieve. Aside from this the addition of more parking rows introduces a lot of difficulties which include determining whether or not each horizontal line corresponds to one or two rows of parking. Although the code may handle varying number of vertical parking lines, the GUI Output can only deal with a maximum of 8 parking slots. This is because a more general form of the GUI to display the results is still needed.

2. Theoretical Background

2.1. RGB to Grayscale Conversion

The 8-bit color format, commonly known as grayscale, has 2^8 or 256 different shades. The shades range from 0 to 255, where 0 represents black, 127 stands for gray, and 255 for white. The 24-bit color format, on the other hand, is known as the true color format. The 24 bits are divided into 3 — 8 bits are allocated for each of the three different color channels of red, green, and blue.

There are two ways of converting an RGB image into grayscale. The first one is the average method which only computes for the average of the three colors. The equivalent grayscale value is equivalent to the sum of the R, G, and B values divided by three [1]. This method assumes that the RGB values have equal contributions in the formation of the image. The second method, however, is the weighted or luminosity method which computes for the grayscale value according to the contribution of the RGB values [1]. The equation is given in (1).

$$¿ = 0.299 * R + 0.587 * G + 0.114 * B \qquad (1)$$

2.2. Otsu Binarization

Otsu binarization, developed by Nobuyuki Otsu, is one of many binarization algorithms. It aims to convert a grayscale image into a binarized image (black and white). The algorithm finds the most appropriate threshold which would separate the foreground from the background. It aims to find the threshold that would result in the least within class variance or spread in the foreground and the background. This also translates to finding the highest variance between the foreground and the background. The calculations of these variances are detailed in equations below [2]. The within class variance denoted by $\sigma^2$was computed by getting the sum of the product of the weight of the background, W_b, and the background variance,σ_b^2, and the product of the weight of the foreground, W_f, and the background variance,σ_f^2. In the equations below, P(i) is the number of pixels that has a certain color value, i; P is the number of pixels in the image; $P_b$is the number of pixels in the background; $P_f$is the number of pixels in the foreground; t is the threshold; $\mu_b$andμ_f are the mean of the background and the mean of the foreground, respectively.

$$\sigma^2 = W_b \sigma_b^2 + W_f \sigma_f^2 \qquad (2)$$

$$W_b(t) = \frac{\sum_{i=0}^{t} P(i)}{\square} \quad (3)$$

$$\mu_b(t) = \frac{\sum_{i=0}^{t} i * P(i)}{\square} \quad (4)$$

$$\sigma_b^2(t) = ¿ \frac{\sum_{i=0}^{t} (i-\mu_b)^2 * P(i)}{\square} \quad (5)$$

$$W_f(t) = \frac{\sum_{i=t+1}^{255} P(i)}{\square} \quad (6)$$

$$\mu_f(t) = \frac{\sum_{i=t+1}^{255} i * P(i)}{\square} \quad (7)$$

$$\sigma_f^2(t) = ¿ \frac{\sum_{i=t+1}^{255} (i-\mu_f)^2 * P(i)}{\square} \quad (8)$$

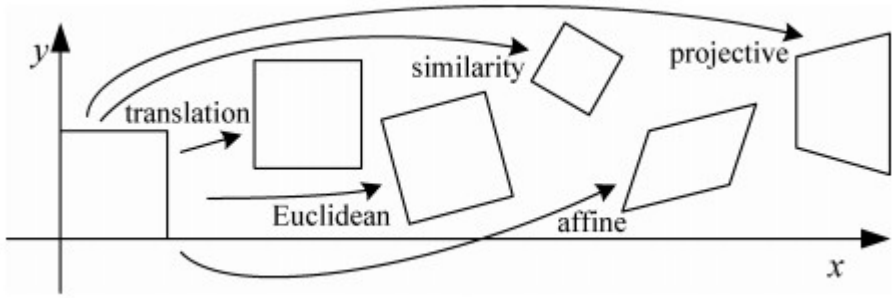
As discussed earlier, finding the threshold value that would result in the lowers within class variance is the same as finding the threshold value that would result in the highest between class variance. Since finding the highest between class variance is faster to compute, it is more appropriate for applications [2]. To compute for the between class variance, the equation below may be used [2]:

$$\sigma^2 = W_b W_f (\mu_b - \mu_f)^2 \quad (9)$$

2.3. Image Alignment

Image alignment entails discovering the corresponding relationships among images that share certain features. Image alignment algorithms are suited for various applications such as video stabilization and image stitching. There are two essential parts to image alignment: a method for alignment and a method for error detection.

For methods of alignment, it makes use of morphological image operations to increase similarity between two images by means of 2D and/or 3D motion such as rotation, projection, and translation. The various forms of alignment can be seen in figure below [3].



\

Figure 1. Forms of image alignment

Methods for error detection aim to determine the difference between two images. Alternatively, error detection aims to determine the best possible translation where the discrepancy between the two images are lowest. These algorithms vary from using direct and feature-based methods, full-search and hierarchal coarse-to-fine techniques. A simple way to quantify the error of comparing two images is to use the sum of squared differences of each of the images pixels. This is represented in the equation below [3]:

$$E_{SSD}(u) = \sum_{i}^{\square} ¿¿ \qquad (10)$$

In this equation, u is the displacement, $c_i(x,y)$ is the discrete pixel coordinations, I_0 is the template image, and I_2 is the image to be aligned.

2.4. Image Matching

Image matching is a method used to determine the similarity between two images by comparing them [4]. One way to implement this is to transform the image in such a way that its similarity with the reference is magnified, or to look at its global features. Another way is to determine regions in the image that have invariant and stable property [4]. These regions would be described by a vector of image pixels that can then be cross-correlated with the same regions in the reference image. There are other techniques that can be used in the implementation of image matching; these include feature detection, blob detection, template matching, speeded-up robust features (SURF) algorithm, and scale-invariant feature transform (SIFT) algorithm.

3. Related Literature

3.1. Smart Parking Using Image Processing

The research entitled “Smart Parking Using Image Processing” by Karthik et. al. aimed to explore the use of image processing to determine the occupancy of parking slots. Their setup involved putting circles in each parking slot as seen in the figure below [5].

Figure 2. System Setup

To detect the parking slots, they first converted the image to grayscale. They then used edge detection as a basis for their image binarization. These edges are then dilated to make them more prominent. After that they detected each circle and each detected circle correspond to an empty parking slot. This program flow can be seen in the figure below. As their output, they display the number of unoccupied parking slots in a GUI.

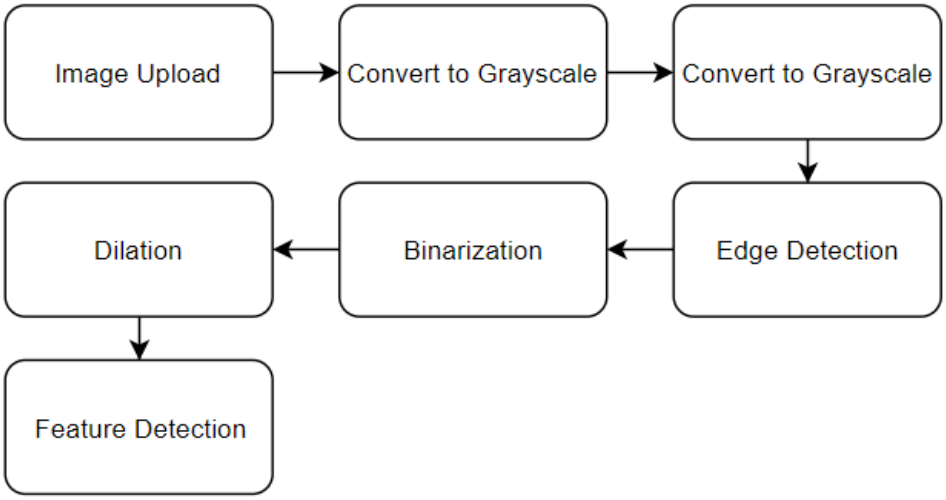


Figure 3. System Block Diagram

3.2. Intelligent Parking Space Detection Using Image Processing

The paper titled “Intelligent Parking Space Detection Using Image Processing” by R. Yusnita et al proposed the use of image processing in the detection of parking spaces. The system consists of five modules — system initialization, image acquisition, image segmentation, image enhancement, and image detection. The parking slots in the setup used in this project are determined using rounded brown images [6].

In the system initialization module, the program searches for these brown rounds to locate the parking slots. The second module is image acquisition where images of the parking scene are captured using a camera. The image segmentation module then converts the captured RGB image to grayscale and performs thresholding. This reduces the complexity of the image and makes recognition and classification easier. The thresholding technique used is basic thresholding where only a single threshold value is selected. Then, the image enhancement module removes the noise in the binarized image using morphological operations namely, dilation, erosion, opening and closing (binary operation). The binary operation opening removes the small objects in the image while closing removes the small holes. The boundaries of the objects in the image are also detected in this module. Lastly, the image detection module determines the shape of the objects in the image; it checks whether the image is round or not through the formula in (11). The value would be equal to 1 if the object is round, and less than 1 if it takes on a different shape [6].

$$Shape = \frac{4\imath \times Area}{Perimeter\square^2} \qquad (11)$$

4. Methodology

4.1. Basic Program Flow

The basic program flow is illustrated in the figure below. The program starts by asking the user to upload a reference image and an input image. After this, these images will be aligned to make comparisons between the images more accurate. After image alignment, the images will be first converted to grayscale so that otsu's binarization may be applied. After that, the slots will be detected by locating the parking lines. After locating the lines, the occupancy of the slots will be checked.

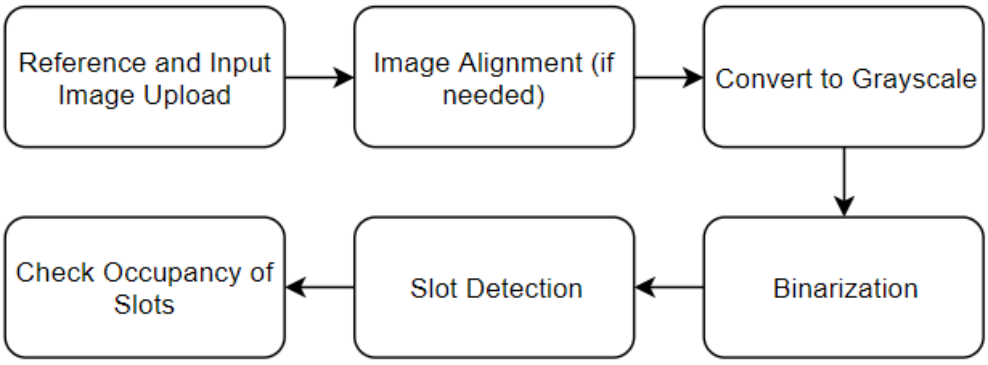


Figure 4. System Block Diagram

4.2. Setup

The setup used, as mentioned before, is composed of two illustration boards and white tape. The illustration boards serve as the black concrete pavement of the parking lot and white tape serves as the parking lot lines. A bird's eye view of the setup can be seen in Fig. 5.

Two illustration boards were used to allow the researchers to vary the number of parking lots seen by the camera. As can be seen in the figure, the first illustration board is composed of the first four pairs of parking spaces on the left while the second illustration board is composed of the last two pairs of parking spaces on the right. The second illustration board can be moved to reduce the number of parking spaces with a maximum of 12 parking spaces. A minimum of two parking spaces can be recreated if the illustration board's black side is used to isolate two parking spaces.

Toy cars made of white and yellow plastic were used for the experiment. These cars can be seen in Fig. 6.

To obtain the necessary photos for processing, the camera of an iPhone XR was used to take a birds eye view picture of the setup with and without toy cars in the parking spaces. The camera was held steady by one of the researcher's hands. Photos were taken for 4, 6, 8, 10, and 12 parking slots with varying car occupancy in each of the cases. These cases will be explored in the succeeding sections.

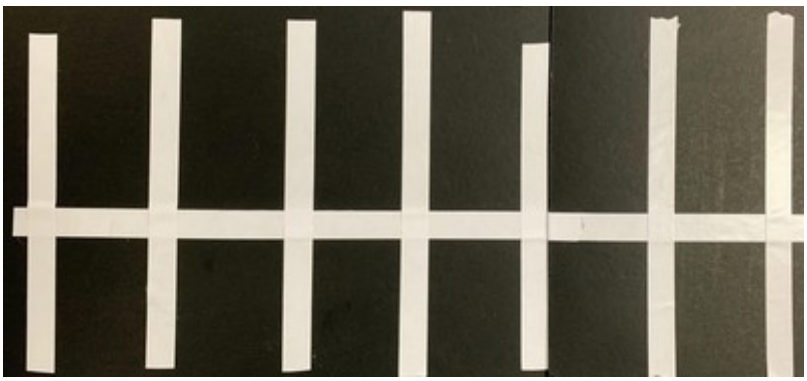


Figure 5. Parking Space Setup

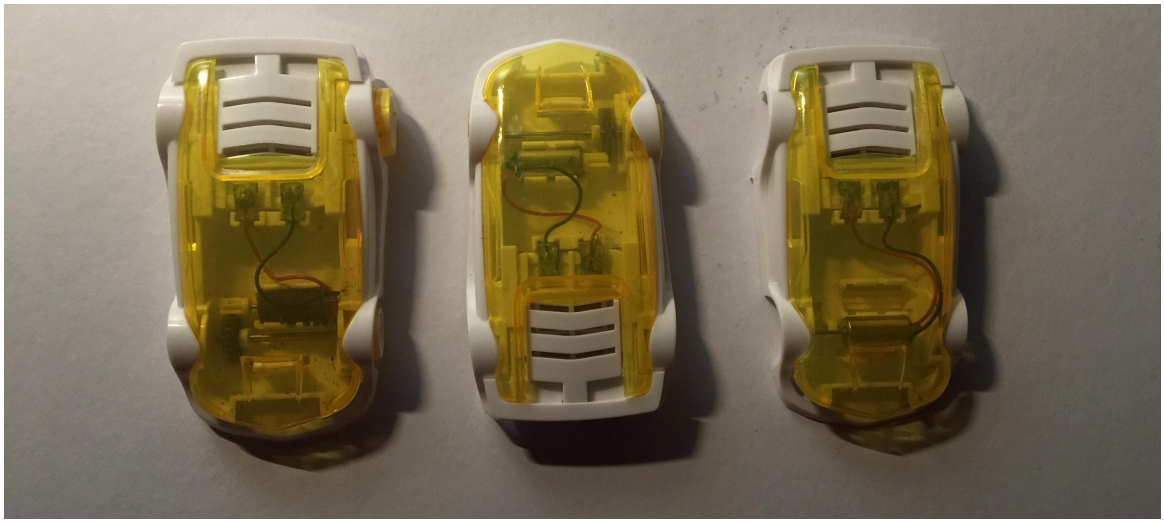


Figure 6. Plastic Cars Used for Experiment

4.3. Code

4.3.1. Image Loading

To load the images, the `wxImage.LoadFile();` was used. To specify the location, a file dialog was used because this allows the user to navigate and browse through the files of the computer. This value is then set to the `wxRichSetControl` object for visualization purposes. This path is also used as the path for the `wxImage.LoadFile()` command. The same process was used to load the reference image.

```
dlg_browse -> ShowModal();
edt_browse -> SetValue(dlg_browse -> GetPath());
```

```
inpImg.LoadFile(dlg_browse -> GetPath(), wxBITMAP_TYPE_ANY);
```

4.3.2. Image Alignment

To align the images, the sum of squared differences and full-search methods were used. Two images were loaded for this function: the reference image and the input image or the image to be aligned. This method shifts the input image by a varying number of pixels with respect to the reference image and obtains the sum of squared differences of each pixel in terms of the RGB values.

```
wxImage imageAlignment (wxImage refImg, wxImage inpImg)
{
  wxImage outImg = refImg;
  int halfShift=5;
  int refImgHeight = refImg.GetHeight();
  int refImgWidth = refImg.GetWidth();
  int product = refImgHeight*refImgWidth*255;
  int minSD = 3*product*product;
  int minx;
  int miny;
```

The function starts with obtaining the initializing the values of `halfShift` and `minSD`. The variable halfShift determines the maximum number of pixels the input image will translate horizontally and vertically during the comparison. The variable `minSD` stands for the lowest sum of squared differences obtained thus far. It is initialized at the maximum possible sum of squared difference for convenience later on in the code as it will be compared to lower values.

The function contains a maximum of four layers of `for` loops. The two upper loops make use of the `halfShift`. These two loops determine the number of pixels by which the input image is translated.

```
for (int x=-halfShift; x<halfShift;x++)
    {
        for (int y=-halfShift; y<halfShift;y++)
        {
            int squareDiff=0;
            int compR=0;
            int compG=0;
            int compB=0;
```

The two lower loops scan through each pixel in both images and compute the sum of their square differences with respect to the translation determined in the upper loops. The square differences of the RGB values are stored in the `compR`, `compG`, and `compB` variables respectively. Each `compR`, `compG`, and `compB` values are added onto the `squareDiff` variable.

```
for (int i=0; i<refImg.GetWidth(); i++)
{
```

```
  for (int j=0; j<refImg.GetHeight(); j++)
  {
    if  ((refImg.GetWidth()>=(i+x))  &&  (refImg.GetHeight()>=(j+y))  &&
((i+x)>=0) && ((j+y)>=0))
    {
       unsigned char r = refImg.GetRed(i,j);
       unsigned char g = refImg.GetGreen(i,j);
       unsigned char b = refImg.GetBlue(i,j);
       unsigned char r2 = inpImg.GetRed(i+x,j+y);
       unsigned char g2 = inpImg.GetGreen(i+x,j+y);
       unsigned char b2 = inpImg.GetBlue(i+x,j+y);
       compR=compR+(r-r2)*(r-r2);
       compG=compG+(g-g2)*(g-g2);
       compB=compB+(b-b2)*(b-b2);
       squareDiff=squareDiff+compR+compG+compB;
    }
  }
}
```

The `squareDiff` variable represents the sum of the squared differences for a specific translation. If this value is lower than the previously recorded sum of the squared differences, it will be stored in the variable `minSD`. Everytime a new lowest sum is stored, the translations x and y are recorded in the variables `minx` and `miny`. Once this is done, the code returns to the upper loops, equates `compR`, `compG`, `compB`, and `squareDiff` to 0, and begins computing for the square difference of another translation.

```
if (squareDiff<minSD)
{
  minSD=squareDiff;
  minx=x;
  miny=y;
}
```

Once the four `for` loops are finished, the minimum sum of square differences with its corresponding x and y translation values would have been obtained. All that is left is to create an image with the translated pixels and output the image using the variable `outImg`.

```
for (int i=0; i<refImg.GetWidth(); i++)
{
    for (int j=0; j<refImg.GetHeight(); j++)
    {
       unsigned char r = inpImg.GetRed(i-minx,j-miny);
       unsigned char g = inpImg.GetGreen(i-minx,j-miny);
       unsigned char b = inpImg.GetBlue(i-minx,j-miny);
       outImg.SetRGB(i,j,r,g,b);
    }
}
return outImg;
}
```

4.3.3. Grayscale Conversion

The algorithm for the RGB to Grayscale conversion is given below. It gets the red,

green, and blue values of each of the pixels of input `img` using `GetRed()`, `GetGreen()`, and `GetBlue()`. The values obtained are then used to compute for the unsigned character `gray` following the formula given in (1). The RGB values of the pixels are then set to the computed `gray` value.

```
for (int i = 0; i < img.GetWidth(); i++)
{
  for (int j = 0; j < img.GetHeight(); j++)
  {
    unsigned char r = img.GetRed(i,j);
    unsigned char g = img.GetGreen(i,j);
    unsigned char b = img.GetBlue(i,j);
    unsigned char gray = (r*306 + g*601 + b*117)>>10;
    img.SetRGB(i,j,gray,gray,gray);
  }
}
return img;
```

4.3.4. Otsu Binarization

The otsu binarization was discussed earlier. The fast way of its computation requires the computation of the background weight, the foreground weight, the background mean, and the foreground mean. The code starts by creating an array, `histogram`, to represent the possible grayscale values of each pixel. It then scans all the pixels of the image to count the frequency of each grayscale value. After this, it then computes for the sum of the product of the frequency and the threshold values and stores it to the variable, `sum`; and the number of pixels in the image, stored in the variable `totalFreq`, for the computation of the mentioned required variables. The frequency of the background pixels and the foreground pixels (stored in `totalBackground` and `totalForeground`, respectively) are also counted. Aside from this, the product of the threshold and the background frequency was also computed and stored in the variable, `sumBackground`. The product of the threshold and the foreground frequency may be computed easier by subtracting `sumBackground` to `sum`. After this the background and foreground weights were computed by dividing the frequency of the background by the total number of pixels and by dividing the frequency of the foreground by the total number of pixels, respectively. The background mean was computed by dividing `sumBackground` by `totalBackground` and the foreground mean was computed similarly. Finally the between class variance was computed by multiplying the foreground weight, the background weight, and the square of the difference of the means; this product is stored in `bcVar`. The maximum between class variable was first set to 0 and is changed

every time a higher between class variance was obtained. This concludes the computations for the appropriate threshold.

```
wxImage binarization(wxImage img)
{
  //Histogram to store the number of pixels in a grayscale value
  double histogram[256] = {};
  unsigned char value;

  //Counting of the pixels
  for (int i=0; i<img.GetWidth(); i++)
  {
    for (int j=0; j<img.GetHeight(); j++)
    {
      unsigned char r = img.GetRed(i,j);
      unsigned char g = img.GetGreen(i,j);
      unsigned char b = img.GetBlue(i,j);
      unsigned char value = (r*306 + g*601 + b*117)>>10;
      histogram[value] +=1;
    }
  }

  //Sum of all frequencies times the threshold for the mean computation
  double sum = 0;
  for (int t=0 ; t<256 ; t++)
  {
    sum += t*histogram[t];
  }

  //Sum all the frequencies for the mean computation
  double totalFreq = 0;
  for (int t=0 ; t<256 ; t++)
  {
    totalFreq += histogram[t];
  }

  double sumBackground = 0;
  double bgWeight = 0;
  double fgWeight = 0;
  double totalBackground = 0;
  double totalForeground = 0;

  double maxVar = 0;
  double threshold = 0;

  for (int t=0 ; t<256 ; t++)
  {
    //frequency of background pixels
    totalBackground += histogram[t];
    if (totalBackground == 0) continue;

    //frequency to get the foreground frequency
    totalForeground = totalFreq - totalBackground;
    if (totalForeground == 0) break;

    //Sum of all frequencies times the possible threshold until the
current threshold
    sumBackground += t*histogram[t];

    double bgMean = sumBackground / totalBackground;
```

```
    double fgMean = (sum - sumBackground) / totalForeground;

    bgWeight = totalBackground / totalFreq;
    fgWeight = totalForeground / totalFreq;

    // Calculate Between Class Variance
    double bcVar = bgWeight * fgWeight *
      (bgMean - fgMean) * (bgMean - fgMean);

    // Check for new maximum
    if (bcVar > maxVar)
    {
      maxVar = bcVar;
      threshold = t;
    }
  }
```

To binarize the image, the grayscale color values of each pixel was checked if they fall below the threshold. When it is below the threshold it was set to black; otherwise, it was set to white.

```
  //Set pixels below threshold to black; otherwise, set it to white
  for (int i=0; i<img.GetWidth(); i++)
  {
    for (int j=0; j<img.GetHeight(); j++)
    {
      unsigned char value = img.GetRed(i,j);
      if(value < threshold)
      {
        img.SetRGB(i,j,0,0,0);
      }
      else
      {
        img.SetRGB(i,j,255,255,255);
      }
    }
  }

  return img;
}
```

4.3.5. Parking Line and Slot Detection

The parking slot detection starts with the detection of key points. In the setup discussed above, it can be seen that the key features of the parking slot are the vertical lines and horizontal lines that bound it. The code starts with determining the locations of the vertical lines. Given the nature of the setup of having a black parking lot with white lines, the code first detects a white line and then a black line to establish one vertical line in the parking lot. The white line is the left border of the line while the black line is the right border of the line. Making this approach also avoids the problem of detecting the same line multiple times. To detect the vertical line, 75 percent of the column of pixels has to be white or black and they must span across at least 5 columns of pixels. When a line is detected, they are stored in a vector, `verticals`. At the end of this code the

locations of the left border and the right border of the verticals lines are stored in the said vector.

```
unsigned char value;
vector<int> verticals;
int refImgWidth = refImg.GetWidth();
int refImgHeight = refImg.GetHeight();
int verticalCondition = refImgHeight*3/4;
int pixel = 0;
int line = 0;
int state = 0;

//binarize Image
refImg = binarization(refImg);

//find vertical lines
for (int i=0; i<refImgWidth; i++)
{
  for (int j=0; j<refImgHeight; j++)
  {
    value = refImg.GetRed(i,j);
    if (state == 0)
    {
      if (value == 255)
      {
        pixel++;
      }
    }
    else
    {
      if (value == 0)
      {
        pixel++;
      }
    }
  }
  if (pixel > verticalCondition)
  {
    line++;
  }
  if (line > 4)
  {
    verticals.push_back(i-4);
    line = 0;
    if (state == 0)
    {
      state = 1;
    }
    else if (state == 1)
    {
      state = 0;
    }
  }
  pixel = 0;
}
```

The detection of horizontal lines comes after the detection of vertical lines because it relies on the location of vertical lines. To avoid the interference of the vertical

lines in the detection of horizontal lines, scanning for it only spans between two identified vertical lines. Similar to the vertical line, the code first detects a white line to detect the top border of the horizontal line and the succeeding black line to detect the bottom border of the line. A horizontal line must have 75 percent of the bounded row of pixels has to be white or black and they must span across at least 5 rows. The code stores the first row of pixels as the first horizontal line and the last row of pixels as the last horizontal line. This is because of the assumption that the first row of parking slots spans from the first row of pixels until the top border of the horizontal line and the second row of parking slots spans from the bottom border of the horizontal line to the last row of pixels. These key locations are all stored on the vector `horizontals`.

```
//find horizontal lines
vector<int> horizontals;
horizontals.push_back(0);
line = 0;
pixel = 0;
int horizontalCondition = (verticals[2]-verticals[1])*3/4;

for (int j=0; j<refImgHeight; j++)
{
  for (int i=verticals[1]; i<verticals[2]; i++)
  {
    value = refImg.GetRed(i,j);
    if (state == 0)
    {
      if (value == 255)
      {
        pixel++;
      }
    }
    else
    {
      if (value == 0)
      {
        pixel++;
      }
    }
  }
  if (pixel > horizontalCondition)
  {
    line++;
  }
  if (line > 4)
  {
    horizontals.push_back(j-4);
    line = 0;
    if (state == 0)
    {
      state = 1;
    }
    else if (state == 1)
    {
      state = 0;
    }
```

```
    }
    pixel = 0;
  }
  horizontals.push_back(refImgHeight-1);
```

4.3.6. Occupancy Detection

The code for occupancy detection makes use of the entries in the `horizontals` and `verticals` vectors obtained in the implementation of parking line and slot detection. The first pair of `for` loops sets the top, bottom, left, and right boundaries of a parking slot, while the second pair counts the number of white and black pixels within a particular slot.

The boundaries are set by obtaining the entries in the `horizontals` and `verticals` vectors. The row and column numbers of these bounds are stored in the `topBorder`, `bottomBorder`, `leftBorder`, and `rightBorder` variables. Next, the code counts the number of black and white pixels within these boundaries. It then computes for the ratio of white pixels to the total number of pixels in the particular slot. If the percentage of white pixels in the slot is greater than or equal to 10%, a value of 1 will be stored in the vector `slots` through the `push_back()` function, otherwise it will store 0. The size of `slots` should be equal to the total number of parking slots in the image; `slot[0]` will be slot 1, `slot[1]` will be slot 2, and so on. The entries in this vector will be the basis in determining the occupancy of the slots (0 for vacant slots, 1 for occupied ones).

```
  inpImg = binarization(inpImg);
  vector<int> slots;
  int slotNumber = (verticals.size()-2)/2;
  float countW = 0;
  float countB = 0;
  float ratio = 0;
  int topBorder, bottomBorder, leftBorder, rightBorder;
  for (int a=0; a<2; a++)
  {
    topBorder = horizontals[2*a];
    bottomBorder = horizontals[2*a+1];
    for (int b=1; b<=slotNumber; b++)
    {
      leftBorder = verticals[2*b-1];
      rightBorder = verticals[2*b];
      for (int i=leftBorder; i<rightBorder; i++)
      {
        for (int j=topBorder; j<bottomBorder; j++)
        {
          if(inpImg.GetRed(i,j) == 255)
          {
            countW++;
          }
          else
```

```
          {
            countB++;
          }
        }
      }
      ratio = countW/(countW+countB);
      if(ratio >= .3)
      {
        slots.push_back(1);
      }
      else
      {
        slots.push_back(0);
      }
      countW=0;
      countB=0;
      ratio = 0;
    }
  }
```

4.3.7. Output Box

The output of the preceding processes is a vector of the same size with the parking lot. Each element in the vector is either a 0 or 1 which corresponds to a slot being vacant or occupied, respectively. The first element in the vector corresponds to the first slot; the second element corresponds to the second slot; and so on. As the output of the program, this vector is interpreted and the contents of a textbox is changed accordingly. Aside from showing whether the slots are vacant or occupied, the program will also output the row and column of the parking slot to make it easier to locate the slot.

To implement the output text box, the code starts with a loop that would iterate 2 times. This corresponds to the two rows. The second loop goes from 0 to half of the size of the vector which corresponds to the columns. This approach allows the program to keep track of the row and column of the parking slot. A single text box is used and the information was appended to the textbox as information is read in the vector.

```
int size = slots.size();
int halfSize = (size+1)/2;
int c, e, d, f;
wxString b, row, column;
WxEdit1 -> SetValue("");
for (int i=0; i<2; i++)
{
  d = i+1;
  row = wxString::Format(wxT("%d"), (int)d);
  for (int a=0; a<halfSize; a++)
  {
    e = i*halfSize+a;
    c = e+1;
    f = a+1;
    column = wxString::Format(wxT("%d"), (int)f);
    if (slots[e] == 0)
    {
```

```
      b = wxString::Format(wxT("%d"), (int)c);
      WxEdit1 -> AppendText("SLOT ");
      WxEdit1 -> AppendText(b);
      WxEdit1 -> AppendText(" VACANT         ROW ");
      WxEdit1 -> AppendText(row);
      WxEdit1 -> AppendText("   COLUMN ");
      WxEdit1 -> AppendText(column);
      WxEdit1 -> AppendText("\n");
    }
    else if (slots[e] == 1)
    {
      b = wxString::Format(wxT("%d"), (int)c);
      WxEdit1 -> AppendText("SLOT ");
      WxEdit1 -> AppendText(b);
      WxEdit1 -> AppendText(" OCCUPIED     ROW ");
      WxEdit1 -> AppendText(row);
      WxEdit1 -> AppendText("   COLUMN ");
      WxEdit1 -> AppendText(column);
      WxEdit1 -> AppendText("\n");
    }
  }
}
```

5. Results

The succeeding paragraphs will discuss the testing of each step in the process of determining the parking lot's car occupancy. As previously mentioned, in detecting parking space occupancy, the input images are put through image alignment and binarization techniques to pre-process the images before comparing. From these techniques, the parking spaces are detected using the parking line and slot detection algorithm. The results for each stage in this process can be seen from Figures 7 to 10. It is important to note that these figures were obtained by editing the code such that the program stops at these techniques. This explains any discrepancy between these figures and the figures of the succeeding tests.

For image alignment, Fig. 7 shows the results of aligning an image with two cars to the reference image. It is important to note that the image with cars is not yet aligned to the reference image. The input image has more black space than the reference image at the bottom edge of the picture. It can be seen that in the output image, at the right side of the figure, black space at the bottom edge became smaller. This due to the image alignment.

For binarization, Fig. 8 shows the results of binarizing the same input image with two cars. It can be seen that the ground or the cardboard became black while the rest of the photo became white. However, upon further testing, the binarization technique was found to have limitations in terms of color. Figure 9 shows a color spectrum going through the binarization technique. The output image shows a black block where the blue-green spectrum was and a white block where the red-yellow spectrum was. This implies that colors closer to blue and green would be considered by the binarization technique to be part of the background. This may be a problem for detecting blue-green cars.

For parking slot detection, Fig. 10 shows an edited version of the program where red lines are printed in the photo where lines are detected. The x and y coordinates of these red lines are stored in the program and serve as the bounds for each parking slot.

To evaluate its effectiveness, the system was tested with parking lots with a varying number of parking spaces and with cars at different spaces. It is also notable to mention that the photos were taken by hand and thus there are small differences between the angle of each photo. Nevertheless, the system was able to detect each parking slot and their corresponding car occupancy state.

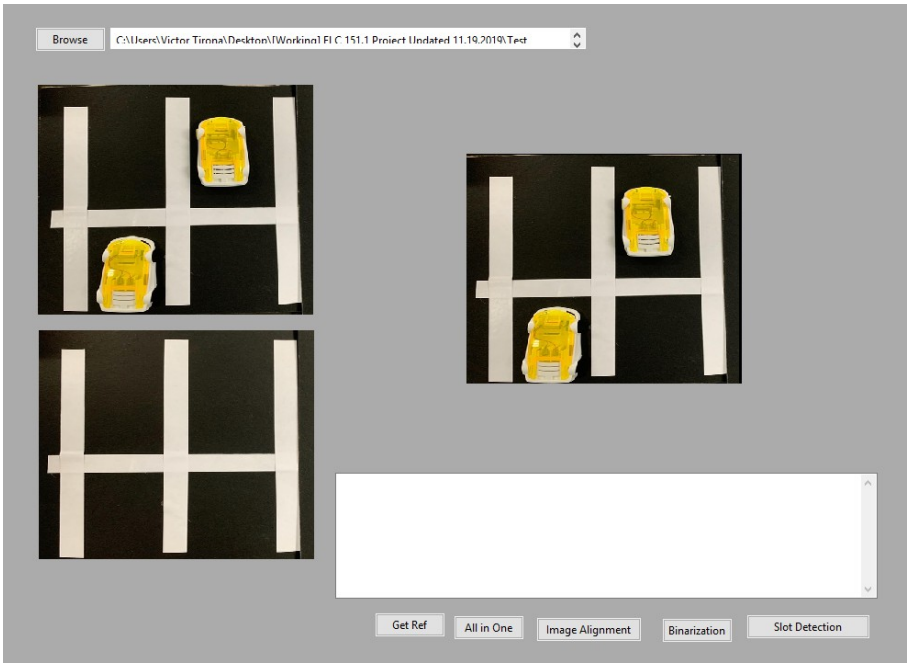


Fig. 7. Image Alignment for an Image with 2 Cars

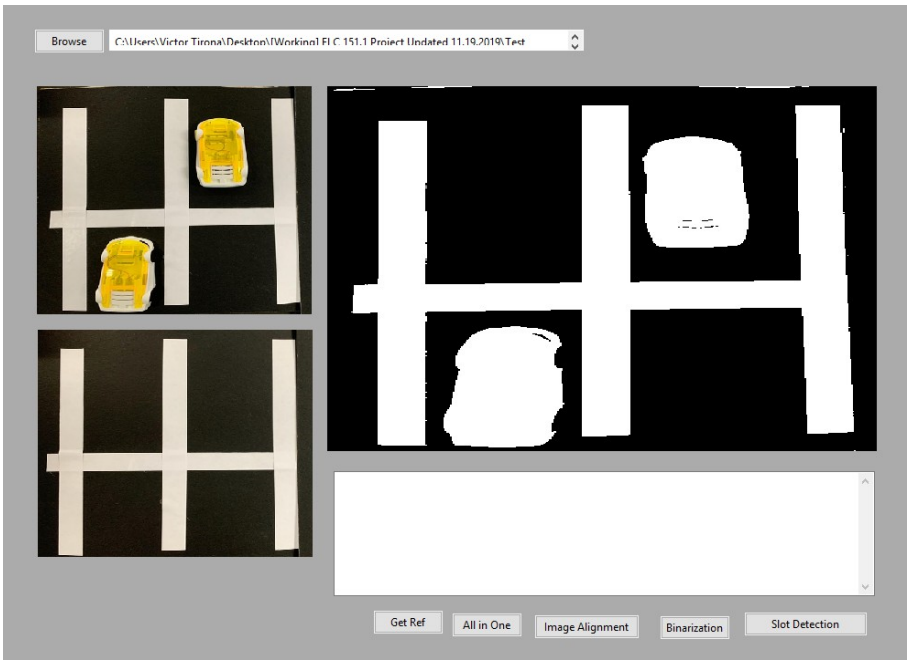


Fig. 8. Binarization for an Image with 2 Cars

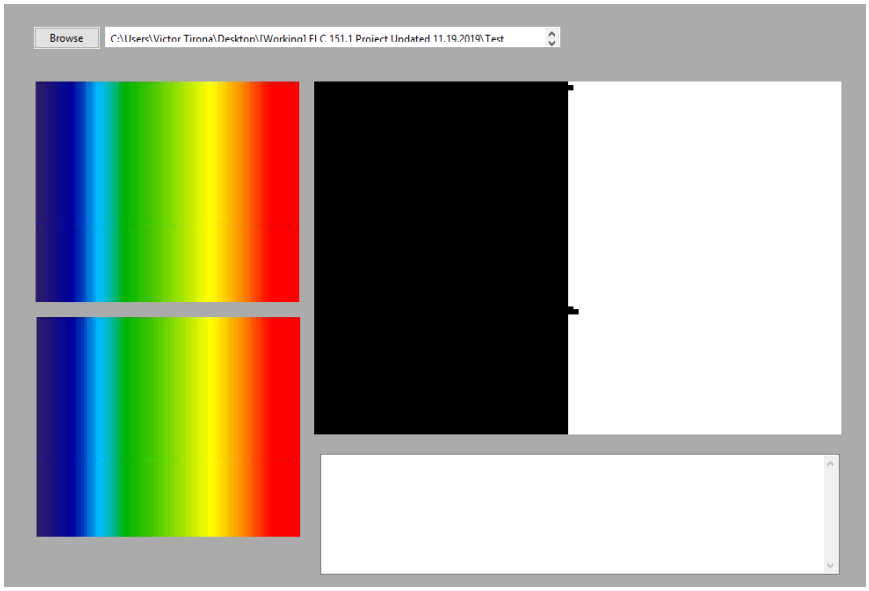


Fig. 9. Binarization of the Color Spectrum Showing the Limitation of the Algorithm to the Red-Yellow Spectrum

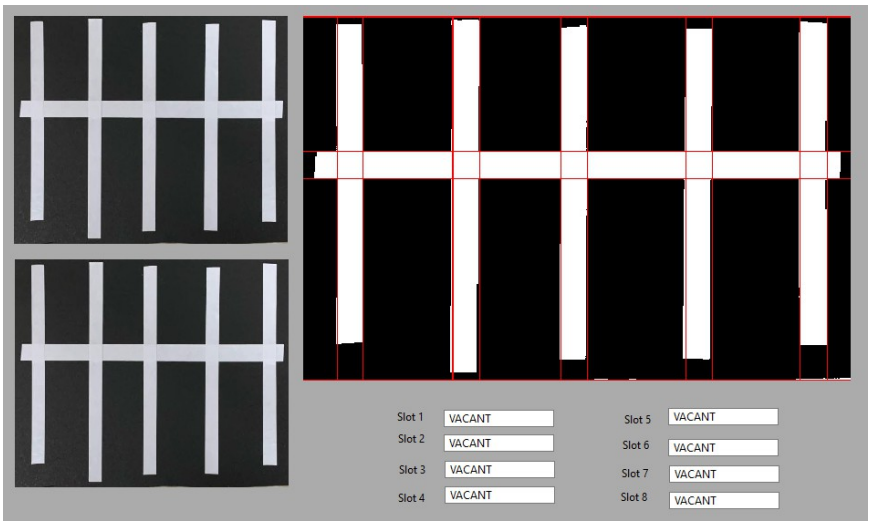


Fig. 10. Line and Parking Slot Detection with Red Markers

Figures 11 to 14 show the test cases where there are 4, 6, 10, and 12 parking spaces, respectively. The lower left image in each figure is the reference image, the upper left image is the input image, and the upper right image is the output image. The occupancy of each parking space can be seen in the output text box which labels each row and column with either “OCCUPIED” or “VACANT.” From this point onwards, the row and column will be referred to as RxCy where x and y are integers. The rows are counted from top to bottom while the columns are counted from left to right both starting with 1.

In the case where there are 4 parking spaces, one car each was placed on *R1C2* and *R2C1*. The output text box outputted that these corresponding row parking spaces were occupied while the others were not occupied.

In the case where there are 6 parking spaces, one car each was placed on *R1C2*, *R2C1*, and *R2C2*. The output text box outputted that these corresponding row parking spaces were occupied while the others were not occupied.

In the case where there are 10 parking spaces, only one parking spot was occupied in slot *R1C1*. The output reflected the occupancy of the parking space.

In the case where there are 12 parking spaces, three parking spaces were occupied. The cars were in parking spaces in *R1C1*, *R1C4*, and *R2C2*. The output, as can be seen in the figure, reflected these parking spaces' occupancy.

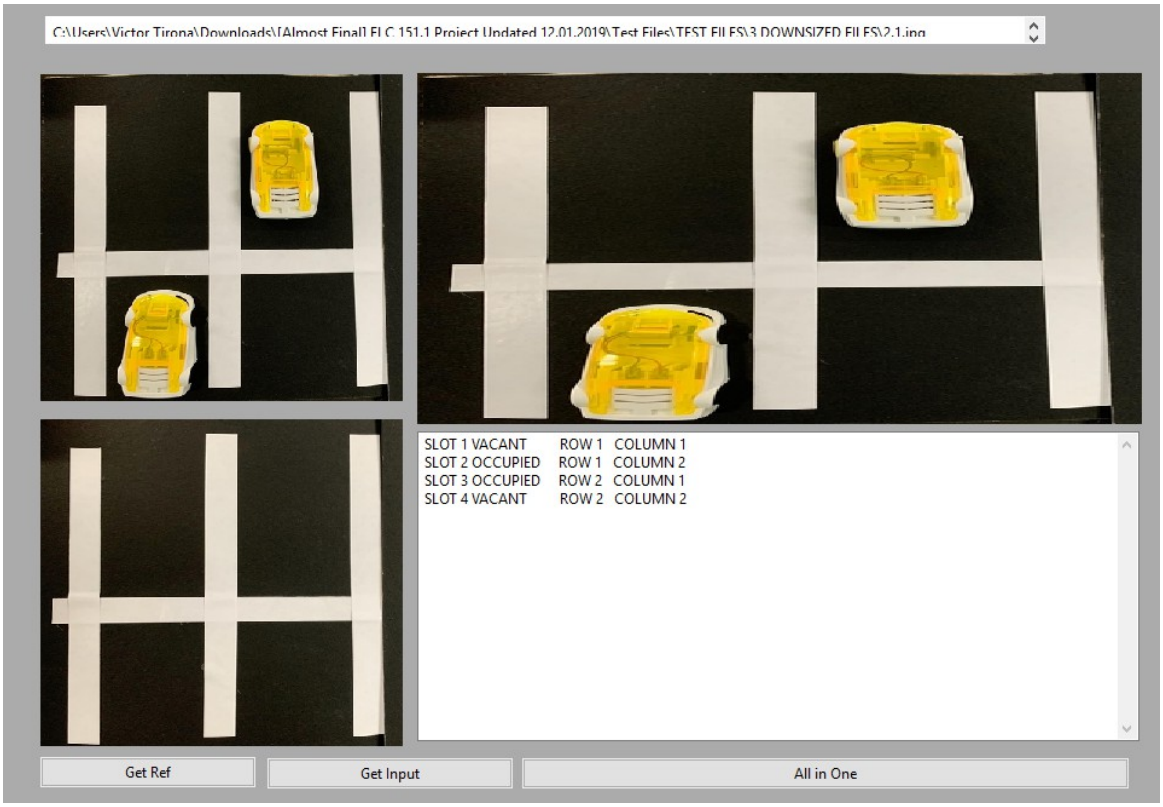


Fig. 11. Parking Occupancy Testing with Cars in Slots

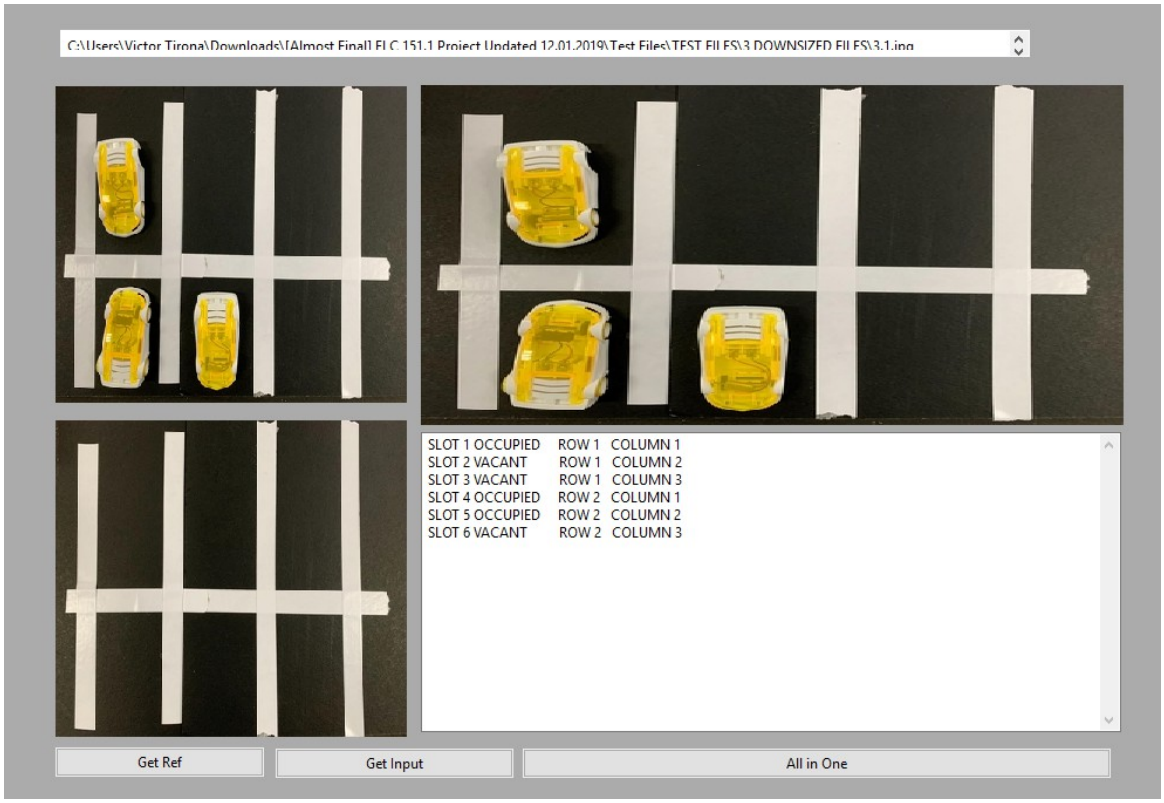


Fig. 12. Parking Occupancy Testing with Cars in Slots

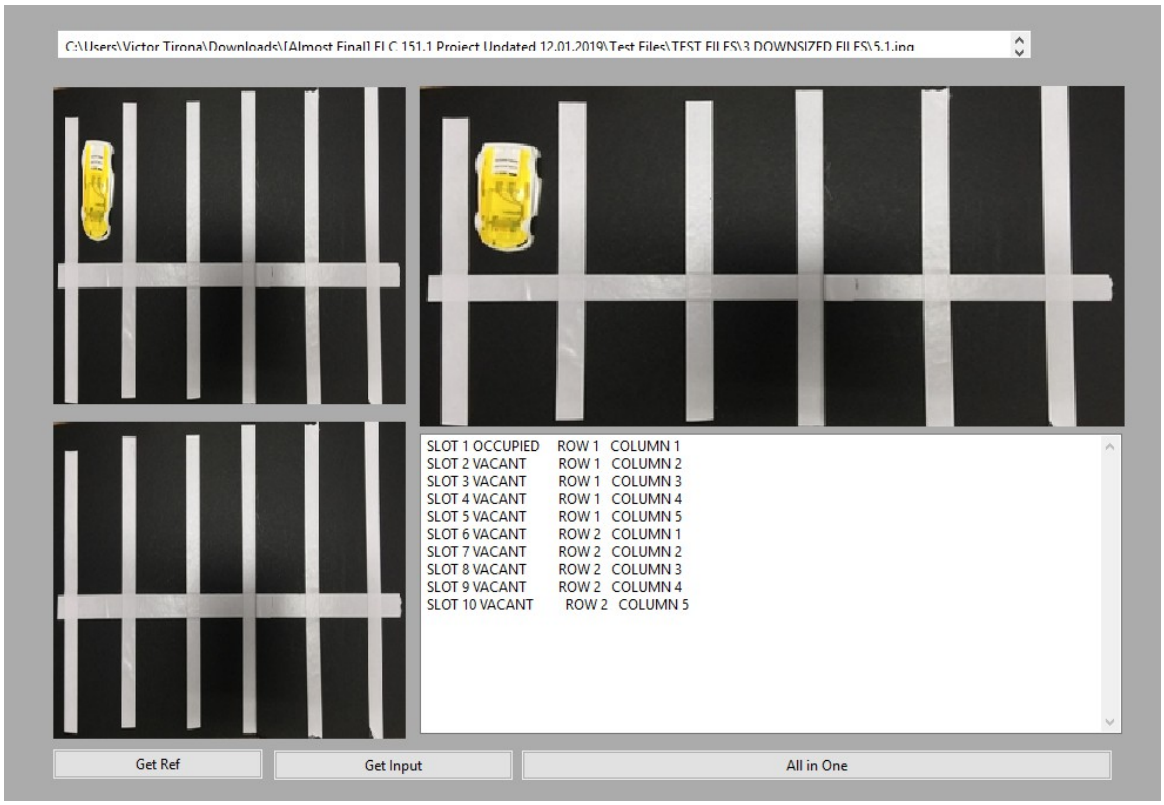


Fig. 13. Parking Occupancy Testing with Cars in Slots

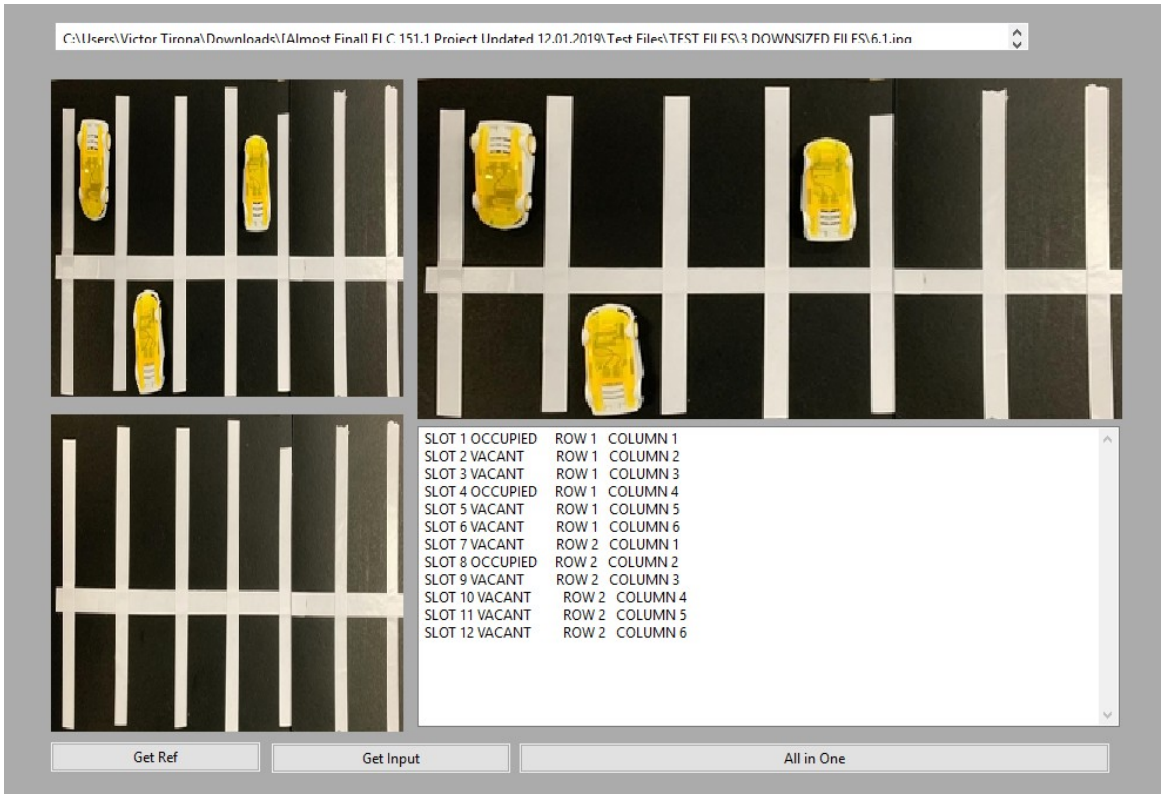


Fig. 14. Parking Occupancy Testing with Cars in Slots

6. Conclusion and Recommendations

The researchers were able to successfully develop a program that can be used in parking slot occupancy detection. The program consists of six modules: (1) image upload, (2) image alignment, (3) grayscale conversion, (4) binarization, (5) slot detection, and (6) occupancy detection. The first module loads the reference and input images uploaded by the user. The second module then aligns the input image with the reference image. Next, the third and fourth modules convert the RGB image into grayscale and binarizes the input image for easier recognition and classification. The fifth module detects the parking slots in the parking space by detecting the horizontal and vertical lines that bound the slots. Lastly, the sixth module detects slot occupancy by getting the percentage of white pixels in each of the slots. One of the known bugs, however, is that the fourth module fails to binarize neon green cars.

The researchers recommend that the pre-processing and binarization of the images could be improved to include the blue-green spectrum. The researchers also recommend that future developers widen the scope of this study. Future developers could look into the real-time implementation of the program in an actual parking space. This could be done by setting up cameras and programming them to capture photos at defined intervals. The image alignment algorithm could also be improved to accommodate for misalignments other than translational shifts. The algorithm for the slot and occupancy detection could also be further improved so that it could work for different kinds of parking setups.

Appendix

A1.1. parkingApp.cpp

```
      //-------------------------------------------------------------------
-----------
      //
      // Name:        parkingApp.cpp
      // Author:      Moshe Adique, Aaron Mandap, Victor Tirona
      // Created:     08/10/2019 11:55:44 PM
      // Description:
      //
      //-------------------------------------------------------------------
-----------

      #include "parkingApp.h"
      #include "parkingFrm.h"

      IMPLEMENT_APP(parkingFrmApp)

      bool parkingFrmApp::OnInit()
      {
          parkingFrm* frame = new parkingFrm(NULL);
          SetTopWindow(frame);
          frame->Show();
          return true;
      }

      int parkingFrmApp::OnExit()
      {
            return 0;
      }
```

A1.2. parkingApp.h

```
      //-------------------------------------------------------------------
-----------
      //
      // Name:        parkingApp.h
      // Author:      Moshe Adique, Aaron Mandap, Victor Tirona
      // Created:     08/10/2019 11:55:44 PM
      // Description:
      //
      //-------------------------------------------------------------------
-----------

      #ifndef __PARKINGFRMApp_h__
      #define __PARKINGFRMApp_h__

      #ifdef __BORLANDC__
            #pragma hdrstop
      #endif

      #ifndef WX_PRECOMP
            #include <wx/wx.h>
      #else
            #include <wx/wxprec.h>
      #endif

      class parkingFrmApp : public wxApp
      {
```

```
      public:
            bool OnInit();
            int OnExit();
};

#endif
```

A1.3. parkingApp.rc

```
//-----------------------------------------------------------------
-----------
//
// Name:        parkingApp.rc
// Author:      Moshe Adique, Aaron Mandap, Victor Tirona
// Created:     08/10/2019 11:55:44 PM
// Description:
//
//-----------------------------------------------------------------
-----------

#include <wx/msw/wx.rc>
```

A2.1. parkingFrm.h

```
///-----------------------------------------------------------------
--
///
/// @file      parkingFrm.h
/// @author    Moshe Adique, Aaron Mandap, Victor Tirona
/// Created:   08/10/2019 11:55:44 PM
/// @section   DESCRIPTION
///            parkingFrm class declaration
///
///-----------------------------------------------------------------
---

#ifndef __PARKINGFRM_H__
#define __PARKINGFRM_H__

#ifdef __BORLANDC__
      #pragma hdrstop
#endif

#ifndef WX_PRECOMP
      #include <wx/wx.h>
      #include <wx/frame.h>
#else
      #include <wx/wxprec.h>
#endif

//Do not add custom headers between
//Header Include Start and Header Include End.
//wxDev-C++ designer will remove them. Add custom headers after
the block.
////Header Include Start
#include <wx/msgdlg.h>
#include <wx/filedlg.h>
#include <wx/textctrl.h>
#include <wx/statbmp.h>
#include <wx/richtext/richtextctrl.h>
#include <wx/button.h>
```

```
	////Header Include End

	////Dialog Style Start
	#undef parkingFrm_STYLE
	#define parkingFrm_STYLE wxCAPTION | wxSYSTEM_MENU |
wxMINIMIZE_BOX | wxCLOSE_BOX
	////Dialog Style End

	class parkingFrm : public wxFrame
	{
		private:
			DECLARE_EVENT_TABLE();

		public:
			parkingFrm(wxWindow *parent, wxWindowID id = 1, const
wxString &title = wxT("parking"), const wxPoint& pos =
wxDefaultPosition, const wxSize& size = wxDefaultSize, long style =
parkingFrm_STYLE);
			virtual ~parkingFrm();
			void WxButton1Click(wxCommandEvent& event);
			void edt_browseBufferReset(wxRichTextEvent& event);
			void parkingFrmActivate(wxActivateEvent& event);
			void WxButton2Click(wxCommandEvent& event);
			void WxButton4Click(wxCommandEvent& event);
			void WxButton3Click(wxCommandEvent& event);
			void WxButton5Click(wxCommandEvent& event);
			void WxButton6Click(wxCommandEvent& event);
			void WxButton7Click(wxCommandEvent& event);
			void WxButton8Click(wxCommandEvent& event);
			void outputTextBufferReset(wxRichTextEvent& event);
			void WxButton5Click0(wxCommandEvent& event);
			void WxEdit1Updated(wxCommandEvent& event);
			void WxRichTextCtrl1BufferReset(wxRichTextEvent&
event);
			void WxButton6Click0(wxCommandEvent& event);


		private:
			//Do not add custom control declarations between
			//GUI Control Declaration Start and GUI Control
Declaration End.
			//wxDev-C++ will remove them. Add custom code after
the block.
			////GUI Control Declaration Start
			wxMessageDialog *WxMessageDialog1;
			wxFileDialog *dlg_browse;
			wxButton *WxButton6;
			wxTextCtrl *WxEdit1;
			wxButton *WxButton2;
			wxStaticBitmap *WxStaticBitmap1;
			wxStaticBitmap *BmpOutput;
			wxStaticBitmap *BmpInput;
			wxRichTextCtrl *edt_browse;
			wxButton *WxButton1;
			////GUI Control Declaration End
			wxImage input;
		  wxImage output;
		  wxImage refImg;
		  wxImage inpImg;
```

```
            private:
                  //Note: if you receive any error with these enum IDs,
then you need to
                  //change your old form code that are based on the
#define control IDs.
                  //#defines may replace a numeric value for the enum
names.
                  //Try copy and pasting the below block in your old
form header files.
                  enum
                  {
                        ////GUI Enum Control ID Start
                        ID_WXBUTTON6 = 1015,
                        ID_WXEDIT1 = 1014,
                        ID_WXBUTTON2 = 1006,
                        ID_WXSTATICBITMAP1 = 1005,
                        ID_BMPOUTPUT = 1004,
                        ID_BMPINPUT = 1003,
                        ID_EDT_BROWSE = 1002,
                        ID_WXBUTTON1 = 1001,
                        ////GUI Enum Control ID End
                        ID_DUMMY_VALUE_ //don't remove this value unless
you have other enum values
                  };

            private:
                  void OnClose(wxCloseEvent& event);
                  void CreateGUIControls();
      };

      #endif
```

A2.2. parkingFrm.cpp

```
      ///-----------------------------------------------------------------
--
      ///
      /// @file      parkingFrm.cpp
      /// @author    Moshe Adique, Aaron Mandap, Victor Tirona
      /// Created:   08/10/2019 11:55:44 PM
      /// @section   DESCRIPTION
      ///            parkingFrm class implementation
      ///
      ///-----------------------------------------------------------------
---

      #include "parkingFrm.h"
      #include <vector>
      #include <iostream>
      #include <sstream>
      #include <string>

      using namespace std;

      //Do not add custom headers between
      //Header Include Start and Header Include End
      //wxDev-C++ designer will remove them
      ////Header Include Start
      ////Header Include End

      //-----------------------------------------------------------------
```

```
------------
      // parkingFrm
      //---------------------------------------------------------------
------------
      //Add Custom Events only in the appropriate block.
      //Code added in other places will be removed by wxDev-C++
      ////Event Table Start
      BEGIN_EVENT_TABLE(parkingFrm,wxFrame)
            ////Manual Code Start
            ////Manual Code End

            EVT_CLOSE(parkingFrm::OnClose)
            EVT_ACTIVATE(parkingFrm::parkingFrmActivate)
            EVT_BUTTON(ID_WXBUTTON6,parkingFrm::WxButton6Click0)

            EVT_TEXT(ID_WXEDIT1,parkingFrm::WxEdit1Updated)
            EVT_BUTTON(ID_WXBUTTON2,parkingFrm::WxButton2Click)


EVT_RICHTEXT_BUFFER_RESET(ID_EDT_BROWSE,parkingFrm::edt_browseBufferRese
t)
            EVT_BUTTON(ID_WXBUTTON1,parkingFrm::WxButton1Click)
      END_EVENT_TABLE()
      ////Event Table End

      parkingFrm::parkingFrm(wxWindow *parent, wxWindowID id, const
wxString &title, const wxPoint &position, const wxSize& size, long
style)
      : wxFrame(parent, id, title, position, size, style)
      {
            CreateGUIControls();
      }

      parkingFrm::~parkingFrm()
      {
      }

      void parkingFrm::CreateGUIControls()
      {
            //Do not add custom code between
            //GUI Items Creation Start and GUI Items Creation End
            //wxDev-C++ designer will remove them.
            //Add the custom code before or after the blocks
            ////GUI Items Creation Start

            wxInitAllImageHandlers();   //Initialize graphic format
handlers

            WxMessageDialog1 =  new wxMessageDialog(this, _(""),
_("Message box"));

            dlg_browse =  new wxFileDialog(this, _("Choose a file"),
_(""), _(""), _("*.*"), wxFD_OPEN);

            WxButton6 = new wxButton(this, ID_WXBUTTON6, _("All in
One"), wxPoint(443, 639), wxSize(502, 25), 0, wxDefaultValidator,
_("WxButton6"));

            WxEdit1 = new wxTextCtrl(this, ID_WXEDIT1, _(""),
wxPoint(356, 369), wxSize(598, 256), wxTE_MULTILINE, wxDefaultValidator,
_("WxEdit1"));
```

```
            WxButton2 = new wxButton(this, ID_WXBUTTON2, _("Get Ref"),
wxPoint(44, 638), wxSize(178, 25), 0, wxDefaultValidator,
_("WxButton2"));

            WxStaticBitmap1 = new wxStaticBitmap(this,
ID_WXSTATICBITMAP1, wxNullBitmap, wxPoint(44, 359), wxSize(300, 270) );

            BmpOutput = new wxStaticBitmap(this, ID_BMPOUTPUT,
wxNullBitmap, wxPoint(356, 73), wxSize(600, 290) );

            BmpInput = new wxStaticBitmap(this, ID_BMPINPUT,
wxNullBitmap, wxPoint(44, 74), wxSize(300, 270) );

            edt_browse = new wxRichTextCtrl(this, ID_EDT_BROWSE, _(""),
wxPoint(47, 25), wxSize(830, 25), 0, wxDefaultValidator,
_("edt_browse"));
            edt_browse->SetMaxLength(0);
            edt_browse->AppendText(_("edt_browse"));
            edt_browse->SetFocus();
            edt_browse->SetInsertionPointEnd();

            WxButton1 = new wxButton(this, ID_WXBUTTON1, _("Get Input"),
wxPoint(232, 639), wxSize(203, 25), 0, wxDefaultValidator,
_("WxButton1"));

            SetTitle(_("parking"));
            SetIcon(wxNullIcon);
            SetSize(10,2,996,713);
            Center();

            ////GUI Items Creation End
      }

      void parkingFrm::OnClose(wxCloseEvent& event)
      {
            Destroy();
      }

      /*
       * WxButton1Click
       */

       void parkingFrm::WxEdit1Updated(wxCommandEvent& event)
      {
            // insert your code here
      }

      wxImage imageAlignment (wxImage refImg, wxImage inpImg)
      {
        wxImage outImg = refImg;
        int halfShift=5;
        int refImgHeight = refImg.GetHeight();
        int refImgWidth = refImg.GetWidth();
        int product = refImgHeight*refImgWidth*255;
        int minSD = 3*product*product;
        int minx;
        int miny;
        for (int x=-halfShift; x<halfShift;x++)
        {
          for (int y=-halfShift; y<halfShift;y++)
```

```
          {
            int squareDiff=0;
            int compR=0;
            int compG=0;
            int compB=0;
            for (int i=0; i<refImg.GetWidth(); i++)
            {
              for (int j=0; j<refImg.GetHeight(); j++)
              {
                if ((refImg.GetWidth()>=(i+x)) &&
(refImg.GetHeight()>=(j+y)) && ((i+x)>=0) && ((j+y)>=0))
                {
                  unsigned char r = refImg.GetRed(i,j);
                  unsigned char g = refImg.GetGreen(i,j);
                  unsigned char b = refImg.GetBlue(i,j);
                  unsigned char r2 = inpImg.GetRed(i+x,j+y);
                  unsigned char g2 = inpImg.GetGreen(i+x,j+y);
                  unsigned char b2 = inpImg.GetBlue(i+x,j+y);
                  compR=compR+(r-r2)*(r-r2);
                  compG=compG+(g-g2)*(g-g2);
                  compB=compB+(b-b2)*(b-b2);
                  squareDiff=squareDiff+compR+compG+compB;
                }
              }
            }
            if (squareDiff<minSD)
            {
              minSD=squareDiff;
              minx=x;
              miny=y;
            }
          }
        }

        for (int i=0; i<refImg.GetWidth(); i++)
        {
          for (int j=0; j<refImg.GetHeight(); j++)
          {
            unsigned char r = inpImg.GetRed(i-minx,j-miny);
            unsigned char g = inpImg.GetGreen(i-minx,j-miny);
            unsigned char b = inpImg.GetBlue(i-minx,j-miny);
            outImg.SetRGB(i,j,r,g,b);
          }
        }
        return outImg;
      }

      /*wxImage greyscale(wxImage img)
      {
        for (int i = 0; i < img.GetWidth(); i++)
        {
          for (int j = 0; j < img.GetHeight(); j++)
          {
            unsigned char r = img.GetRed(i,j);
            unsigned char g = img.GetGreen(i,j);
            unsigned char b = img.GetBlue(i,j);
            unsigned char gray = (r*299 + g*587 + b*114)/1000;
            img.SetRGB(i,j,gray,gray,gray);
          }
        }
        return img;
```

```
}*/

wxImage binarization(wxImage img)
{
  //Histogram
  double histogram[256] = {};
  unsigned char value;

  //Store histogram values
  for (int i=0; i<img.GetWidth(); i++)
  {
    for (int j=0; j<img.GetHeight(); j++)
    {
      unsigned char r = img.GetRed(i,j);
      unsigned char g = img.GetGreen(i,j);
      unsigned char b = img.GetBlue(i,j);
      unsigned char value = (r*306 + g*601 + b*117)>>10;
      histogram[value] +=1;
    }
  }

  //Sum of all frequencies times the possible threshold
  double sum = 0;
  for (int t=0 ; t<256 ; t++)
  {
    sum += t*histogram[t];
  }

  //Sum all the frequencies
  double totalFreq = 0;
  for (int t=0 ; t<256 ; t++)
  {
    totalFreq += histogram[t];
  }

  double sumB = 0;   //Sum of frequencies times possible threshold
until the background
  double bgWeight = 0; //Background weight
  double fgWeight = 0; //Background foreground

  double totalBg = 0; //Background weight
  double totalFg = 0; //Background foreground

  double maxVar = 0;
  double threshold = 0;

  for (int t=0 ; t<256 ; t++)
  {
    totalBg += histogram[t];
    if (totalBg == 0) continue;

    //Subtract background frequency from the total
    // frequency to get the foreground frequency
    totalFg = totalFreq - totalBg;

    //Sum of all frequencies times the possible
    // threshold until the current threshold
    sumB += t*histogram[t];

    double bgMean = sumB / totalBg;          // Mean Background
    double fgMean = (sum - sumB) / totalFg;  // Mean Foreground
```

```
          bgWeight = totalBg / totalFreq;
          fgWeight = totalFg / totalFreq;

          // Calculate Between Class Variance
          double bcVar = bgWeight * fgWeight *
            (bgMean - fgMean) * (bgMean - fgMean);

          // Check if new maximum found
          if (bcVar > maxVar)
          {
            maxVar = bcVar;
            threshold = t;
          }
        }

        //Set all pixels below threshold to white, otherwise set it to
black
        for (int i=0; i<img.GetWidth(); i++)
        {
          for (int j=0; j<img.GetHeight(); j++)
          {
            unsigned char value = img.GetRed(i,j);
            if(value < threshold)
            {
              img.SetRGB(i,j,0,0,0);
            }
            else
            {
              img.SetRGB(i,j,255,255,255);
            }
          }
        }

        return img;
      }//end binarization

      vector<int> slotDetection (wxImage refImg, wxImage inpImg, wxImage
&output)
      {
        inpImg = imageAlignment(refImg, inpImg);
        unsigned char value;
        vector<int> verticals;
        vector<int> horizontals;
        int refImgWidth = refImg.GetWidth();
        int refImgHeight = refImg.GetHeight();
        int verticalCondition = refImgHeight*3/4;
        int pixel = 0;
        int line = 0;
        int state = 0;

        horizontals.push_back(0);
        //binarize Image
        refImg = binarization(refImg);

        //find vertical lines
        for (int i=0; i<refImgWidth; i++)
        {
          for (int j=0; j<refImgHeight; j++)
          {
            value = refImg.GetRed(i,j);
```

```
    if (state == 0)
    {
      if (value == 255)
      {
        pixel++;
      }
    }
    else
    {
      if (value == 0)
      {
        pixel++;
      }
    }
  }
  if (pixel > verticalCondition)
  {
    line++;
  }
  if (line > 4)
  {
    verticals.push_back(i-4);
    line = 0;
    if (state == 0)
    {
      state = 1;
    }
    else if (state == 1)
    {
      state = 0;
    }
  }
  pixel = 0;
}

//find horizontal lines
line = 0;
int horizontalCondition = (verticals[2]-verticals[1])*3/4;
for (int j=0; j<refImgHeight; j++)
{
  for (int i=verticals[1]; i<verticals[2]; i++)
  {
    value = refImg.GetRed(i,j);
    if (state == 0)
    {
      if (value == 255)
      {
        pixel++;
      }
    }
    else
    {
      if (value == 0)
      {
        pixel++;
      }
    }
  }
  if (pixel > horizontalCondition)
  {
    line++;
```

```
  }
  if (line > 4)
  {
    horizontals.push_back(j-4);
    line = 0;
    if (state == 0)
    {
      state = 1;
    }
    else if (state == 1)
    {
      state = 0;
    }
  }
  pixel = 0;
}
horizontals.push_back(refImgHeight-1);

//check of individual slots
inpImg = binarization(inpImg);
vector<int> slots;
int slotNumber = (verticals.size()-2)/2;
float countW = 0;
float countB = 0;
float ratio = 0;
int topBorder, bottomBorder, leftBorder, rightBorder;
for (int a=0; a<2; a++)
{
  topBorder = horizontals[2*a];
  bottomBorder = horizontals[2*a+1];
  for (int b=1; b<=slotNumber; b++)
  {
    leftBorder = verticals[2*b-1];
    rightBorder = verticals[2*b];
    for (int i=leftBorder; i<rightBorder; i++)
    {
      for (int j=topBorder; j<bottomBorder; j++)
      {
        if(inpImg.GetRed(i,j) == 255)
        {
          countW++;
        }
        else
        {
          countB++;
        }
      }
    }
    ratio = countW/(countW+countB);
    if(ratio >= .3)
    {
      slots.push_back(1);
    }
    else
    {
      slots.push_back(0);
    }
    countW=0;
    countB=0;
    ratio = 0;
  }
```

```
  }

  for (unsigned int a=0; a<horizontals.size(); a++)
  {
    for(int b=0; b<inpImg.GetWidth(); b++)
    {
      inpImg.SetRGB(b,horizontals[a],255,0,0);
    }
  }

  for (unsigned int a=0; a<verticals.size(); a++)
  {
    for(int b=0; b<inpImg.GetHeight(); b++)
    {
      inpImg.SetRGB(verticals[a],b,255,0,0);
    }
  }
  output = inpImg;
  return slots;
}


void parkingFrm::WxButton2Click(wxCommandEvent& event)
{
      dlg_browse -> ShowModal();

  if (dlg_browse -> GetPath().IsEmpty())
  {
    return;
  }

  edt_browse -> SetValue(dlg_browse -> GetPath());
  refImg.LoadFile(dlg_browse -> GetPath(), wxBITMAP_TYPE_ANY);
  int h = refImg.GetHeight();
  int w = refImg.GetWidth();

  if (h != 270 && w != 300) {
    wxImage buff = refImg;
    WxStaticBitmap1 -> SetBitmap(buff.Scale(300, 270));
  }
  else
    WxStaticBitmap1 -> SetBitmap(refImg);
}

void parkingFrm::WxButton1Click(wxCommandEvent& event)
{
      // insert your code here
  dlg_browse -> ShowModal();

  if (dlg_browse -> GetPath().IsEmpty())
  {
    return;
  }

  edt_browse -> SetValue(dlg_browse -> GetPath());
  inpImg.LoadFile(dlg_browse -> GetPath(), wxBITMAP_TYPE_ANY);

  int h = inpImg.GetHeight();
  int w = inpImg.GetWidth();

  if (h != 270 && w != 300) {
```

```
    wxImage buff = inpImg;
    BmpInput -> SetBitmap(buff.Scale(300, 270));
  }
  else {
    BmpInput -> SetBitmap(inpImg);
  }
}



/*
 * edt_browseBufferReset
 */
void parkingFrm::edt_browseBufferReset(wxRichTextEvent& event)
{
      // insert your code here
}

/*
 * parkingFrmActivate
 */
void parkingFrm::parkingFrmActivate(wxActivateEvent& event)
{
      // insert your code here
}

/*
 * WxButton2Click
 */


/*
 * WxButton4Click
 */
void parkingFrm::WxButton4Click(wxCommandEvent& event)
{
      // insert your code here

      output = binarization(inpImg);
  if (output.GetHeight() != 400 && output.GetWidth() != 600){
    wxImage buffOut = output;
    BmpOutput -> SetBitmap(buffOut.Scale(600, 400));
  }
  else
    BmpOutput -> SetBitmap(output);
}
/*
 * WxButton3Click
 */
void parkingFrm::WxButton3Click(wxCommandEvent& event)
{
      // insert your code here
  output = imageAlignment(refImg, inpImg);
  if (output.GetHeight() != 250 && output.GetWidth() != 300){
    wxImage buffOut = output;
    BmpOutput -> SetBitmap(buffOut.Scale(300, 250));
  }
  else
    BmpOutput -> SetBitmap(output);
}
```

```
	/*
	 * WxButton5Click0
	 */
	void parkingFrm::WxButton5Click0(wxCommandEvent& event)
	{
		// insert your code here
		//output = slotDetection(inpImg, refImg);
	  if (output.GetHeight() != 290 && output.GetWidth() != 600){
	    wxImage buffOut = output;
	    BmpOutput -> SetBitmap(buffOut.Scale(600, 290));
	  }
	  else
	    BmpOutput -> SetBitmap(output);

	  WxEdit1 -> SetValue("HI");
	}

	/*
	 * WxRichTextCtrl1BufferReset
	 */
	void parkingFrm::WxRichTextCtrl1BufferReset(wxRichTextEvent&
event)
	{
		// insert your code here
	}

	/*
	 * WxEdit1Updated
	 */

	/*
	 * WxButton6Click0
	 */
	void parkingFrm::WxButton6Click0(wxCommandEvent& event)
	{
	  // insert your code here
	  vector<int> slots;
	  wxImage output;
	  slots = slotDetection(refImg, inpImg, output);

	  if (output.GetHeight() != 290 && output.GetWidth() != 600){
	    wxImage buffOut = inpImg;
	    BmpOutput -> SetBitmap(buffOut.Scale(600, 290));
	  }
	  else
	    BmpOutput -> SetBitmap(output);
	  int size = slots.size();
	  int halfSize = (size+1)/2;
	  int c, e, d, f;
	  wxString b, row, column;
	  WxEdit1 -> SetValue("");
	  for (int i=0; i<2; i++)
	  {
	    d = i+1;
	    row = wxString::Format(wxT("%d"), (int)d);
	    for (int a=0; a<halfSize; a++)
	    {
```

```
        e = i*halfSize+a;
        c = e+1;
        f = a+1;
        column = wxString::Format(wxT("%d"), (int)f);
        if (slots[e] == 0)
        {
          b = wxString::Format(wxT("%d"), (int)c);
          WxEdit1 -> AppendText("SLOT ");
          WxEdit1 -> AppendText(b);
          WxEdit1 -> AppendText(" VACANT         ROW ");
          WxEdit1 -> AppendText(row);
          WxEdit1 -> AppendText("   COLUMN ");
          WxEdit1 -> AppendText(column);
          WxEdit1 -> AppendText("\n");
        }
        else if (slots[e] == 1)
        {
          b = wxString::Format(wxT("%d"), (int)c);
          WxEdit1 -> AppendText("SLOT ");
          WxEdit1 -> AppendText(b);
          WxEdit1 -> AppendText(" OCCUPIED     ROW ");
          WxEdit1 -> AppendText(row);
          WxEdit1 -> AppendText("   COLUMN ");
          WxEdit1 -> AppendText(column);
          WxEdit1 -> AppendText("\n");
        }
      }
    }
  }
```

References

[1] “Grayscale to RGB Conversion,” *Tutorialspoint*. [Online]. Available: https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm. [Accessed: 11-Nov-2019].

[2] A. Greensted, “The Lab Book Pages,” *Sitewide RSS*. [Online]. Available: http://www.labbookpages.co.uk/software/imgProc/otsuThreshold.html. [Accessed: 09-Nov-2019].

[3] R. Szeliski, “Foundations and Trends in Computer Graphics and Vision,” *Image Alignment and Stitching: A Tutorial*. [Online]. Available: http://www.cs.toronto.edu/~kyros/courses/2530/papers/Lecture-14/Szeliski2006.pdf. [Accessed: 10-Nov-2019].

[4] J. Perez Lorenzo, R. Vazquez Martin, R. Marfil, A. Bandera and F. Sandoval (2007). Image Matching based on Curvilinear Regions, Vision Systems: Segmentation and Pattern Recognition, Goro Obinata and Ashish Dutta (Ed.), ISBN: 978-3-902613-05-9, InTech, Available from: http://www.intechopen.com/books/vision_systems_segmentation_and_pattern_recognitio n/image_matching_b ased_on_curvilinear_regions

[5] J. Karthik, S. Kaur, N. Reddy, and U. Raw, “International Journal for Research in Applied Science & Engineering Technology (IJRASET),” *Smart Parking Using Image Processing*, vol. 5, no. 10, pp. 2283–2286, Oct. 2017.

[6] R. Yusnita, F. Norbaya, and N. Basharuddin, “Intelligent Parking Space Detection System Based on Image Processing,” *International Journal of Innovation, Management and Technology*, vol. 3, no. 3, pp. 232–235, Jun. 2012.