Safety Critical 2D Pen Plotter

Arran Jones

954187

Project Dissertation submitted to Swansea University in Partial Fulfilment

for the Degree of Master of Science



Department of Computer Science

Swansea University

15th December 2020

# 

# Declaration

This work has not been previously accepted in substance for any degree and is not being con- currently submitted in candidature for any degree.

Signed Arran Jones**** (candidate)

Date 15/12/20****

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed Arran Jones**** (candidate)

Date 15/12/20****

Statement 2

I hereby give my consent for my thesis, if accepted, to be made available for photocopying and inter-library loan, and for the title and summary to be made available to outside organisations.

Signed Arran Jones**** (candidate)

Date 15/12/20****

# 

# 

# 

# Abstract

This document is a report on the progress and findings of a dissertation to implement a 2D pen plotter using safety critical design and implementation techniques. This project is split into 4 major sections. The conversion of DXF files into GCODE, the conversion of rasterized images (such as a PNG file) into GCODE, a simulator for a 2d pen plotter robot and a robust Graphical User Interface for controlling the entire system. The original plan was to create a robot out of Lego Mindstorms which would move a pen around a sheet of paper to draw the product, but unfortunately COVID-19 caused the country to go into lockdown and obtaining the Lego Mindstorms kit became problematic. It was then that a simulator would be necessary to complete the project.

This project will require rigorous testing, multiple fail-safes and emergency stop buttons for each section in order to produce a safe and reliable product. In order to ensure this is all complete on time, a strict time schedule will be required. This will be maintained by using a modified version of scrum. This will need to be modified as scrum is usually used for a team of developers, whereas this will be a solo project.

# Table of Contents

[Declaration 3](#_Toc58906759)

[Abstract 5](#_Toc58906760)

[Table of Contents 7](#_Toc58906761)

[Chapter 1 Sprint 1 - Background Research 11](#_Toc58906762)

[1.1 Objective 11](#_Toc58906763)

[1.2 GCODE 12](#_Toc58906764)

[1.3 DXF 13](#_Toc58906765)

[1.4 Rasterized Images 13](#_Toc58906766)

[1.4.1 Contours – Marching Squares 13](#_Toc58906767)

[1.4.2 Entire Image 14](#_Toc58906768)

[Chapter 2 Sprint 2 - DXF to GCODE 15](#_Toc58906769)

[2.1 Design 15](#_Toc58906770)

[2.1 Implementation 15](#_Toc58906771)

[2.1.1 Importing DXF 15](#_Toc58906772)

[2.1.2 Converting Entities to DXF 16](#_Toc58906773)

[2.1.3 Output 17](#_Toc58906774)

[2.2 Testing **Error! Bookmark not defined.**](#_Toc58906775)

[Chapter 3 Sprint 3 – Rasterized Image to GCODE 18](#_Toc58906776)

[3.1 Design 18](#_Toc58906777)

[3.1.1 The Naïve Approach (Entire Image) 19](#_Toc58906778)

[3.1.1.1 The algorithm 19](#_Toc58906779)

[3.1.1.2 Converting Lines to GCODE 19](#_Toc58906780)

[3.1.2 Marching Squares (Contours only) 19](#_Toc58906781)

[3.1.2.1 The Algorithm 19](#_Toc58906782)

[3.1.2.2 The Ambiguous Case 20](#_Toc58906783)

[3.1.2.3 Converting Contours to GCODE 20](#_Toc58906784)

[3.1.3 Output GCODE 20](#_Toc58906785)

[3.2 Implementation 20](#_Toc58906786)

[3.2.1 Importing Image and Grayscale Conversion 20](#_Toc58906787)

[3.2.2 The Naïve Approach 21](#_Toc58906788)

[3.2.3 Marching Squares 21](#_Toc58906789)

[3.2.4 Output GCODE 22](#_Toc58906790)

[3.3 Testing **Error! Bookmark not defined.**](#_Toc58906791)

[Chapter 4 Sprint 3 – Simulation Engine 23](#_Toc58906792)

[4.1 Research 23](#_Toc58906793)

[4.2 Design 23](#_Toc58906794)

[4.3 Implementation 24](#_Toc58906795)

[4.4 Testing **Error! Bookmark not defined.**](#_Toc58906796)

[Chapter 5 Sprint 4 – Integrate Systems 25](#_Toc58906797)

[5.1 Design 25](#_Toc58906798)

[5.2 Implementation 26](#_Toc58906799)

[Chapter 6 Sprint 5 – Graphical User Interface 27](#_Toc58906800)

[6.1 Design 27](#_Toc58906801)

[6.1.1 Home Page 27](#_Toc58906802)

[6.1.2 Calibration Page 27](#_Toc58906803)

[6.1.3 Custom Dialog 28](#_Toc58906804)

[6.2 Implementation 28](#_Toc58906805)

[6.2.1 Home Page 29](#_Toc58906806)

[6.2.2 Calibration Page 30](#_Toc58906807)

[6.2.3 Custom Dialog 31](#_Toc58906808)

[6.3 Interface Testing 1](#_Toc58906809)

[Chapter 7 Sprint 6 – Working Prototype 1](#_Toc58906810)

[7.1 Design 2](#_Toc58906811)

[7.2 Implementation 2](#_Toc58906812)

[7.2.1 Simulation 2](#_Toc58906813)

[7.2.2 Calibration 3](#_Toc58906814)

[7.2.3 Opening Files 3](#_Toc58906815)

[Chapter 8 Sprint 7 – System Testing 1](#_Toc58906816)

[8.1 1](#_Toc58906817)

[Chapter 9 Evaluation 1](#_Toc58906818)

[9.1 Future Development 1](#_Toc58906819)

[9.1.1 Emergency Button 1](#_Toc58906820)

[9.2.2 Marching Squares 1](#_Toc58906821)

[9.2.3 Size Preview 2](#_Toc58906822)

[Bibliography 3](#_Toc58906823)

# Chapter 1 Sprint 1 - Background Research

Before starting the project, a lot of background research is required, as there is no prior knowledge of GCODE, DXF files or safety critical systems. This chapter outlines researched topics required to gain enough understanding to begin the project and what is learnt during the research

## 1.1 Objective

The first section which requires research is safety critical systems. This is the main topic of the dissertation and there is no prior knowledge of the topic going into the project. This means that a lot of prior research is required in order to ensure the system follows a safety critical level design, implementation and testing.

Safety is a not usually talked about in conventional computer science. It is not related to reliability or security but is its own quality. Safety is the act of preventing of accidents or loss. These can come in the forms of injuries, deaths or damage to equipment.

Alongside the accidents, the main concepts to consider with safety are the risks, hazards, failures, errors and faults. The risk is a combination of probability of an accident occurring and the severity of the accident should it occur. The equation risk = p(a) ∗ s(a) is usually used as a guideline for the risk of an accident, although quantifying severity can be difficult in some unforeseen cases. Hazards are the criteria that must be fulfilled for an accident to take place. When an accident occurs, the hazard is usually very minor, but minor hazards have the ability to cause accidents of major severity. A failure is when a component of the system fails randomly due to a fault. An error is when a component fails due to a predictable fault in the system. A fault is either an error or a failure.

Safety is far more expensive to add to a system as an afterthought than it is to incorporate into the design and implementation phase during the entire development of the system. There is an eight-step guide to safety design and implementation:

1. Identify Hazards
2. Determine Risks
3. Define Safety Measures
4. Create Safe Requirements
5. Create Safe Designs
6. Implement Following Safe Designs
7. Review Previous Safety Steps
8. Test Extensively

Stages one through 3 are known as the safety analysis. These are the steps which require most forethought, as all following steps will be ensuring the chance of an accident occurring from all hazards stated in these steps are minimised. As the project is now using a simulation, instead of a robot, the hazards have already been reduced, however some will be considered during the simulation stage as if a robot were still being created.

## 1.2 GCODE

The second topic that requires research is GCODE. Prior to beginning the project, GCODE was an unfamiliar language as we had minimal knowledge of CAD. When looking through the many instructions the language consisted of, some instructions can be chosen as relevant to this project:

* G0 - Rapid Movement to specified co-ordinates. This will be useful for moving the pen when it is not meant to be drawing. This command is used in the format: G0 X1 Y1
* G1 - Linear Movement to specified co-ordinates. This will be useful for moving the pen when it is meant to be drawing. This command is used in the format: G1 X1 Y1
* G2 - Linear movement to specified co-ordinates in a clockwise arc. The arc is centred around 2 other co-ordinates. This command is used in the format: G2 X1 Y1 I1 J1
* G3 - Linear movement to specified co-ordinates in a counter-clockwise arc. The arc is centred around 2 other co-ordinates. This command is used in the format: G3 X1 Y1 I1 J1
* M0 - Program Stop
* M1 - Optional Stop
* M2 - End of Program
* F - Feed Rate, or in the case of the pen plotter, Movement Speed.

There are many more GCODE instructions, however they seem unnecessary to the project at this point in time.

In order to ensure the GCODE created is always created in an industry standard format we will be using a python library called pygcode.

## 1.3 DXF

At a basic level, DXF files are separated into features called entities. These entities define many things, such as the shape of an object, the thickness, the start and end co-ordinates, the angle at which it begins and ends, and much more.

As we are only creating a 2D pen plotter, we will be minimising the number of entities we use in order to maintain simplicity and minimise errors. The entities we will use for this project will be "LINE" which is a straight line from one co-ordinate to another, "ARC" which is a curved line, "POLYLINE" which is a combination of the 2 previously stated entities and "CIRCLE" which is a 360-degree arc defined by its centre point and radius.

In order to deal with these entities, we will be using a library called ezdxf. This will help simplify the code we produce and means that any changes in the DXF structure can be updated in the program by updating the library version.

## 1.4 Rasterized Images

In order to covert rasterized images into GCODE, the image will first need to be converted into grayscale in order to allow for a single value to govern what is considered a desired part of the image and what is not. This will be the black value of the image, a number between 0 and 1. The user will then be given the option of drawing the contours of the image, or the entire image of black value greater than the given threshold. In order to find the contours of the image, the marching squares algorithm will be used.

### 1.4.1 Contours – Marching Squares

Marching squares is an image analysis algorithm, used to determine the contours. This is done by splitting the image into small squares and finding the black value of the corner of each square, these values are then compared to the threshold provided by the user to check if the corner is within an object or not. These squares can then be categorized into 16 unique cases, defining how the contours travel through each square. Once the case of each square is determined, a form of interpolation is required to approximate the co-ordinate at which the contour crosses the boundaries of the squares. This is usually done, and will also in the case of this project, using linear interpolation. The main issue with this algorithm is the large amount of approximation and that 2 of the cases contain ambiguity as to the orientation of the contour. This would not cause any real-world problem should the program require the objects remain close but may cause issues when drawing the images.

### 1.4.2 Entire Image

In order to draw the entire image, an algorithm will be required to read the image row by row and convert the lines of pixels with a black value within the threshold of the user to line GCODE. This will probably be done with a self-defined algorithm.

# Chapter 2 Sprint 2 - DXF to GCODE

After completing the background research, the first sprint was dedicated to the conversion of a DXF file into basic GCODE instructions.

## 2.1 Design

In order to simplify this section and ensure that it can be completed during the sprint, 2 libraries will be used. These are ezdxf and pygcode. The current versions being used are pygcode 0.2.1 and ezdxf 0.13. Since completing this section, a new version of ezdxf has been released, which is causing issues with the code. Unfortunately, due to time constraints these cannot be addressed.

The first thing required for this section is to read the DXF file into python. In order to do this, we can use a function defined in ezdxf called “readfile”. This will allow us to store all entities in a variable and manipulate them as required by ezdxf.

Ezdxf also has a backend addon for matplotlib which can create a preview image of DXF files. This will be very useful later down the line when we want to give users a preview of the file chosen. As this feature is available, we can incorporate it into the file reading section.

After reading the file, the first thing that will be required is the understanding the output of the ezdxf entities, this will be done using basic print functions and then using the knowledge gained from this to extract the necessary information to create the relevant GCODE statement.

## 2.1 Implementation

### 2.1.1 Importing DXF

While reading the file was a very simple task, creating the preview image was far from simple. When first implementing this section of the project, the matplotlib backend had a bug which just making a reference to this section of code caused it to call a function that was not yet implemented. This was very frustrating as no other library was as complete as ezdxf and after searching for a long period of time, no other library was found that could produce an image from a DXF file. It was at this point it seemed futile to implement a DXF preview. Fortunately, the library is still getting regular updates and the bug was fixed before this section of the project was completed. With some minor adjustments to the code, this feature worked well. However, the image preview produced for some DXF files is a blank image. This may be fixed in future updates, however due to the time constraints there is not much that can be done about this.

### 2.1.2 Converting Entities to DXF

The conversion of the LINE entity into GCODE was very simple. The feature extraction of ezdxf allowed for the start and end co-ordinates of the line to be stored as lists. This meant that the x co-ordinate was simply the first element in the list and the y co-ordinate was the second co-ordinate. The pygcode functions GCodeRapidMove and GCodeLinearMove, which take x and y co-ordinates and convert them into the G0 and G1 GCODE statements, could then use the start and end co-ordinates to simply convert the DXF entity to GCODE.

Although the LINE entity was extremely simple, the ARC is where things started to change. As the project had now changed into a simulation rather than a robot, some changes had to be made. For simplicity, python turtles were chosen to create the simulator for the robot. Although turtles are a simple library, there are some features that meant it would not work well with conventional GCODE. The main thing that differed is that in GCODE, and arc is defined by the start point, and the centre point of the curve. Turtles however works using a start point, a radius and the angle travelled through while drawing the arc (also known as the extent of the angle) in a counter-clockwise direction. This can also be defined by: extent = end\_angle – start\_angle. In order to keep some form of industry standard feeling for this project we modified G3 to the format G03 R1 X1 Y1 S1 E1, where R is the radius of the arc, S is the start angle and E is the end angle. The start and end angles were added instead of the extent as this was how it was stored in the DXF file and this made it feel nearer to some form of industry standard. The extent can then be calculated by the simulation engine.

In order to draw CIRCLEs in turtles it is possible to set the co-ordinate to the most southern point of the circle and call the function t.circle(r). However, this did not feel like building a simulator and meant it was necessary to find a way to draw a circle in a more robotic style. This function on the other hand is the only way to create an arc. This means the function will be required, but we will not be using it with only a single parameter. This first problem to overcome was that there is no start co-ordinate stored in a DXF file, this was easily calculated by using the x co-ordinate of the centre and subtracting the radius from the centre y co-ordinate. The next issue was one that was not foreseen. Turtles measures all angles plus a multiple of 360 as the same angle, e.g. -90 and 270 are seen as the same angle. This meant that in order to draw a circle in turtles you cannot simply tell it to draw an extent of 360 as it would not draw anything. In order to overcome this, we defined a circle as two arcs of 180 extent.

After implementing the LINE and ARC entities, POLYLINEs were extremely simple as all that was required was an if statement to check each entity in the polyline to determine whether the entity is a line or an arc. It can then call the function implemented earlier to convert the entity.

### 2.1.3 Output

The final function that is required in this sprint is to output the gcode to an output file. This will allow users who understand how GCODE works to edit it if required. However, this will not be a feature implemented in the program as it could allow users with insufficient knowledge to change the GCODE and this could be problematic.

# Chapter 3 Sprint 3 – Rasterized Image to GCODE

The goal of this sprint is to covert rasterized images in a png or jpg file format to GCODE. This will be done using two methods, one using marching squares to find the contours of objects in an image and one which will draw any pixels with a black value greater than the threshold.

## 3.1 Design

For this section of the project, several image manipulation libraries will be used. These consist of skimage, matplotlib and Image. Along with these, numpy will also be required to manipulate the 2D array that is made of the image.

Like DXF files, the first thing that is necessary is to import the image into the system. Fortunately, skimage makes this extremely simple using the function io.imread(). This reads the image and outputs it as a 2D numpy array.

Following this, skimage also has a function color.rgb2gray() which will be extremely useful as the pen plotter will only be able to draw in a single colour, so the image will need to be converted to grayscale first. Although the pen plotter will only be drawing in one shade of black, so black and white may seem a simpler option than grayscale, grayscale will still be required for the linear interpolation of marching squares.

After importing the image, it is now necessary to create the lines which will end up being converted into GCODE. As stated earlier, this will be done using in two ways. The first way will be implementing the marching squares algorithm to draw the contours of an image, the second way will be a self-developed algorithm for drawing the entire image.

### 3.1.1 The Naïve Approach (Entire Image)

#### 3.1.1.1 The algorithm

This section has been named the naïve approach as it does not use any common algorithm. The algorithm used for this section is a self-defined algorithm which will scan through the image line by line and determine where the black pixels start and end on that line, converting each into a line with beginning and end co-ordinates.

#### 3.1.1.2 Converting Lines to GCODE

These can then be sequentially converted into gcode. This removes any approximation, making the algorithm slower and more computationally demanding than marching squares but increases accuracy, which is more important when drawing the entire image.

### 3.1.2 Marching Squares (Contours only)

#### 3.1.2.1 The Algorithm

Marching squares is a commonly used algorithm for finding the contours of an image. It is a relatively simple algorithm that splits the image into small squares and uses linear interpolations to make approximations on where the contours may be within each square.

Each square can be categorized as one of 16 cases depending on the black value of each corner of the square. We will define the cases as follows:

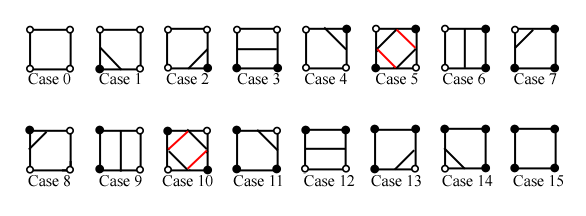


Figure 1: Marching squares cases [http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html]

The white circles on the corner of the square define a “black” pixel, or one within the object. This can also be defined as a value less than the threshold number where pure black is defined as a value zero and pure white is defined as a value of one.

As we will not be using marching squares to draw the entire image, some of these cases are identical for the purpose of this project. Therefore, we will group cases 0 and 15, 1 and 14, 2 and 13, etc.

After the cases for each square is determined, linear interpolation will be required for determining where the contour intersects the edges of the square, should a contour be found. This can be done using the equation:

Where value of corner one and corner two are the black values of the corners corresponding to the edge being intersected, the black threshold is a user defined number between 0 and 1 for how dark the pixel must be before it is considered an object and the size of edge is the length of the edge defined for each square.

#### 3.1.2.2 The Ambiguous Case

Looking again at figure 1, we can see that cases 5 and 10 have two black contours and two red contours. This is because the algorithm cannot determine whether the contours follow the black or red line. Although this seems like a big issue, according to the algorithm there is no error involved as the objects will remain closed, however it may slightly change the appearance of the image. These will be the only cases which could cause some issues further down the line. This is also the reason marching squares will only be used for finding contours, and not for drawing the entire image.

#### 3.1.2.3 Converting Contours to GCODE

As all contours are straight lines with start and end co-ordinates calculated using linear interpolation, it should be as simple as using G0 to move to one end of the contour and G1 to draw the line to the other edge.

### 3.1.3 Output GCODE

For outputting the GCODE to a file, it should be as simple as opening a file and writing the GCODE instructions to the file, as all logic was done in the previous stages.

## 3.2 Implementation

### 3.2.1 Importing Image and Grayscale Conversion

As predicted during the design phase, skimage made this stage of the image extremely simple. Both importing the image and converting it to grayscale can be done with a single line of code each, provided the file path is provided correctly.

All that was required for this step was some to convert the file location from a string into an os path, and a small amount of validation.

### 3.2.2 The Naïve Approach

When coming up with the idea for this algorithm, it seemed simple. The system would find the first black pixel in the row and save its co-ordinates. The system would then check all following pixels until it found a white pixel. Upon finding a white pixel, the system would add the co-ordinates of the previous pixel as the end co-ordinate. This worked well for some images but caused problems when the picture had black pixels in the final column as the system never found a white pixel. This caused a bug where the line inverted the image until it hit a row with a black pixel in the final column again, where it swapped back to the intended behaviour. This was fixed by adding a check to see if the black pixel was in the final column of the row. If so, its co-ordinate would be added to the list. In the case that this were to fail, to ensure one row does not affect the next a check was put in place to ensure an even number of co-ordinates is always added to the list. This would ensure there were start and end pixels to each row.

Once these bugs were resolved, the project was far simpler. As predicted, all that was required was G0 instructions to the start co-ordinates, and G1 instructions to the end co-ordinates.

### 3.2.3 Marching Squares

The first thing that was required for marching squares was to define each case, this would normally be implemented with a switch case statement. However, python does not have switch case statements. After some research, the Dictionary.get() function seemed like the perfect way to implement this, as it returns a value when given a specific key. This meant that if each case were defined by a 4-number list, they could then be referenced as a number between 0 and 15.

Once the cases had been defined, determining the state of the corners was the next task. This was extremely simple as it required checking the corners black value. Upon doing this it could return a 1 or a 0 to say whether it was black or white. The only issue with this was that splitting the image into consistently sized squares did not always ensure the corners of the squares were in range of the image. To combat this, a check was put in place to see if the corner were out of range, if so then it would change the values of the corner to the last pixel of the image before performing the check.

In order to simplify the main function of the algorithm, the linear interpolation calculation was moved to its own function. The main function does some simple checks to ensure all corners are inside the boundaries of the image, and then proceeds to check the case of the square it receives. The squares are defined by the co-ordinates of the upper left corner, as the pixels are read from the top left to the bottom right. Following these preliminary checks, the system determines the case of the square and calculates the relevant interpolants for the square. These interpolants are then added to the necessary corners co-ordinates. In hindsight this could have been split up more effectively as a couple of bugs that appeared were tough to find due to the large block of code.

Once the logic for calculating the squares was implemented, all that was left was to do was find the co-ordinates for each squares reference corner and to call the necessary functions to add the contours to a list. A check was required here to prevent the system from attempting to access co-ordinates from a blank list. This was done because cases 0 and 15 will not produce a contour but will instead return an empty list.

Once this was completed the function for converting the contours to GCODE was implemented. This was as simple as the previous section as the contours are yet again straight lines and only require the simulator to move to a start point with G0 and then draw the line using G1.

Upon completing this step, it became clear that the image was flipping upside down due to the image being read from the top left, but the co-ordinate system reads from the bottom left. This was simple to fix as numpy has a function called flipud() which can reverse the order of rows in a 2D numpy list.

After completing the implementation of marching squares, it became clear that the algorithm had 2 major flaws that after extensive research no solution was found that could be directly linked to the algorithm. The first obvious issue was the ambiguous case. For simplicity, a single direction was chosen for these cases, but this led to the closing of contours in the opposite to intended direction in several images. The second and possibly worse flaw with the algorithm was how it did not deal with edge squares that were case 0. This is a square that is completely black. In this case, the algorithm implies no contour should be drawn, which is technically true. However, this means that the contour does not stay closed and defeats the purpose of the algorithm.

### 3.2.4 Output GCODE

As predicted, the output of the GCODE was an extremely straightforward process. All that was required was to open the output file, loop through all the GCODE instructions, writing them to the file on separate lines, before closing the file.

# Chapter 4 Sprint 3 – Simulation Engine

The simulation for this project will be made using python turtles. This was originally meant to be the controller for a Lego Mindstorms robot but due to COVID-19 and the UK going into lockdown, the Lego Mindstorms kit could not be sent out. This led to the change in the project description. Having chosen turtles as the platform for implementing the simulation engine, some research was required to understand the library.

## 4.1 Research

The primary research for the project will be understanding how to move the turtle, how to change the size of the window and how to close the window. At first glance these seemed simple as there were commands to perform each of these actions. This also included a penup() and pendown() function to move rapidly and draw when necessary. The goto() function and circle() function will be the main way the turtle is moved in the simulator. These will allow for the drawing of both straight lines and arcs.

In order to close the turtles window, you can use two functions. One is exitonclick(), which allows the user to click the screen when the turtle is finished and close the window. The other is bye() which closes the window on command. We will be using both of these in the project. The exitonclick() function will provide a simple and elegant way to close the window when the simulation has finished drawing its product. The bye() function will be used to create an emergency stop button to keep with the idea of safety critical design.

## 4.2 Design

The first part of this section which will require designing is the ability to set the home location of the simulator. This is important as the system will draw the image with the initial position of the turtle as the bottom left corner. This can be done using global variables to store the home location of the turtle and the setposition() function to move the turtle to the home location before proceeding with drawing the output.

To create a large space for the simulation to draw, it would be useful to make the simulation window full screen. This would allow for larger images to be drawn and for larger scaling of output. Scaling will be a very important aspect of the simulation as many DXF files use extremely small co-ordinate systems. This means that the ability to enlarge the co-ordinates will be necessary to get a meaningful output.

In order to draw the images with the ability to scale them, simply moving to the defined co-ordinates will not be useful. This will also devalue moving the turtle to a new home location as it will just move back to draw at the specified co-ordinates. To combat this, the following equation can be used:

This equation can also be used for the initial co-ordinates for arcs, although some further calculations will be required for other values. The radius will also need to be multiplied by the scale and the extent will be:

However, as DXF files store the start and end angle relative to the x-axis and the arc is always defined as counter-clockwise, this can cause some values to be negative after subtraction. A negative extent in turtles causes the turtle to draw clockwise. This means 360 must be added to the extent if negative. Once all this has been calculated, the arc can be drawn.

The final stage to complete the simulation engine will be converting GCODE into values the simulator can use to manipulate and draw the desired output. This should be simple as all values and movement type has its own character preceding the value. All that is required is a loop through the input file, extracting all values following the recognised characters, then determining which function is required based on the whether G00, G01 or G03 is present at the start of the instruction.

## 4.3 Implementation

Setting the home location of the turtles was as simple as in the design phase. All that was required were global x\_home and y\_home variables which would be able to be changed using a set\_home() function. Creating the turtle and modifying the window was also simple as turtles uses tkinter for its User Interface, which is relatively well documented. This means that once t.screen() is stored, the window is simple to modify.

Although exitonclick() and bye() did their respective jobs correctly, there is a major inconvenience to the implementation of the two. When the screen is closed, the turtle is also deleted. This means that if the user wishes to open the screen again to draw another image it cannot be done with the same turtle. In order to get around this a resurrect\_environment() function was made to clear the screen memory and reset the screen state to running. Once this was done, a new turtle could be created and set up. Finding this fix for this inconvenience was not simple and took far longer than initially anticipated. But once this was achieved felt far more polished than having to restart the kernel after every simulation.

As predicted by the design, the rest of the implementation was simple and predictable and did not take long.

# Chapter 5 Sprint 4 – Integrate Systems

This section of the project is dedicated to integrating all of the previously implemented systems into one localised backend for the Graphical User Interface (which will be developed later) to interact with, simplifying the Graphical User Interfaces overall structure.

## 5.1 Design

The first thing this section must do is import all previous files, as each section has been saved in its own .py file. This is simple to do in python, by using the \_\_import\_\_() command. This allows for a variable to be used as a reference for the file, e.g. png = \_\_import\_\_("PNG ") would allow you to use png.import() if there is a function called import in the file PNG.py.

Once all the files are imported, the first thing that is required is to create the turtle and close it as an initial setup. This will allow the system to use resurrect\_environment() for all future openings of the turtles environment.

Once the initial setup is complete, functions will be required for opening the simulator, closing the simulator, calibrating the home location, moving the turtle to initial co-ordinates and drawing the output of the GCODE. These should all be as simple as calling the corresponding functions in the simulator file.

Having implemented all the necessary backend for the simulator, functions were now required for DXF previewing and GCODE generation. Previewing the DXF file will require the get\_dxf() function from a previous section, as this not only imports the DXF entities but also generates the preview image. Generating the GCODE will also use this function but will also call the function output\_all\_info() which also does all of the conversion and saving to a file.

As PNG and JPG files are already images, they do not require a conversion to an image for previewing. Because of this, the only thing that is required is allowing for the choice of algorithm to be a determining factor on which method is used for the GCODE instructions. This can be done using a simple if statement and a Boolean value as only 2 methods are available. After the instructions are chosen, output\_gcode() can be called from the png file to save all GCODE to the output file.

### 5.2 Implementation

As predicted in the design phase of this sprint; due to careful planning of the structure of files earlier in the project, the implementation of this sprint was extremely straightforward and quick to implement. This unfortunately did not move the schedule forward as some of the previous sections required more research than anticipated. The changes to the project specification also meant that large portions of prior research became irrelevant and new sections needed to be researched which had not been prior.

# Chapter 6 Sprint 5 – Graphical User Interface

This section of the project is dedicated to the Graphical User Interface (Graphical User Interface). This is only for the interface itself, none of the logic or connection to the backend.

## 6.1 Design

### 6.1.1 Home Page

The vast majority of this program will be accessed from one page. On this page, the user should be able to select the DXF or image file they wish to print, change the scale of the image, print the image and perform an emergency stop should it be necessary. This page will also contain a tabbed layout which will allow the user to preview the selected file and ensure that it is the intended file. The other tab of the layout should preview the GCODE generated while selecting the file. This will allow the user to search through the GCODE should an error occur to see why it is not performing as expected. This page should also contain one other button that will link to the only other page in the program, the calibration page.

Taking all of these uses into account a basic layout for this would be:

### 6.1.2 Calibration Page

Tabbed layout for image gcode and preview

Text box for scale

Scale:

Text box for file path

File Path:

Emergency Stop

Print Image

Calibrate

Open

This page is an extremely simple design, it will open along with the simulator and will only have text boxes for changing the home x and y position, along with a button to set the home and a done button to return to the previous page.

Home y value:

Home x value:

Done

Set Home

Text box

Text box

### 6.1.3 Custom Dialog

This section of the Graphical User Interface is getting called a dialog instead of a page as it will be displayed over the home page and is only opened to get information if an image is selected as the input file rather than a DXF file. This dialog will be used to determine which algorithm is desired by the user and the black threshold they desire.

Text box

Radio button for complete image

Radio button for marching squares

Black threshold:

Text box

Text box

## 6.2 Implementation

The implementation of this section was by far the most frustrating and time consuming as the research into this section was not deep enough. In order to make a clean User Interface the initial plan was to use JavaFX for the interface and use a library called Jython to run the necessary python functions. After researching JavaFX, relearning the language, and beginning to implement the interface, it was decided that testing Jython with one or two of the simpler functions of the system would be a smart idea. It was at this point floods of errors started appearing for what seemed like no reason. Even running a simple print() command from the user interface was causing issues. Upon further research it was discovered that Jython currently only supports python 2 and the whole project so far has been developed in python 3. This meant that using java was now not feasible and creating an interface with python was the only way to ensure the project was finished on time. After even more research, wxpython was chosen as the library to create the graphical user interface. This was due to its comprehensive documentation and more modern styling than tkinter. The main difference between the aesthetics of tkinter and wxpython is that tkinter has a single styling irrespective of the operating system it is run on. Wxpython on the other hand uses OS specific styling, ensuring a cleaner aesthetic.

After researching wxpython and learning the basics of the library, there was very little time to practice, so the implementation of the interface was a massive learning curve. Due to this the code is far from clean and could do with a reformat. However, not enough knowledge of the library has been gained to clean it up much more than what it is.

### 6.2.1 Home Page

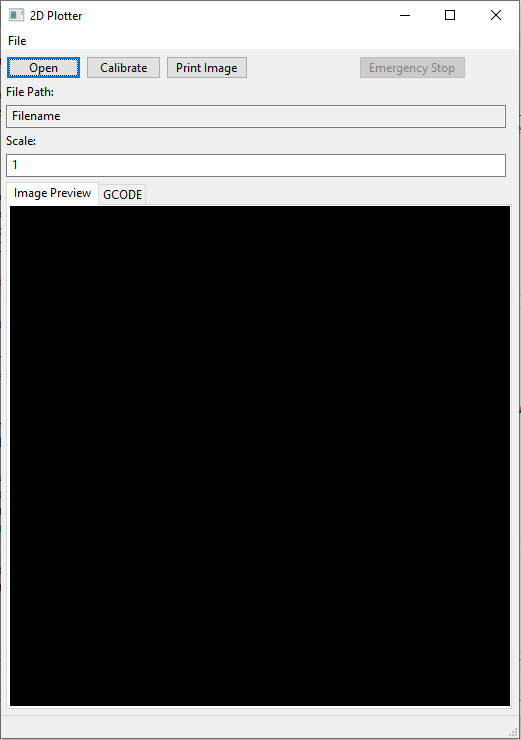
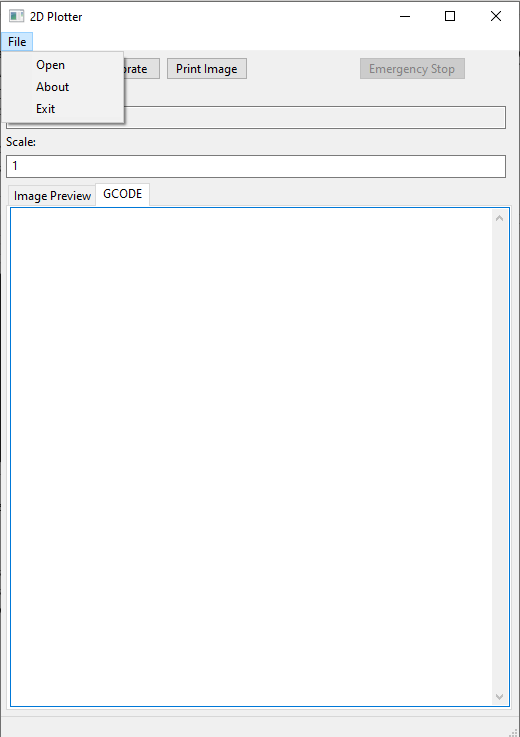
The home page took by far the most time to implement as it was this that was used to learn how the library worked. It was at this stage the decision was made to add a simple menu bar to the home page to add to the industry standard feeling this project is trying to emulate.

To clean up the code the window and panel for this page were split. This allowed the menu bar and window to be separated from the buttons, text boxes and other features.

On the window section of this page, there are only two things that are added. The first is the menu bar, which is a simple addition where menu objects are created, and the dropdown choices are appended to the object. These objects are then appended to the bar. The second feature of the window section is the sizer. This is an object which ensures a clean layout to the interface. In this case a BoxSizer is used, which merely acts as a box that any items can be placed into. The reason for this is to ensure that the items fit within the size of the window.

The panel section of this page is where the buttons, text boxes and other items are added. To maintain a tidy layout, a GridBagSizer is placed within a BoxSizer. This ensures the items remain in the size of the window, while also creating a grid layout that the items can fit neatly into. This helps organise and line up all of the items.

The toughest section of this page was adding the tabbed layout for the previews. This required an object called a notebook, which essentially is just a tabbed layout, but the way the notebook works is it essentially shows panel objects like the section created earlier. In other words, to display the image and multiline textbox, two whole new panels had to be created. The strangest part of this was that the multiline textbox would not scale to a textbox larger than one row unless if it were placed inside a sizer of some kind. This took longer to find an answer to than expected. Having completed the notebook, the home page ended up looking like:

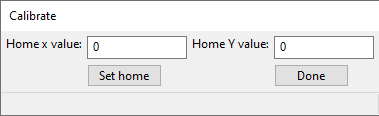
 

Which was far nearer the initial design than expected. This was mostly due to the GridBagSizer which made keeping everything tidy far simpler.

### 6.2.2 Calibration Page

Like the Home Page, the Calibration Page is also split into a window and panel. However, unlike the Home Page this page will not have a menu bar. This page will be kept incredibly simple to prevent confusion when trying to set the home location of the simulator. This means that the only use of the window is housing the panel in a BoxSizer.

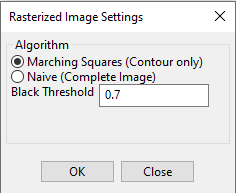
Yet again, because of the simplicity of the GridBagSizer, it was possible to get this to look incredibly similar to the design concept for this page:



### 6.2.3 Custom Dialog

Although not as large a learning curve, a decent amount of research was required to implement this as there are minor differences in the behaviour of windows and dialogs. Windows are the parent object of all objects within them, and while this is somewhat true for dialogs, once the dialog is destroyed, all of the date from the objects within them is stored in a variable in a window. This means that the window is technically the parent object of the dialog, but none of the data can be passed between the two without the destruction of the dialog. This is fine for the necessary use case in this project as the dialog is only required to get information for the black threshold and algorithm to use for the png to GCODE conversion.

As the initial designs were created with javafx in mind, it was surprising that this was still possible to achieve in a library of which no prior knowledge was known. Keeping the implementation as similar to the initial design as was achieved here really saved a decent amount of time compared to what could have been had a complete redesign been needed:



## 6.3 Interface Testing

|  |  |  |  |
| --- | --- | --- | --- |
| **Button** | **Expected Result** | **Actual Result** | **Pass/Fail** |
| About menu item | Dialog displaying message about project |  | Pass |
| Exit menu item | Closes project | Project closed | Pass |
| Calibrate button | Confirmation dialog |  | Pass |
| Yes | Calibration opens |  | Pass |
| No | Returns to Home Page |  | Pass |
| Image Preview Tab | Image display box |  | Pass |
| GCODE Tab | Multiline textbox |  | Pass |

# 

Following the testing, all interface links have performed as expected. The system has passed the interface testing.

# Chapter 7 Sprint 6 – Working Prototype

The aim of this section is to finally integrate every section of the project and achieve the first fully functional prototype before system testing and bug fixing.

## 7.1 Design

At this stage in the project there is very little left to design. The main aspects of this section will be gathering information from user inputs and calling the correlating functions in the backend of the system.

## 7.2 Implementation

### 7.2.1 Simulation

Although this may seem like an odd place to start, this was the section of the program that had caused many issues and temperamental behaviour. Due to this, it was decided that starting here would make most sense as time is starting to run short. This turned out to be the right decision as while implementing this section some extra work was needed for the simulation and the backend.

While getting the simulation to draw an image was not difficult to do and closing and reopening had already been solved by resurrecting the environment, another flaw in the system was discovered when trying to emergency stop the system. The flaw was that while the simulator was drawing, the bye() and exitonclick() functions did not interrupt the system. This prevented the emergency stop button from activating and just continued the program. In order to resolve this, an if statement was put at the start of the loop which converted and drew each instruction to check is a Boolean value called stop was set to false. If so, it was to continue as normal, otherwise it would reset the value to false and close the simulator. Having solved this issue it was now possible for the system to read a file called “output.gcode” and draw it out on screen. Once the system was finished drawing, the screen would become clickable and would close when clicked. Also, the emergency stop button on the home screen now stopped the system and closed it when pressed.

However, an error was occurring when the emergency stop button was pressed while no simulation environment was running. To combat this, when the simulator closed a Boolean value is returned to the home page and when this Boolean is true, the emergency stop button is disabled. This was not my favourite way of fixing this issue as the emergency stop button should always be available. This is due to the fact that an error in the checking system could cause it to be disabled while the simulation is running. If this were to happen and the emergency stop were required, this could be catastrophic. However, due to time constraints another method for solving this issue would have taken too long.

### 7.2.2 Calibration

Having finished implementing the simulation section of the project the next obvious section to complete was ensuring the image was drawn in the right location. This was where it became clear that the planning and effort earlier in the project made this section far simpler. All that was required from here was to get the x and y values from the textboxes in the calibration page, store them as global variables to keep for interface purposes, and input them into them into the calibrate\_sim() function in the backend. However, while implementing this section a mistake was made that caused an error which had been seen before. The environment had not been resurrected before trying to change the home co-ordinates. Upon seeing this error, the idea of resurrecting the environment and showing the user where the values they are choosing will set the bottom left corner to was thought of. This was extremely simple to implement and made the whole calibration process feel far more complete.

### 7.2.3 Opening Files

While not the hardest section of the project, this section could have done with more fore thought. Far more logic has been added into the GUI file than was necessary. A lot of the logic performed in this section of the project could be refactored into the backend given more time. However, it is functional, it could just be made tidier.

The first thing required when opening the file was a FileDialog. This is the usual file search dialog most programs use. Having created the dialog, the file extension selection was limited to .dxf, .png and .jpg. This should remove the chance of incompatible file extensions. Having selected a file, the system would then be required to determine whether an image or DXF file was chosen. If a png or jpg image was chosen, then the custom dialog would show to get the information for the chosen algorithm and black threshold. These can then be passed to the png\_generate\_gcode() function and the image can be displayed on the home page. If a DXF file is selected, then the system merely generates the DXF preview and proceeds to call the dxf\_generate\_gcode().

Overall, this section was relatively east to produce, but it took a relatively long time to implement as a lot was done at the press of the button. On saying this, having a lot of the stages such as the gcode generation and preview generation hidden from the user makes the whole system feel much more complete and fluid.

# 

# Chapter 8 Sprint 7 – System Testing

In this section we will be testing the entire system to see if there are any major errors that can be resolved.

## 8.1 Validation Testing

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Field** | **Testing Value** | **Expected result** | **Actual Result** | **Pass/Fail** |
| Threshold | a | Message Box |  | Pass |
| Threshold | 3 | Message Box |  | Pass |
| Threshold | -1 | Message Box |  | Pass |
| Threshold | 0.4 | Works successfully | Works successfully | Pass |
| Scale | a | Message Box |  | Pass |
| Scale | 3 | Works successfully | Works successfully | Pass |
| Scale | -1 | Message Box |  | Pass |
| Scale | 0.4 | Works successfully | Works successfully | Pass |
| Set home x value | a | Message Box |  | Pass |
| Set home x value | 3 | Works successfully |  | Pass |
| Set home x value | -1 | Works successfully |  | Pass |
| Set home x value | 0.4 | Message Box |  | Pass |
| Set home y value | a | Message Box |  | Pass |
| Set home y value | 3 | Works successfully |  | Pass |
| Set home y value | -1 | Works successfully |  | Pass |
| Set home y value | 0.4 | Message Box |  | Pass |

# Chapter 9 Evaluation

Having finished the project, there is far more functionality than was originally planned. This was meant to be a simple file in, drawing out style system. On the surface, this system still seems like that due to the limited number of buttons on the UI. However, there are some factors that have been added to this system, such as the scaling and multiple algorithms for rasterized image analysis that it has evolved to a far more complete system.

During this project I personally learnt a lot about several aspects of computer science I had never touched upon before. This game me a new insight into many of the differences between conventional programming and pre-planning every stage of the project. I believe scrum was a very suitable methodology for this project and seemed to work well to help me manage my time, although some unforeseen issues made it more difficult to stay on track than I would have hoped.

## 9.1 Future Development

### 9.1.1 Emergency Button

As the main topic of this project was safety critical design, the fact that the emergency stop button gets disabled is a massive flaw in the program. This is by far the most important area to be fixed in a future update.

### 9.2.2 Marching Squares

Although marching squares works as intended in this project, if this were to become a commercial product some tweaks would be required.

The first tweak would be the fix for the ambiguous case. Although this is not a difficult fix, it was not one this project needed as the marching squares algorithm was more used for output preview purposes than an actual product. However, in industry this inaccurate output could cost companies substantial money. So, fixing the algorithm would make it a much more desirable product.

The second tweak required is the fix for the black pixels at the edge of the image. Although this could be seen as intended behaviour, an option could be added to the custom menu to enable contours to be created on the last pixels of the columns and rows. This would prevent situations where the contours are not entirely closed.

The final tweak to this section of the program would be to enable squares of different sizes to be chosen. Although this has been partially implemented in the project, completing this would be desirable. The reason this was not completed was that inaccuracies were already appearing with 5x5 squares, so increasing the size of the squares would only make things worse.

### 9.2.3 Size Preview

The final feature that should be added in a future development would be one where the size of the image to be drawn can be previewed. This would be a simple addition but would make the scale section of the program far more complete. Having an extra button which could open the simulation environment and draw a box around the area that would be drawn in would make it far simpler to increase or decrease the scale for DXF files which may by far too small, or far too large.

# 

# Bibliography

|  |  |
| --- | --- |
| [1] | B. P. Douglass, “Safety Critical Systems,” 1998. [Online]. Available: https://www.omg.org/news/meetings/workshops/RT\_2002\_Workshop\_Presentations/01-3\_Douglass\_Safety\_Critical\_Systems\_Design.pdf. |
| [2] | “stackoverflow,” [Online]. Available: https://stackoverflow.com/questions/44249534/re-open-turtle-after-turtle-bye. |
| [3] | “The Marching Cubes,” 31 March 2002. [Online]. Available: http://users.polytech.unice.fr/~lingrand/MarchingCubes/algo.html. |
| [4] | “ezdxf - readthedocs,” 2020. [Online]. Available: https://ezdxf.readthedocs.io/en/stable/introduction.html. |
| [5] | fragmuffin, “GitHub,” 28 July 2017. [Online]. Available: https://github.com/fragmuffin/pygcode/wiki. |