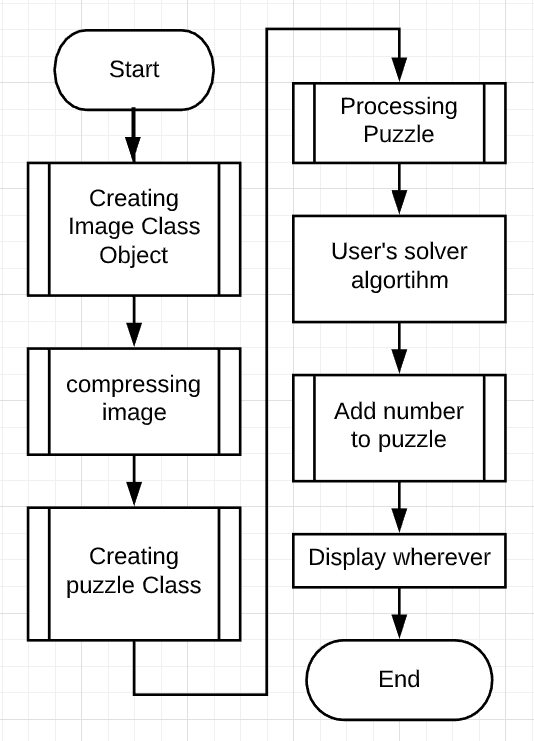
Overview:

The way the program will perform its steps depends on the way the user writes the code initially. Generally, they will create the image object, and pass it into a puzzle class, where they will call a method to process the puzzle.

1. **Creating the Image class**
   1. The image class is used to fetch image data from a source file.
   2. It is then decoded and turned to grey scale, getting rid of any colour present.

4

1. **Abstracting the image**

1

* 1. The user has the option to compress the file. It is generally recommended as it will fail otherwise; this code searches for black pixels, and doesn’t have any measures that would handle grey pixels (what constitutes grey that you want to process, and grey that you want to ignore?)

2

1. **Passing into the Puzzle class**
   1. Fetching template data

5

* + 1. If a user has a custom template data file, then they have the option to pass it into the Puzzle class. However, if they don’t it’ll go to the default template file.

3

* + 1. It’ll split the file into 9 pieces, as there will be numbers 1-9 (0 excluded because it’s not in the sudoku puzzle and will never be used)
    2. Numbers are compressed and optimised
    3. Then it appends the numbers into a dictionary, where the number templates can be referenced

1. **Processing the puzzle**
   1. Normalising image
      1. Finds corners

Calls the corner finding class to start identifying the corners

Returns a list of corners so that cropping and rotating can happen

* 1. Split into squares
     1. Goes through the image file and splits the squares into 9x9
  2. Extract number objects
     1. Deletes squares that don’t contain any numbers
     2. Inserts the image into a Number class for it to be classified
     3. Adds it to a number list
  3. Form puzzle
     1. Creates an array and passes the numbers into their original place in the puzzle

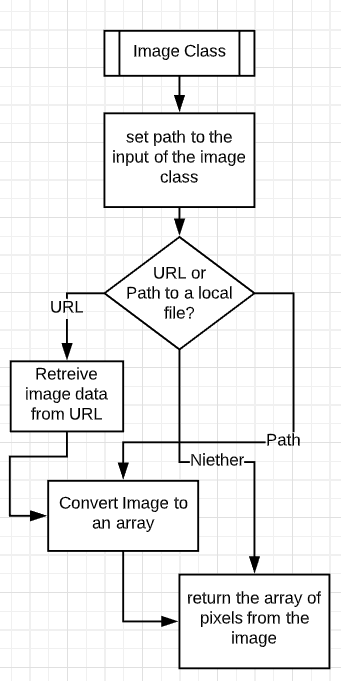
1. **Adding numbers to the puzzle**
   1. The number is extracted from the number template class and is scaled up/down to fit the dimensions of the square

**Contents**

1. Creating Image Class
   1. Reading File, URL or path to file in computer?
2. Abstracting Image
   1. Description of problems with shadow
   2. Image operations (row, column, square) (Matrix Class)
   3. Image class functions
      1. Compression
      2. Black and white pixels
      3. Exaggerate pixels
3. Pass into the puzzle class
   1. Number Template class
   2. Number class
      1. Optimise number
4. Processing Puzzle
   1. Normalising image
      1. Finding corners
      2. Trace
      3. Next centre
      4. Calculating corners
      5. Rotation and crop
   2. Split into squares
      1. If square empty
      2. Calculate number
   3. Form puzzle
5. Add number
6. Data Structures and classes
7. Object diagram

**1. Creating Image Class:**

This section is necessary for the program because it is important to be able to pass in different types of image data into the class. It will be able to handle: paths to local files, URLs to images on the internet, and lists of pixels that have already been decoded/ created manually

* 1. URL or Path to a local file?

One of my user’s complaints (see Evalutation) was that there was no way of reading files from URLs.

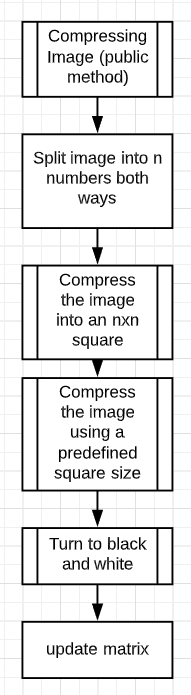
PIL[[1]](#footnote-1) is a library that allows the extraction of image data from any given data type. This is also a great choice, because PIL handles BytesIO objects.

The BytesIO[[2]](#footnote-2) requests the page from the link. All this is under the assumption that this is a link to an online resource. If this is not a link, then an exception will be raised. However, by handling this, through process of elimination, it must either be a path, or a list.

1

Therefore by trying to find the resource on the local computer through a path will give the image. If an exception occurs then it must be a list.

This list must also be normalised through numpy. This is because PIL.convert returns an image object with the array of pixels embedded within it. Therefore, normalising the array is also another way of extracting information from it.

**2. Abstracting the image**

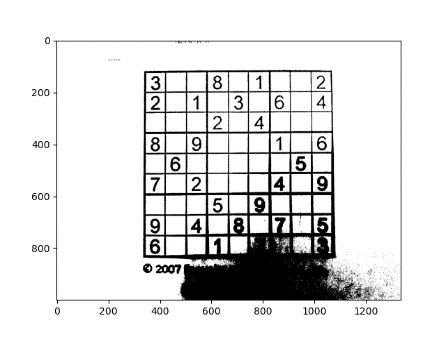
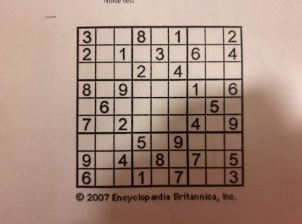
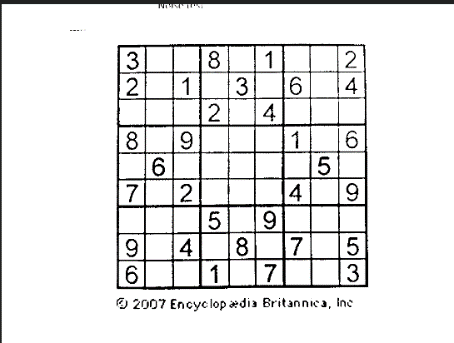
1. To avoid the problem of shadows the image must be split into an n x n square. This way, by finding the average value of that section of the image, the black and white method can dynamically alter its threshold[[3]](#footnote-3) depending on which section of the image a pixel is in.

If there was only one threshold then areas with shadows will let through more black pixels than areas without, resulting in a very noisy black and white image. This is because the average value of the image (given its majority is white with a bit of shadow over the puzzle) will be high, so therefore the threshold will let through more black pixels of the shadow than of the puzzle itself.

*(refer to the figure below)*

However, when the program crops out an area with excessive shadows the average value of the pixels will be significantly lower (black = 0, white = 255). This means that the only pixels that will go through will be those pixels which have a lower number in relation to the pixels around it.

This method allows the assessment of pixel values depending on their surroundings; if a pixel is darker in relation to other pixels around it then likelihood is that it is an important pixel to the image.

Beneficial also is that the use of the private compression function can be used in order to compress the image into an n x n grid.

(left-right) image with shadow covering part of the puzzle, the output if the section of the puzzle with the shadow wasn’t handled, the output with shadow handling

The Image class must have all properties that the image has. This means having the width, height and being able to extract pixels, rows, columns and additionally a square of data. However, these will be deduced from a 2D matrix, which has its own properties which are inherited by the image. Therefore, it makes sense for the image class to inherit a class responsible for 2D array calculations.

**Matrix functions**

To get the row:

Input parameter (y: int)

if -1 < y < height:  
 return image\_list[y]

To get the columns:

Input parameter (x: int)

if -1 < x < width:  
 column\_list = []  
 for y from (0 -> height step 1)  
 column\_list.add(get\_item(x, y))  
 return column\_list

image[y][x]

To get the square:

Testing whether the square will go off the canvas. Handles the exception by testing whether the left corner + the side length will go off the square.

Parmeters -> (left\_corner\_x, left\_corner\_y, width, height)

if 0 <= left\_corner\_x < width of canvas and 0 <= left\_corner\_y < height of canvas:

square = []

if the left\_corner\_y + height goes off the canvas:  
 bound\_y -> height of canvas  
 else  
 bound\_y -> left\_corner\_y + height

if the left\_corner\_x + width goes off the canvas:  
 bound\_x -> width of canvas  
 else  
 bound\_x = left\_corner\_x + width

for y from (left\_corner\_y -> bound\_y):  
 row = get\_row(y)[left\_corner\_x:bound\_x]   
 square.add(row)  
 return square

**Image Functions**

**Private compression function pseudocode algorithm:**

The parameters to be passed into the compression function are either manually set or a calculated from the size of the image.

Where 2000 is an abstract number that seems to work best for estimating size

Dynamic\_threshold = 16

compression\_ratio = round up((height + width) / 2000))

The parameters to this function are dimension\_X and dimension\_Y

Parmeters -> (dimension\_x, dimension\_y)

compressed\_matrix = []  
width\_list = []

squares\_x -> width of canvas / dimension\_x  
squares\_y -> height of canvas / dimension\_y  
iterations -> int(squares\_x \* squares\_y)

for i from (0 to iterations)  
 x -> int(i % squares\_x) \* dimension\_x  
 y -> int(i / squares\_x) \* dimension\_y

square -> get\_square(x, y, dimension\_x, dimension\_y)

total -> sum(square)  
 average -> round down (total / (square height \* square width))  
 width\_list.add(average)

if (i + 1) mod squares\_x == 0   
 compressed\_matrix.append(width\_list)  
 width\_list = []

return compressed\_matrix

Calculating the number of squares that will fit onto each axis

Finding the x and y coordinates of the square

Using method described above

Checking whether the squares have reached the width’s end

**Black and white function pseudocode algorithm**:

for h (from 0 to height\_of\_list)

for w (from 0 to width\_of\_list)

calculating the index of the threshold that that pixel should be compared to  
pos\_h = h / round up (pixel\_list height / split\_h)  
pos\_w = w / round up (pixel\_list width / split\_w)

pixel\_list[h][w] = exaggerate\_pixels(int(threshold\_list[pos\_h][pos\_w] / divisor),pixel\_list[h][w])

return pixel\_list

Calculating the index of the threshold that that pixel should be compared to

**Exaggerate\_pixels function pseudocode algorithm**:

The threshold is the pixel value which separates black pixels and white pixels. There is an option to either pass a single value in, or a list,

if a single integer has been passed through:

if value > threshold return 255 (white)

else return 0 (black)

If a list has been passed through

for i (from 0 to height\_of\_list)

for j (from 0 to width\_of\_list)  
 if pixel at [i][j] > threshold  
 pixel at [i][j] = 255 (white)

else  
 pixel at [i][j] = 0 (black)

**3. Passing into puzzle class**

So that any changes that are made to the image don’t affect the original image, it stays only with the puzzle instance of the image

Puzzle class:

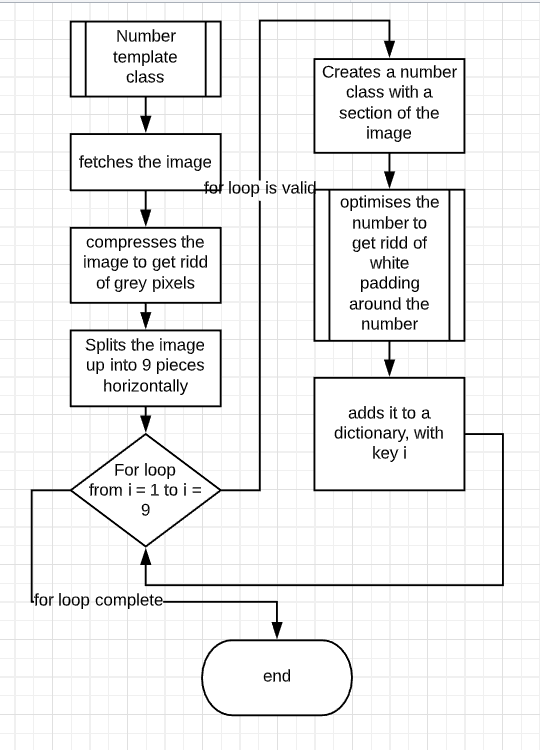
Makes a copy of the list of the compressed image

Reads the template data image

Creates a number-template class

The puzzle class requires a number template object so that it can reference the number template images

Below, the flowchart for a number template class includes functions that can get the image of any number given that it is between 1 and 9 inclusive.



This number template consists of a list of numbers. This means that the number class shouldn’t take in a number template because the number could already be known. The only place for the template would be in the calculation of the number (if it wasn’t already known)

However, the number template class should not inherit the number class because the number template consists of a list of numbers, not a number itself which has other properties. It would be wise to inherit if number template was another number with certain other characteristics, but since it’s a list of numbers, this class won’t behave as a number.

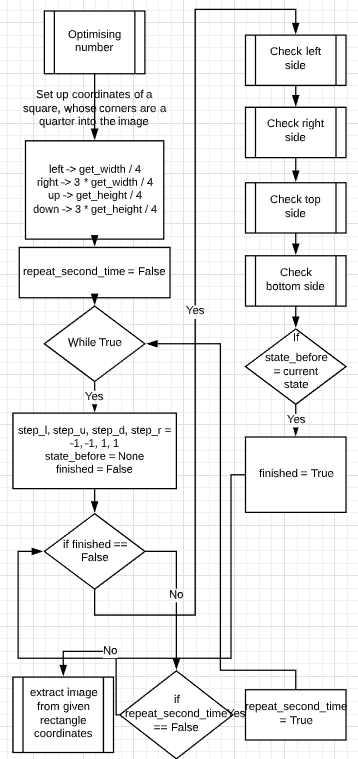
Number class constructor:

Makes a copy of the image

Turns image to black and white

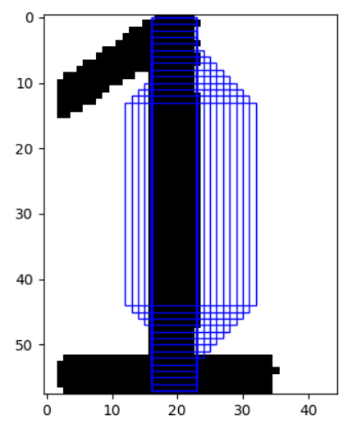
Optimising image

The functions of the number are those that are important. The main function of the number class is optimising the number and fetching the number (fetching will be covered later when that function will come into play)

1. setting the square to be a quarter into the image is sufficient, because, given that there is a number in the image that’s been passed into the function, the number will be partially inside the square.

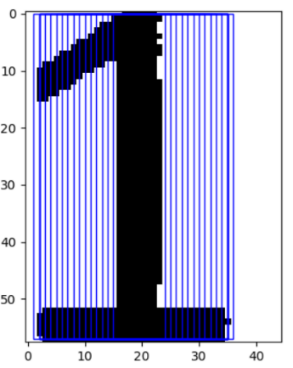
Expanding or narrowing the rectangle will allow the it to cover the entirety of the number.

The process repeats twice (notice how the while true loop terminates after two repeats) because on the chance that the rectangle stops expanding on one side, it doesn’t mean that that’s the number cropped out. (see below)



The square stopped expanding left and right because at a certain stage, it didn’t detect black pixels either side of it, so it therefore stopped expanding. By repeating the same process, but this time the corners remaining from the previous loop, it can now expand to cover the entire number

If not, update state\_before to current state



1. The check (…) side method is in charge of deciding whether the side should expand or constrict on that side. See the pseudo algorithm below:

if side is either "right" or "down"  
 initial\_direction -> 1  
else   
 initial\_direction -> -1

if side is either "right" or "left"  
 side -> get\_column(at current index of side)[bound\_lower:bound\_upper]

else  
 side -> get\_row(at current index of side)[bound\_lower:bound\_upper]

if black is not in side  
 if current\_direction != -initial\_direction:  
 current\_direction = (-1) \* current\_direction  
 current\_index = current\_index + current\_direction  
else if current\_direction != -initial\_direction  
 current\_index = current\_index + current\_direction

return current\_index, current\_direction

1

2

3

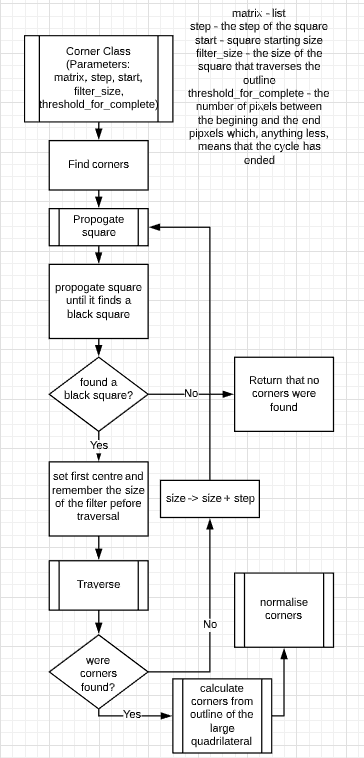
1. This dictates the initial direction, because the coordinate system of an image stems from the top left corner, and because all sides are set to expand initially, the right side (for example) will have a step of +1 in the x direction.
2. If the side is not right or left, it must be down or up. Since the right and left are vertical components of the rectangle, it is necessary to find the column, and vice versa for the up and down (row). At their current index, the row (in the case of up or down) is limited by how far the rectangle is wide, and column is limited by how far the rectangle is high; there’s no knowledge yet of what is beyond those bounds, it could (for example) be remainders of the border which shouldn’t be in the output.
3. if there is no black pixel on that side, that means that it needs to go inwards. This is because if it is expanding initially (as every side will be) and there’s no black pixel, that means that the number is inward from the side (because the number has no discontinuities). However, if the side does contain black pixels that means its safe to keep expanding.

If there are black pixels on the side AND the side is moving inwards it needs to stop moving all together; it has found the edge of the number.

**4. Processing the puzzle**

1. **Normalising image**
2. Corners are found by creating a new column object which will start trying to find the outline of the large quadrilateral.

Finding Corners

1. Find corners is a method which calls all the functions to do the dirty work of finding the corners. It is the logic of this class.
2. First it calls propagate square which starts at a particular size. It extends with a given step until it finds a black pixel, after which it returns the coordinates of the hit, as well as its size when the discovery was made:

While (size <= height and size <= width) {

hit, coordinates = apply\_square(size)

if hit return coordinates, size

otherwise size = size + step

}

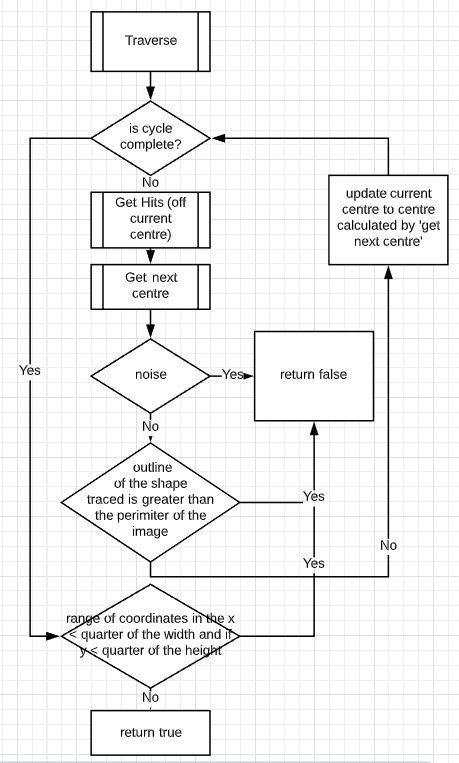
Return None, None

This incorporates the use of apply\_square function which reads the pixel values at a particular size of the square.

1. Once a black square has been found, it is necessary to traverse that square to see if it is noise, or if a full cycle can be done around it.

This section of the algorithm becomes obsolete. The point of normalising the corners was to have them in a defined order, but if the function which returns the corners returns them in the correct order, then the use of this function becomes unnecessary. However, the calculation of corners always outputs in set order through the nature of the algorithm

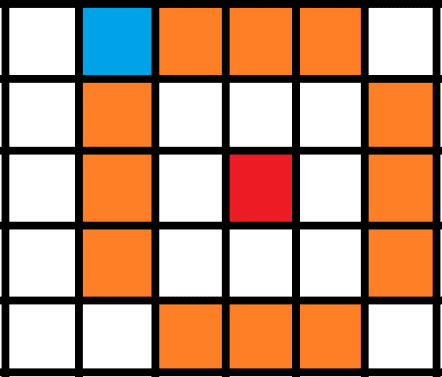
See below the algorithm for traverse and calculating corners from the outline.

Trace:

1. Get hits and get next centre will be covered in the next page
2. The purpose of checking whether the shape is greater than the perimeter of the image, is because it is an unfortunate consequence of the logic sometimes getting stuck and not being able to get out of a tail from a corner. To avoid this, by merely starting past the tail (traverse is part of a loop, see above), the algorithm will still be able to get out of the dead end. It does mean that the algorithm might get stuck often but the speed of execution makes this problem invisible to the eye.
3. If the range of x and y coordinates are less than a quarter of the width and height respectively that means that although the system found and traced the shape, it also means that the object is too small to be considered a shape, and is therefore noise.

This assumption can be made because it is another assumption that the main quadrilateral of the sudoku puzzle will take up the majority of the space of the image. Sources of such errors which classify as ‘noise’ could come from text, like titles or side paragraphs or random specs of dirt etc.

In this algorithm*, get next centre* will take, as an input, *get hits*.

Get Hits:

current = (x - gap, y - gap - 1)  
hits = []  
side = []  
change\_x = 1  
change\_y = 0

Blue = current

Orange = places where ‘current’ needs to visit

Red – current centre

This will place the current pixel on the top left of the square. (pictured to the right). It

shows a filter which is of dimensions 3x3 with the centre in red. Due to how to filter is

positioned, the gap is always going to follow (n-1)/2 (where n is the size of the filter). The blue pixel will travel clockwise

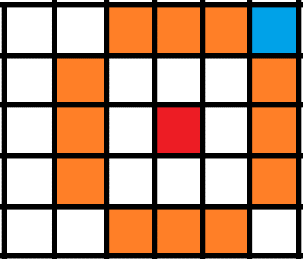
and read whether there is a black pixel anywhere along the perimeter of the filter.

for i (from 1 to 4 \*dimension + 1)  
 try:  
 if pixel at position (current\_x,current\_y) == 0  
 side.append(current)  
 except IndexError:  
 pass

This tests whether or not the filter is going out of the image. This is required because if the filter should go off the image it would return an index error because there’s no value of a pixel there. Therefore its sufficient just to skip it as it’ll be the same as there being white in that space (nothing)

if i % self.dimension == 0:  
 # then this means that the pixel needs to change direction  
 current = (current[0] + change\_x, current[1] + change\_y)  
 temp = change\_x  
 change\_x = change\_y  
 change\_y = temp  
 if i % (2 \* self.dimension) == 0:  
 change\_x = (-1) \* change\_x  
 change\_y = (-1) \* change\_y  
 # if i is divisible by dimension that means that one side has been traced  
 hits.append(side)  
 side = []

current = (current[0] + change\_x, current[1] + change\_y)

The change\_x and change\_y variables above show the current direction and future changes of the blue pixel above. Initially, the pixel is set to travel horizontally to the right (hence change\_x = 1 and change\_y = 0). It comes to no surprise that the direction of the pixel should only change when it is on an edge; when the for-loop counter has reached a multiple of the dimension of the filter. Therefore, the filter must be changed. Current is updated at the beginning of the cycle so that it sits in the physical corner of the filter, not in the edge of the orange pixels.

Then calculations of what change should happen occurs. First, the x and y exchange roles; this is because the blue pixel now must go from travelling horizontally, to travelling vertically. If, however, the blue pixel has done traversing the sides where change is positive (top and right) it must now go the other way once it reaches the bottom of the right side. It reaches the bottom when the counter ‘i’ becomes divisible by 2\*the filter size.

After this has been done, the current position needs to be updated again so that the blue pixel sits along the orange sides.

return hits

The output of this function, therefore is a list of lists, which goes through clockwise, starting at the top and ending on the left, stating the coordinate of a pixel that has a black colour value.

Get Next Centre

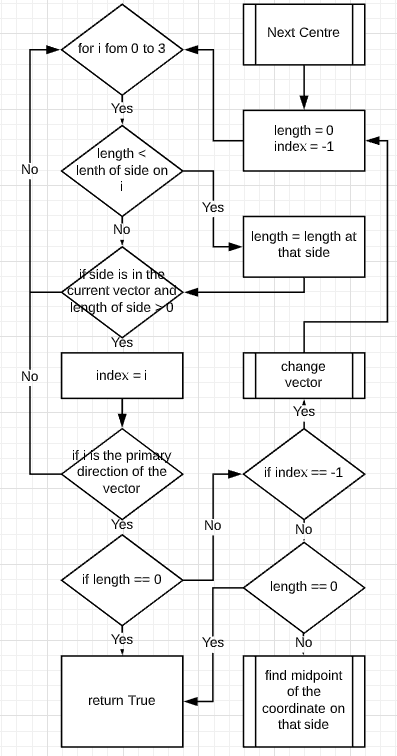
This section is made to calculate the next position of the centre along the filter.

It first needs to find the most hits present on a side, which it will then check to see if its is part of the vector.

The vector is the direction in which the filter should be moving:

1. Up
2. Right
3. Down
4. Left

It has been integrated so that the filter wont: backtrack if it has more hits on the side it came from, will keep to the edges of the quadrilateral, and will be able to get out of tails and dead ends. The vector has two states, the first, which is the primary direction (if both sides have bits it’ll prioritise the primary) and secondary (if the primary has no hits it’ll use the next best direction).

After the side with the most hits has been found, it checks whether the side is the primary direction of the vector. If, however, none of the sides contain the sides the vector wants, that means that it has reached a state where it needs to turn. Therefore, the vector is changed to accommodate the clockwise travel through the quadrilateral. The cycle is repeated again with the new vector to get a centre. When a side has been selected, the point which is on the midpoint of the hits is calculated and returned as the next centre of the quadrilateral’s trace.

Calculating corners after outline has been found

sum\_x -> 0  
sum\_y -> 0  
  
foreach coordinate in outline:  
 sum\_x -> sum\_x + coordinate(x)  
 sum\_y -> sum\_y + coordinate(y)  
  
centre\_x -> int(sum\_x / len(self.\_\_outline))  
centre\_y -> int(sum\_y / len(self.\_\_outline))  
  
distances = []  
maximum = -math.inf  
for i in range(len(self.\_\_outline)):  
 distance = (outline[i](x) - centre\_x)^2 + outline[i](y) - centre\_y)^2  
 distances.add(distance)  
 if distance > maximum:  
 maximum = distance

Calculating the centre of the quadrilateral

Calculating distance from centre to the coordinates

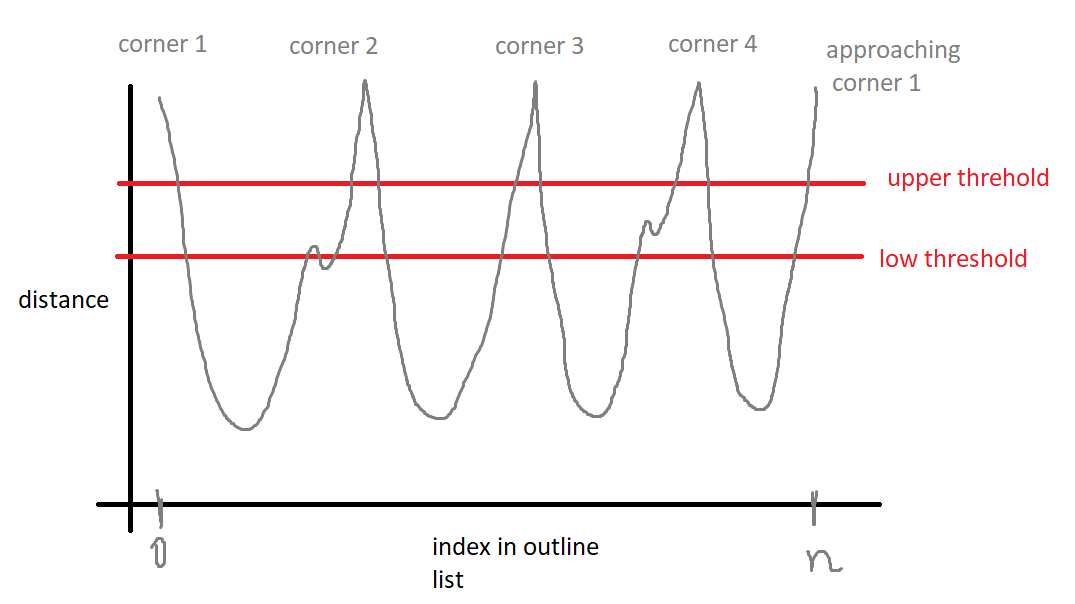
Additionally finding which coordinate is the longest distance away from the centre

lower\_threshold = 0.7 \* maximum  
upper\_threshold = 0.8 \* maximum  
  
change = []  
  
bigger = True  
threshold = lower\_threshold  
for i in range(len(distances)):  
 if distances[i] < threshold:  
 if bigger:  
 bigger = False  
 change.append(i)  
 threshold = upper\_threshold  
 else:  
 if not bigger:  
 bigger = True  
 change.append(i)  
 threshold = lower\_threshold

This separates the sides by index. First, the upper and lower thresholds are found. This means that any coordinate with a distance that is greater than them is going to be near the corner.

In this for loop, it detects the change from being close to the corner, and being along the straight line (which will be used to find the equation of the line)

The reason for having two thresholds is so that it is ‘harder’ to trigger a change once it happens. Say, for instance, the distance passes the lower threshold. It is possible, that in an extreme case, the outline may come back in towards the centre of the square, meaning that it will trigger a change. By setting the threshold to a higher number, the coordinate distance has to come out far from the corner. This concept is illustrated below:



To try to explain this better:

To avoid jitter in the decision of starting a new group two different thresholds are used. One threshold is more difficult to meet (low threshold). Another threshold is easier to meet (upper threshold). So, the decision of starting a new group is taken using harder threshold (low). Another such decision to start a new group is made easier to meet (high value). Similarly, at the end of the group the threshold is changed again to the harder threshold. Therefore, if noise exists either at the start or the end of the group such noise will not result in splitting separate points from the group. Only noise substantially larger than the two thresholds will be a problem.

Let’s consider a case, pictured above, but without using this hysteresis in threshold values. Let’s use the low threshold value. All points that are below the two red lines are points of interest, which mark the straight sections of the quadrilateral square. It is obvious that when approaching peak 2 the group will stop, index will be noted down, and then restart a few points later. It will introduce the need to analyse the length of each group to join small groups into the adjacent larger groups. Suggested hysteresis threshold (at 0.8 and 0.75) is more economical to implement.

If the length is 9 this means that the first coordinate to be tested with the above conditions was in the straight section less than the length. It can therefore be ignored as we don’t know how far down the line it is

amount = 0  
if len(change) == 9:  
 change = change[1:len(change)]  
 amount = 1  
  
  
sides = []  
for i in range(0, len(change), 2):  
 sides.add(  
 split\_array(change[(i + amount) % len(change)], change[(i + amount + 1) % len(change)], outline))  
  
equations = []  
for i in range(len(sides)):  
 equations.append(self.\_\_custom\_line\_equation(sides[i]))  
  
# calculating intersections of the 4 equations, which are the corners  
corners = []  
for i in range(4):  
 equation1 = equations[i % 4]  
 equation2 = equations[(i + 1) % 4]  
 corners.append(self.\_\_find\_intercept(equation1[0], equation1[1], equation2[0], equation2[1]))  
return corners

Amount is there to ensure that the adjacent indexes picked are those that are between two straight lines

Covered below, but finds the equation of a line

A corner is defined to be the intersection between two lines. This is just simple equating y = mx+c and wx+d

Calculating equation

sum\_x = 0  
sum\_y = 0  
sum\_xy = 0  
sum\_y\_squared = 0  
sum\_x\_squared = 0  
for coordinate in side:  
 x = coordinate[0]  
 y = coordinate[1]  
 sum\_x = sum\_x + x  
 sum\_y = sum\_y + y  
 sum\_xy = sum\_xy + (x \* y)  
 sum\_x\_squared = sum\_x\_squared + x \* x  
 sum\_y\_squared = sum\_y\_squared + y \* y  
n = len(side)  
gradient = (n \* sum\_xy - sum\_x \* sum\_y) / (n \* sum\_x\_squared - sum\_x \* sum\_x)  
y\_intercept = (sum\_y - gradient \* sum\_x) / n

if gradient \* gradient > 1:  
 gradient1 = (n \* sum\_xy - sum\_x \* sum\_y) / (n \* sum\_y\_squared - sum\_y \* sum\_y)  
 x\_intercept = (sum\_x - gradient1 \* sum\_y) / n  
 # x = x\_intercept + gradient1 \* y  
 # y = (x - x\_intercept) / gradient1  
 # gradient = 1/gradient1; y\_intercept = -x\_intercept/gradient1  
 gradient = 1 / gradient1  
 y\_intercept = -x\_intercept / gradient1  
return gradient, y\_intercept

This equation works well for lines that are NOT vertical. This is because the change in x becomes so negligible that it can no longer predict a correct approximation.

if the gradient is too steep, the algorithm can't handle no change in x.  
so by replacing x and y, it’s possible to find the equation in terms of 'y', and convert it to 'x'.

Rotating and cropping image

Its sensible to do i % 4 because the coordinates are always executed in the same format:

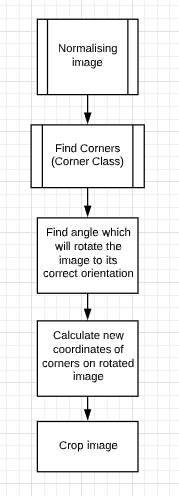
Top right, bottom right, bottom left, top left.

PIL has a rotate function which rotates around the centre. It rotates anticlockwise so the following math is done.

for i from 0 -> 3  
 coord1 = corners[i % 4]  
 coord2 = corners[(i + 1) % 4]  
 angle = math.atan2((coord2[1] - coord1[1]), (coord2[0] - coord1[0]))  
 angle = 360 \* angle / (2 \* pi)  
 total = total + angle  
  
if total >= 0  
 total = total - 180  
else if total < 0  
 total = total + 180

Because tan has a repeating period of pi. Therefore to make sure its rotated in the right direction

average = total / 4  
average\_rad = average \* math.pi / 180

Calculating the new coordinates of the corners after rotation:

foreach coordinate in corners  
 old\_coordinate\_y = coordinate.y - height / 2  
 old\_coordinate\_x = coordinate.x - width / 2

finding the coordinates of the corners as if the centre of the image (the point through which the image is rotated by) is (0,0)

radius = sqrt(old\_coordinate\_y^2 + old\_coordinate\_x^2)

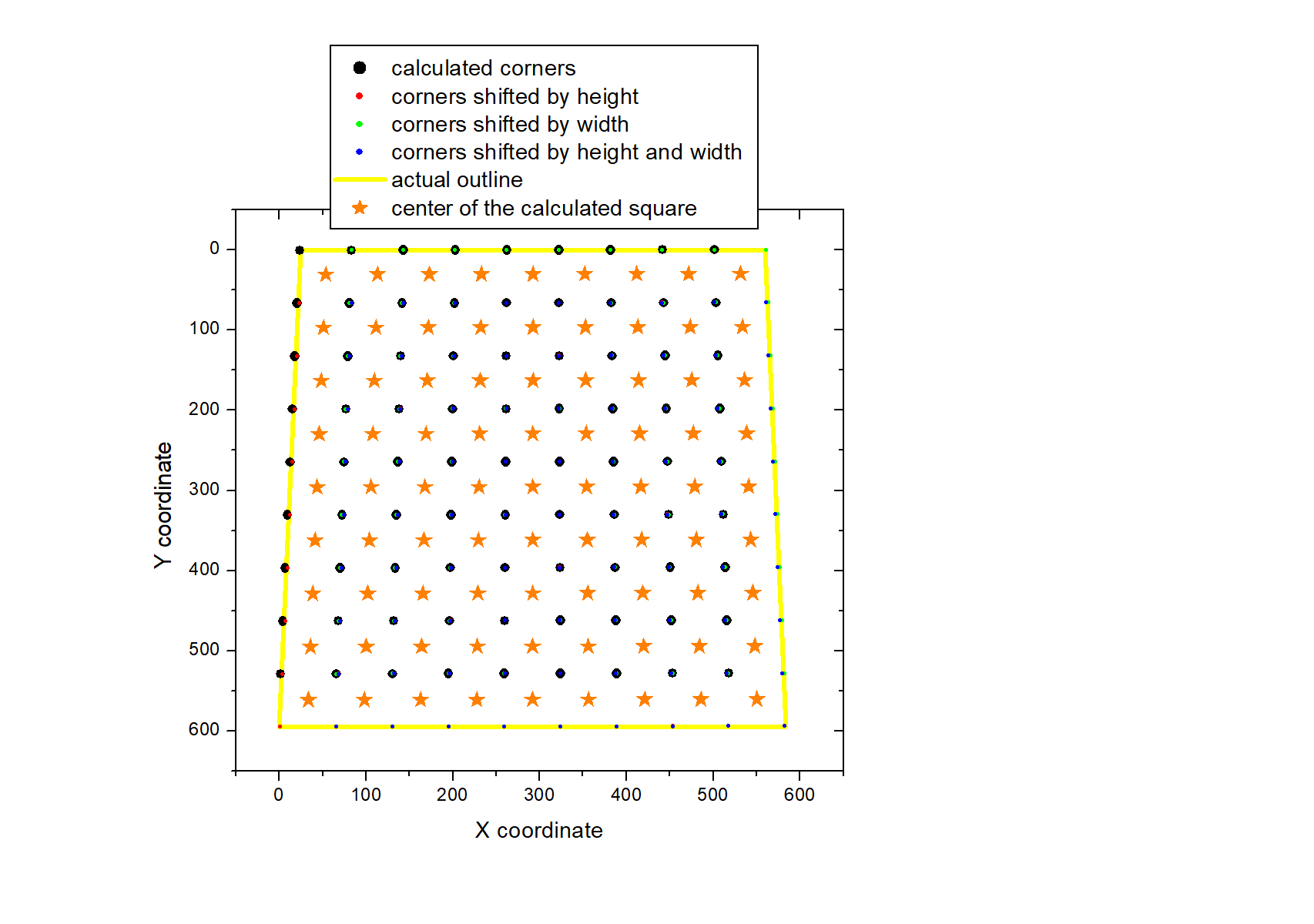
finding the distance from the relative coordinates to the centre  
 phi = math.atan2(old\_coordinate\_y, old\_coordinate\_x)  
 angle\_after = phi - average\_rad  
 new\_coordinate\_h = height / 2 + radius \* math.sin(angle\_after)  
 new\_coordinate\_w = width / 2 + radius \* math.cos(angle\_after)

Split into squares

This section of the algorithm will split the image into 81 sections,

Dimension = 9

TopX = top right corner(x) – top left corner(x)  
BotX = bottom right corner (x) – bottom left corner (x)  
LeftY = bottom left corner (y) - top left corner (y)  
RightY = bottom right corner (y) - top right corner (y)  
grid = []  
for j (from 0 -> 8):  
 row\_of\_squares = []  
 for i (from 0 -> 8):  
 width = (TopX \* (8.5 - j) + BotX \* (j + 0.5)) / 81  
 corner\_x = (top left corner (X) \* (8.5 - j) + bottom left corner (x) \* (j + 0.5)) / 9 + width \* i  
 height = (LeftY \* (8.5 - i) + RightY \* (i + 0.5)) / 81  
 corner\_y = (top left corner (y) \* (8.5 - i) + top right corner (y) \* (i + 0.5)) / 9 + height \* j  
 square = Image(image.get\_square(int(corner\_x), int(corner\_y), int(width), int(height)))  
 row\_of\_squares.add(square)  
 grid.add(row\_of\_squares)  
return grid



First term startX(i)=[p3(x)\*(8.5-j)+p2(x)\*(j+0.5)]/9 represent linear interpolation of the corner position for the left column in the sudoku image matrix. When Y coordinate (index J) is moving from 0 to 8, the corner is shifting from its startX(0) position near p3(x) to the startX(8) position near p2(x).

The width value of the squares is equal to TopX/9=[p0(x)-p3(x)]/9 for the top raw and is equal to the BotX/9=[p1(x)-p2(x)]/9 for the bottom raw.

The formula Width=[TopX\*(8.5-j)+BotX\*(j+0.5)]/81 is close to the TopX/9 for the top raw (j=0) and Botx/9 for the bottom raw (j=8)

the actual corner position is equal to the X(i,j)=startX(j)+width(j)\*i

The weight values such as (8.5-j) and (0.5+j) are introduced to appreciate that we are trying to define the optimum location for the center of the square , not its upper left corner.

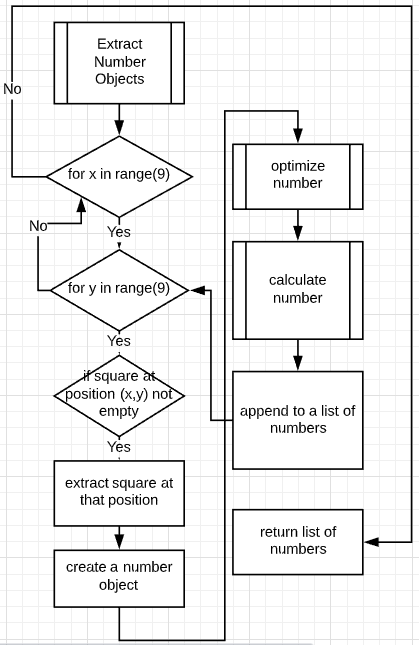
The center is shifted by "0.5" from the integer index value

the formula for linear interpolation for the Y-coordinate is very similar in its nature.

The first term startY(i)=[p3(y)\*(8.5-i)+p0(y)\*(i+0.5)]/9 represent linear interpolation of the corner position for the top raw in the sudoku image matrix. When X coordinate (index I) is moving from 0 to 8, the corner is shifting from its startY(0) position near p3( y) to the startY(8) position near p0( y)

Similarly, in the linear approximation the Height=[LeftY\*(8.5-i)+RightY\*(i+0.5)]/81 is close to the LeftY/9 for x=0 and RightY/9 for the x=Xmax.

the actual corner position is equal to the Y(i,j)=startY(i)+height(i)\*j

1. Going through each square
2. checks if the square is empty or not

does so by stepping in a quarter from both sides, and finding the average value at the centre of the square. If the average value is above a certain threshold value then it is deemed a blank square

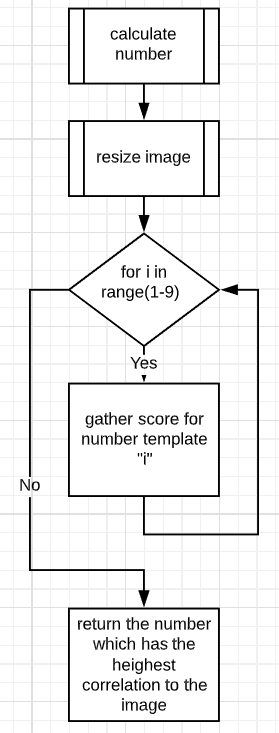
1. Deletes/ignores squares that don’t contain any numbers
2. Inserts the image into a Number class for it to be processed and classified (optimization of number is covered above)
3. Adds it to a number list

If square empty:

Inputs: border

step = int(square.height/border)  
box = Image(square.get\_square(step, step, square.width-2\*step, square.height-2\*step))  
average = sum(box.matrix) / (box.height\*box.width)  
if average > 230:  
 return True # there is no number  
else:  
 return False # there is a number in the square

Border is an integer. This function will step into the image 1/border amount of the way in and take the square that remains after removing that margin. Then, after checking the average value, it will decide whether or not the level of black pixels satisfies the condition of there being a number in there

Calculate number

The first thing to do to the number is to resize it to match the height of the template. It is done height wise and not width wise because the height of numbers stays constant. The algorithm stretches/ compresses the image so that its height is the same as the template image’s height.

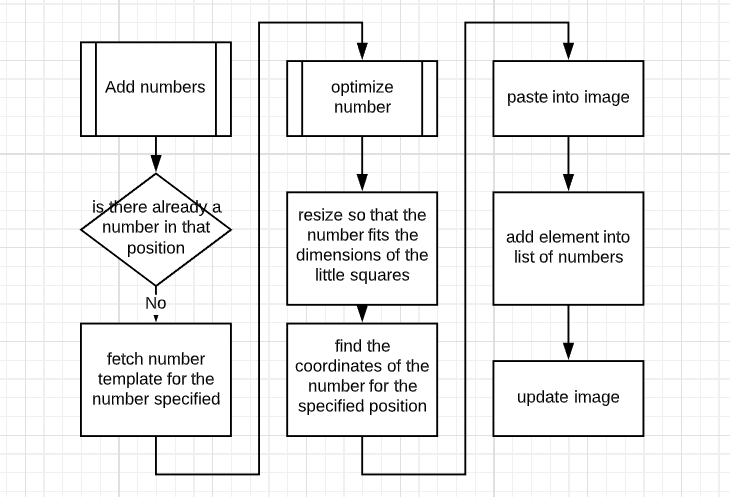
PIL is still able to achieve these results with the method resize() into the input size.

baseheight = height  
hpercent = (baseheight / image\_object.height)  
wsize = image\_object.width \* hpercent  
img = new resized image

form puzzle

return\_list = []  
for x (from 0-8)  
 row = []  
 for y (from 0-8)  
 if squares.contains\_position((y, x))  
 row.append(squares.get\_number(at position in list))  
 else  
 row.append(0)  
 return\_list.append(row)  
return return\_list

**Adding numbers to the puzzle**



The number is extracted from the number template class and is scaled up/down to fit the dimensions of the square

Where the position of the top left corner of the number is calculated by following the same procedure as in separating the squares.

TopX = top right corner(x) – top left corner(x)  
BotX = bottom right corner (x) – bottom left corner (x)  
LeftY = bottom left corner (y) - top left corner (y)  
RightY = bottom right corner (y) - top right corner (y)  
  
width = (TopX \* (8.5 - y) + BotX \* (y + 0.5)) / 81  
 corner\_x = (top left corner (X) \* (8.5 - y) + bottom left corner (x) \* (y + 0.5)) / 9 + width \* x  
 height = (LeftY \* (8.5 - x) + RightY \* (x + 0.5)) / 81  
 corner\_y = (top left corner (y) \* (8.5 - x) + top right corner (y) \* (x + 0.5)) / 9 + height \* y  
  
  
coordinates = (int(corner\_x + 0.35 \* width), int(corner\_y + 0.25 \* height))

Where *0.35 and 0.25 and 0.6* are numbers to be determined through testing; what is the best amount to step into the image by so that the number sits comfortably in the square. That is, so that it has enough padding on the sides to not look out of place and to be centred on the image.

Data Structures:

This structure will handle the two-dimensional lists. It is imperative that is has the following functions:

This structure is necessary for the ‘split into squares’ method and adding the numbers into the list.

|  |  |
| --- | --- |
| Class Number\_List() | |
| Functions: | |
| + Constructor | Creates a blank list |
| + append | Appends the number to the list |
| + get\_number | Returns the number in the list at position i |
| + get\_position | Returns position of a number in the list |
| + contains\_position | Returns whether the list contains a number at position (x,y) and returns the index of position in the list |
| + get\_length | Returns length of list |
| Parameters and Variables: | |
| - list | A private variable of list |

This structure is needed for backtracking and storing things in the order they come up in for example in finding a black pixel and tracing the outside of the quadrilateral

|  |  |
| --- | --- |
| Class Stack() | |
| Functions: | |
| + constructor | Creates a list |
| +get length | Returns length of stack |
| +get top | Returns top item of the stack without removing |
| + pop(remove=True) | Pops the top item |
| + push(object) | Pushes the item onto the top of the stack |
| + get stack | Returns stack |
| + set\_stack(new\_stack) | Updates stack |
| + get n top numbers(n) | Returns the top n numbers from the stack (if there is enough) |
| Parameters and Variables: | |
| - list | List of numbers |

|  |  |
| --- | --- |
| Class Image(Matrix) | |
| Functions: | |
| + constructor(image\_data) | Gets the 2D image array |
| - get\_2D\_array() | Returns the 2D image array given that it’s a URL or a file location |
| - compress(dimension\_h, dimension\_w) | Applies a rectangle of dimension\_h x dimension\_w and gets the average value of the square |
| - exaggerate pixels(threshold, matrix) | Turns numbers to black and white with a threshold. |
| - dynamic exaggerate pixels (pixel list, threshold\_list, split\_h, split\_w, divisor) | Refines the entire image into black and white |
| + compress( compression\_ratio, dynamic\_size, divisor) | Compresses and turns into black and white |
| + print image(save, title) | Prints image with the option to save image |
| + save(title) | Saves the image to the root directory |
| + add element(image, pos\_x, pos\_y) | Adds a picture onto a certain position on the image |
| + resize (end\_height) | Resizes the image to a given height |

|  |  |
| --- | --- |
| Class Matrix() | |
| Functions: | |
| + Constructor(input matrix) | Requires an input matrix |
| + width() | Returns the width (x axis) of the image matrix  By default the row it gets the width of is the first row. |
| + height | Returns the height (y axis) of the image matrix |
| + get item(x,y) | Returns the item at a given position |
| + get row(y) | Returns the row at a given y coordinate |
| + get column(x) | Returns the column at a given x coordinate |
| + matrix | Returns the list |
| + matrix.setter | Updates list |
| + copy | Returns an instance of the same matrix only coppied |
| + get\_square(Lcx,Lcy,w,h) | Lc\_\_ = left corner \_\_\_  Returns a square positioned over the image |
| Parameters and Variables: | |
| - matrix | The private variable which stores the matrix |

|  |  |
| --- | --- |
| Class Square\_Detection() | |
| Functions: | |
| - propogate\_square(size) | Extends a square from the top left of the image until a black pixel is detected |
| - calc\_first\_move(coordinates) | Finds the point closest to the origin as the starting point |
| - apply\_square(dimension) | With the current dimension of the large square, finds where there are hits |
| + finds corners(size) | Propagates the large square until a black pixel is found. Once found, the pixel is traversed by a filter until it reaches a conclusion of whether it has a corner. If unsuccessful repeats until it reaches the end of the image |
| - calculate corners | Form the outline of the shape, calculates the corners |
| - linear regression(side) | Finds gradient and y intercept of a line of coordinates |
| - custom\_line\_squation(side) | Is a fall back on linear regression. If the gradient calculated is undefined (a vertical line) this function picks up on the exception and handles it. |
| - Find intercept(grad1, y1, grad2, y2) | Finds the intercept between two lines |
| - split array with wrap around(a,b,input\_list) | Creates a sublist of another list. If a > b then the array is taken from a -> end and from the start -> b |
| - find in array(item, array) | Finds item in array |
| - distance(coord1, coord2) | Calculates distance between two points |
| + corners | Returns corners |
| + has corners | Returns Boolean if there are corners |
| Parameters and Variables: | |
| - image | Holds image data |
| - size | Size of the filter that will traverse the image |
| - step | What the step of the large square is |
| - backtrack | Acts as a fall back if something goes wrong, stores the size of large square before traversal |
| - outline | List of corners |
| - Threshold for complete | Integer which moderates how close a pixel has to be to the start pixel for the cycle to count as complete |
| - corners | List of corners |

|  |  |
| --- | --- |
| Class Outline | |
| Functions: | |
| + constructor( filter\_dimension, matrix) | Initialises stack of centres, filter and initial direction |
| - next centre(hits) | Calculates next centre of filter |
| - change vector | Changes vector from a given state |
| + trace (threshold) | Traverses the image until a cycle is complete or its ruled out that it’s noise |
| -is cycle complete(start, threshold) | Boolean of whether the cycle is complete |
| + set\_centre | Sets the current centre |
| + get outline | Returns outline |
| Parameters and Variables: | |
| - image | Pixel value Matrix of image |
| - threshold\_for\_complete | The number of filters away from each other 2 coordinates are for them to be considered ‘close enough for completed cycle’ |
| - outline | Stack of centres as the filter goes around the image |
| - vector | Current vector |
| - filter | Filter object |
| - start | First centre |

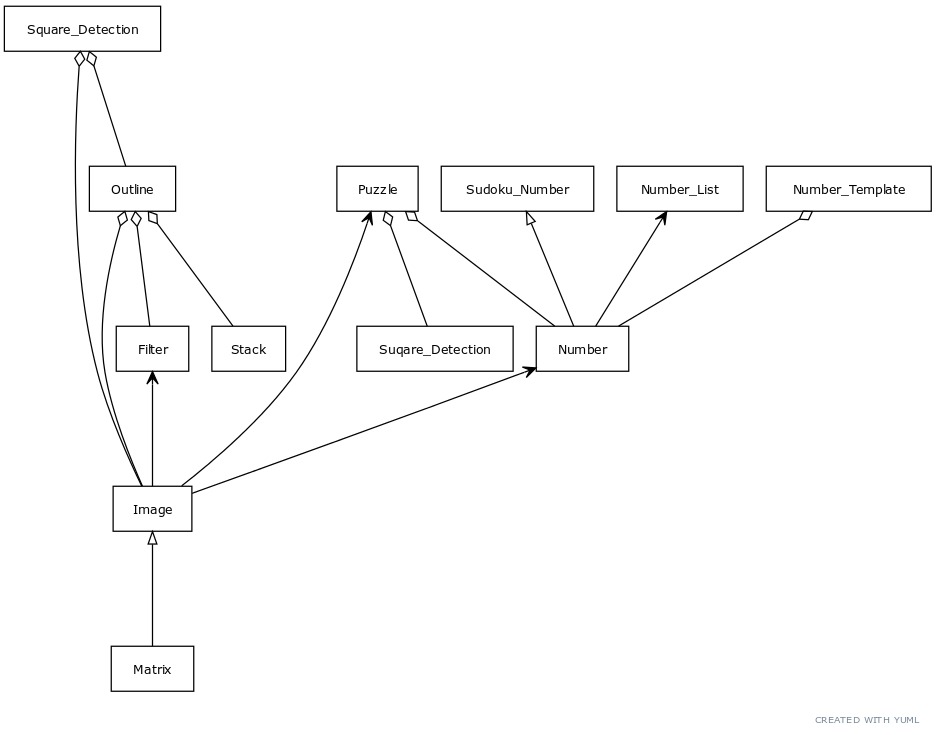
|  |  |
| --- | --- |
| Class Filter | |
| Functions: | |
| + constructor |  |
| + size | Size of the filter |
| + gap | Gap length from edge to centre |
| + centre | Centre coordinate |
| + centre.setter | Setting the centre |
| + get\_hits | Returns list of hits around the perimeter of the square/filter wherever theres a black pixel |

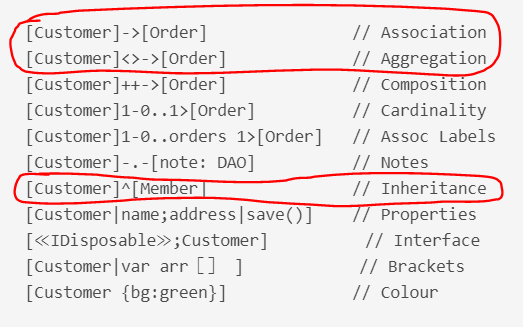
|  |  |
| --- | --- |
| Class Number\_Template | |
| Functions: | |
| +constructor (image\_obj) | Creates number list and sets up dictionary |
| - split\_into\_numbers | Splits image into 9 |
| + get\_image | Returns image for certain number |
| Parameters and Variables: | |
| - number image | Image of template |
| - number image matrix | Matrix of pixel values |
| - number list | List of number templates |
| + number dictionary | Dictionary of numbers and their image templates |

|  |  |
| --- | --- |
| Class Number | |
| Functions: | |
| + constructor(image obj, number template, number) | Obtains image data |
| - check side(side, current\_direction, current index, bound\_lower, bound upper) | Checks side of square which crops number |
| + calculate number(method) | Calculates number using templates |
| + number | Returns number |
| + image | Returns image of number |
| + Optimize num | Optimizes by cropping |
| Parameters and Variables: | |
| - number image | Image object |
| - number | int |

|  |  |
| --- | --- |
| Sudoku\_Number(Number) | |
| Functions: | |
| + constructor | Inherits from Number and initialises position |
| + position | Returns position of number in sudoku puzzle |
| Parameters and Variables: | |
| - Position | (x, y) in relation to sudoku puzzle |

|  |  |
| --- | --- |
| Class Puzzle | |
| Functions: | |
| + constructor (image, template\_path) | Sets up number templates and image object |
| - split\_into\_squares(dimension 9 ) | Splits puzzle into a 9x9 array of squares |
| - is empty | Calculates whether a square is empty |
| - extract number objects(border) | Tests if the square contains a number and if so classifies it and adds it to a list of numbers |
| - form puzzle | Forms integer array from number list |
| + puzzle | Returns form puzzle matrix |
| + find\_corners(corners) | Finds corners of the puzzle |
| + normalise\_image(corners) | Rotates and crops image |
| + image | Returns image object |
| + process\_puzzle | Rotates, splits into squares, extracts number objects and forms puzzle |
| Corners | Returns a list of corners |
| + add number(number, position) | Adds number to the image |
| Parameters and Variables: | |
| - puzzle image | Image object |
| - number\_template | Number template class |
| - grid | Image list |
| - numbers | List of numbers |
| - puzzle | List of integers relative to their position in the puzzle |





1. <https://pillow.readthedocs.io/en/stable/> [↑](#footnote-ref-1)
2. <https://docs.python.org/2/library/io.html> [↑](#footnote-ref-2)
3. Where threshold relates to the value beyond which pixels will take the value 255 (white) and below which will take the value 0 (black) [↑](#footnote-ref-3)