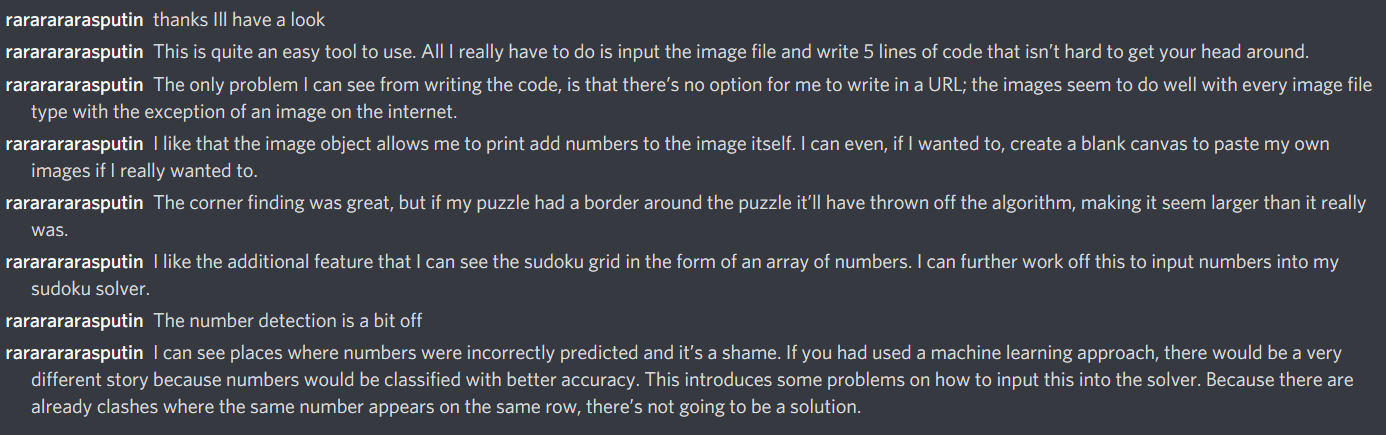
|  |  |  |
| --- | --- | --- |
| **No** | **Objective** | **Completed? How?** |
| **1** | Read Image Files (of any format) from a URL or a local path on the computer. | I completed this section by using an online library. Initially, I was planning on writing my code that would read the binary data of the image and try to decrypt it using online structures for different file types[[1]](#footnote-1). However, since the user needs to be able to input any kind of image, it was best to use pre-written code.  I used a combination of PIL (image library) and io.ByteIO (reading webpages and retrieving content) |
| **2** | The image should be compressed by abstracting image (getting rid of the noise, converting to black and white, reducing the number of pixels in the image) | I achieved this by changing the values of the pixels at different places, depending on their value. Compression perhaps could've been more elegant because it takes up the majority of the computation time. Still, nevertheless, this was achieved by passing a square over the top of the image and taking their average value.  Through playing around, I discovered the issue with shadows, which I solved with a method described in the documented design section. |
| **3** | Trace the outside of the largest quadrilateral | Achieved by using a filter that will traverse the outline. With the logic of direction of preferred travel, the filter can even navigate out of dead ends. This became possible through three classes. |
| **4** | Identify the corners of the quadrilateral | Achieved this by finding the gradients of the lines as required. This was found by finding the distances from the centre to the coordinates in the outline of the square. However, when calculating the equation of the line it introduced a source of error of ZeroDivision, where the square was correctly rotated, and the vertical lines had no change in x, so dy/dx was undefined. This was fixed by looking at dx/dy instead and then rearranging to find the equation in terms of x. |
| **5** | Rotate and cut the puzzle out of the image | Achieved by finding the average angle that the puzzle should be rotated and then by finding the new coordinates of the corners on the new and rotated image. This didn't solve the issue of the perspective warp of the image. However, this was partially solved when extracting number objects and when inserting the numbers into the grid. Further expanded below. |
| **6** | Creating a number handling object that will store the Number | Achieved. Also introduced a Suoku\_Number class which inherits Number. This was great because I could leave all Number operations to the number class, and leave the extra attribute (position) to be handled by the Sudoku\_number class |
| **7** | Create a template structure to hold template number information | Done just by screenshotting the screen with '123456789' and making sure that the numbers were sitting compactly inside the screenshot. Perhaps it could be a lousy way to extract numbers because some numbers are wider than others, which could introduce an error in classification. Maybe a solution could be to move a column through the image until it finds a black pixel, afterwards obtaining the Number and then continuing the cycle. |
| **8** | Classify numbers | Achieved by using cv2 which compares two images. One thing that I believe I did quite sloppily is that I used two different image libraries, and I had to save the image and read it with a different library. |
| **9** | Insert numbers into the image | Achieved by finding the position that the Number is being inserted into and calculating its coordinates on the actual image |
| **10** | Must be able to update the number grid after inserting new numbers | Done, just by adding new Number to the list which contained the Sudoku\_Numbers |

**Optional Targets:**

|  |  |  |
| --- | --- | --- |
| **1** | Implement a machine learning approach for classifying the numbers | Didn't add because of the added trouble of training the neural network on a dataset of numbers. This would make execution slow, and it is known that machine learning is slower than a deterministic approach. Additionally, it would heavily rely on imported functions which I would just have to string together, not making it my project. |
| **2** | Solve the sudoku puzzle | Didn't add because the accuracy of my digit recognition is mediocre, so the incorrect numbers added will affect the outcome of the puzzle's solution (given there is one) |

End-user interview: James Anderson

I asked my user to send me their opinion and improvements for my system:

**[19:58]**: thanks, I'll have a look

**[20:24]**: This is quite an easy tool to use. All I really have to do is input the image file and write 5 lines of code that isn't hard to get your head around.

**[20:26]**: The only problem I can see from writing the code, is that there's no option for me to write in a URL; the images seem to do well with every image file type with the exception of an image on the internet.

**[20:30]**: I like that the image object allows me to ~~print~~ add numbers to the image itself. I can even, if I wanted to, create a blank canvas to paste my own images if I really wanted to.

**[20:31]**: The corner finding was great, but if my puzzle had a border around the puzzle it'll have thrown off the algorithm, making it seem larger than it really was.

**[20:31]**: I like the additional feature that I can see the sudoku grid in the form of an array of numbers. I can further work off this to input numbers into my sudoku solver.

**[20:35]**: The number detection is a bit off

**[20:40]**: I can see places where numbers were incorrectly predicted and it's a shame. If you had used a machine learning approach, there would be a very different story because numbers would be classified with better accuracy. This introduces some problems on how to input this into the solver. Because there are already clashes where the same Number appears on the same row, there's not going to be a solution.

James Anderson

User Signature \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

User Feedback Analysis and overall effectiveness of my system and how it could be extended or improved:

I think that the outcome of my project is quite synonymous with the aims.

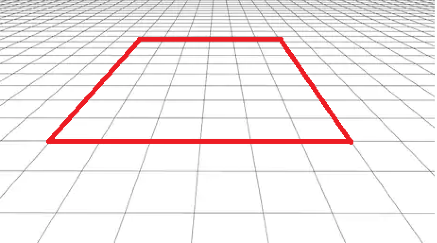
**Image Reading:** One of my user's criticisms (see above) was that they were not able to read the data of every type of image format, which included online images. However, by using an additional library, I was able to do this and achieve great results in reading from a website, thus solving one of his points of criticism (covered more in Documenting Design).

One further criticism would be that my code requires the image location to be hardcoded into the main area of my code. This could cause some complications because the user might be using a UI which will utilise the customer's camera. Taking a picture would save the image in a location which will be dynamically obtained during execution.

**Image adding numbers:** A great addition to my project was adding numbers to the puzzle. One downside was that sometimes, if the crop turned out not perfect, the numbers wouldn't sit comfortably in their square, but rather teeter on the edge or a row would be misaligned by a noticeable amount. This was fixed by adding the numbers in relation to the corners and not in relation to the dimensions of the square. This approach also managed to deal with some sort of perspective warp. This meant that if it was warped the algorithm still managed to get a reasonable estimate of the numbers in the grid. One downside that was too complicated to calculate was that the adding of numbers didn’t always work out in the y axis.

An improvement to this section would also be to have the ability to add the numbers onto the original image (the image which had not been cropped and rotated) which would just require some additional math to calculate their positions. It'll probably be quite trivial to do it off the coordinates of the image before rotation.

**Finding the outline of the puzzle**: This was my strong point, I managed to identify the square that the puzzle was in almost all the time. It even managed to find the corners of a puzzle which was quite blurred, even though the number recognition was dreadful.

The only thing that could have made this section of the project better was to fix any sort of perspective warping that had happened while taking the image.

Pictured to the left is a highly exaggerated example of what I mean. The natural skew that happened from the position of the camera meant that the corners didn't form a perfect square. Therefore some crops came out badly with slight gaps either side. My project would have been better if I had fixed this during the crop, and not when trying to calculate the corners, however this involved matrices and invariant lines that was too complicated to pursue (and also too many of my functions wouldn’t have been written by me)

One criticism of the customer was that if the puzzle were enclosed by another border (which had, e.g. 1cm gap between the puzzle), there would be cascading errors in outputs. It would end up trying to identify a number when it's an outline of a square, e.g. There's nothing that I can do about that.

**Getting Corners:** If I was to use a pre-written library, this part of the code would execute very fast, and likely not need the filter to traverse the outline. This is because if I applied a corner finding function to the image, it'd highlight everywhere there was a change in direction and therefore show more accurately the corners. My corner algorithm works very well, identifying 100% of instances where there is a main quadrilateral. So no complaints here apart from there being a secondary border around my main square.

**Rotating and Cropping:** As mentioned above, the issue of corners being effected by a perspective warp has a direct effect on rotation and cropping. If the image had also been warped to make it into a perfect square, a more accurate crop and rotation could be obtained.

**Numbers:** I was tasked with creating "a tool that programmers can incorporate into their sudoku solver algorithms to bypass the issue of having to type out numbers into a grid". I achieved this well because I managed to extract number data from the image. However, one obvious downside is the success rate of identifying images; with a rate of 80%, this tool isn't useful enough for extracting numbers from a puzzle when trying to obtain a partially solved sudoku puzzle.

Never-the-less I achieved my target of identifying at least 80% of numbers from the puzzle (and even more when the puzzle was from an online source, aka not printed out and photographed).

There's no way that I can improve my current algorithm with the current technology that I possess. I've made sure that the template and the Number are the same sizes. This shouldn't be an issue, however, and it comes down to the numbers calculated by the function which tries to match the numbers to each other. Machine learning could give me better results.

**Number Templates:** I managed to create a template class which can, if needed, work with other template images (but could be an area of concern as mentioned in the grid). An improvement on this would be to, again, use a machine learning approach which wouldn't need the number template file, but just the neural network. My user also pointed out that the matching of numbers wasn't perfect, and a machine learning approach would give more accurate results, which would make it a more appealing product to use.

If I wasn’t to use machine learning, I would have used multiple fonts. Apart from learning, an ML approach uses a huge data set to match data onto. If I had the time I would have been able to add a font checker which would check numbers against multiple number fonts to see which number has the highest likelihood of being chosen.

**General Structure:**

Some aspects are well programmed; for instance, the Number classes have attributes and functions only related to them with no added nonsense, making it also easy to reuse. I am proud that my code is very reusable. I made great use of inheritance and aggregation to make sure than my code was sufficient, concise and optimal.

One improvement that I can think of is possibly creating a coordinate class, which will house the x, y and colour value of any pixel. This can also be extended to keep opacity, different RGB values etc. However, I think this would be an extra step in the process which wouldn’t be worth making, because it would take extra computing time for something that’s already there.

1. <https://dev.exiv2.org/projects/exiv2/wiki/The_Metadata_in_PNG_files> [↑](#footnote-ref-1)