Sudoku Solver

The objective of this project is, given a picture of a sudoku from https://www.sudoku-online.org/, to solve it.

To do this we separate the process in three steps.

We first get the individual pictures of each square, we then identify the number in each square to build a vector which represents the sudoku. At last, we use a backtracking algorithm to solve the sudoku using this vector.

Frist step: Individual Images

https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123

We first use canny edge detection to identify sudoku board borders.

This algorithm consists of 5 steps:

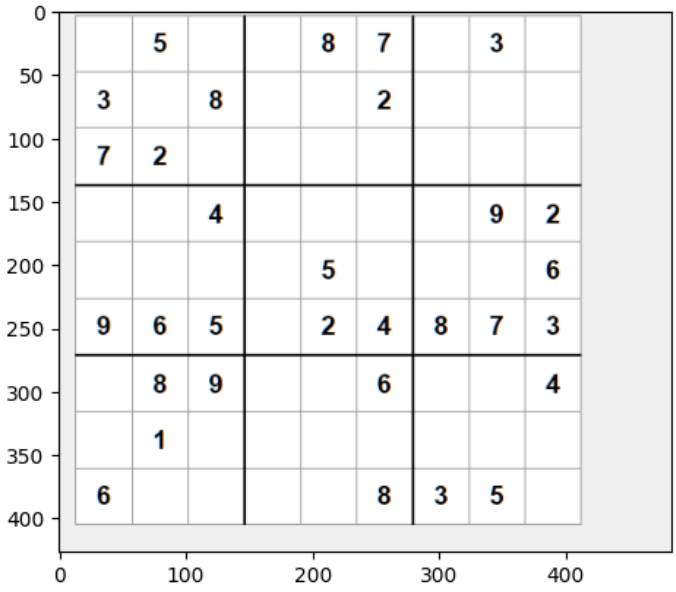
Noise reduction; Remove outliers for easier gradient calculations

Gradient calculation: Use Sobel filters to detect change in intensity of pixels

Non-maximum suppression: Make all edges the same value based on near pixels

Double threshold: Apply two thresholds to identify strong pixels and weak pixels

Edge Tracking by Hysteresis. Transform weak pixels near strong pixels into strong ones



Original Picture

```python
def DetectEdge(path,thr1,thr2):
    "Apllies CannyEdgeDetection and returns original image and coo
    img = cv.imread(path)
    img_gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    #Use Canny Edge Detection Algorithm
    edged = cv.Canny(img_gray, thr1, thr2)
    #get contours from the edged image
    contours, _ = cv.findContours(edged, cv.RETR_EXTERNAL,
                                  cv.CHAIN_APPROX_SIMPLE)
    return img,contours
```

Open CV function applies the algorithm in one simple step. We can provide the thresholds for the fourth step, the double threshold one.

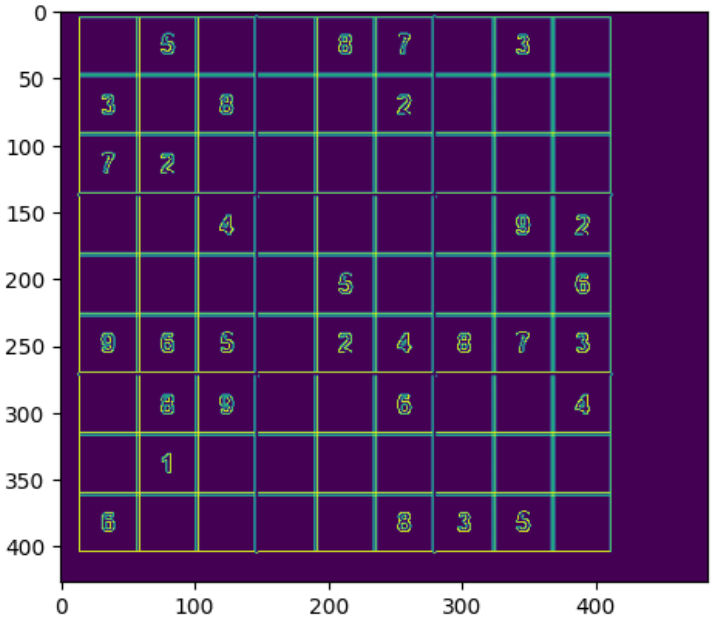


Image after canny edge detection algorithm

We then look for the contours of the board.

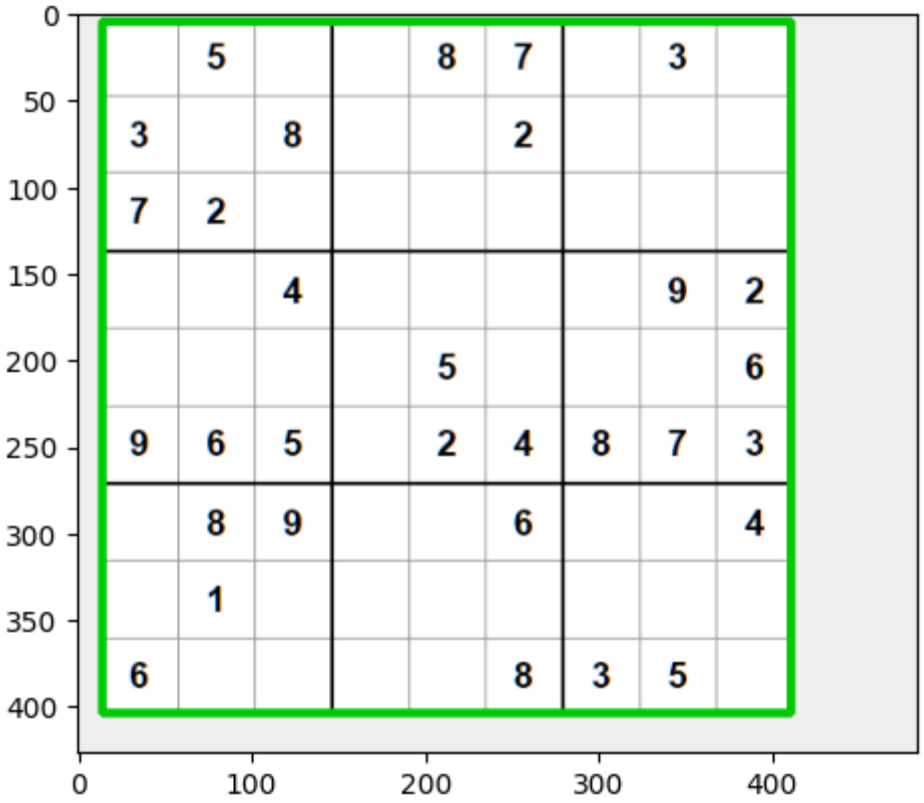
Open CV has a function that does this. It detects the change in color in a point in an image. When there are a lot of points forming a line with the same color and intensity, then we found a contour.

```
w, h = img.shape[1], img.shape[0]
for cntr in contours:
    imgx, imgy, imgw, imgh = cv.boundingRect(cntr)
    if imgw < w/5 or imgw < h/5 or imgw/imgh < 0.25 or imgw/imgh > 1.5:
        continue
  # Approximate the contour with 4 points
    peri = cv.arcLength(cntr, True)
    frm = cv.approxPolyDP(cntr, 0.1*peri, True)
    if len(frm) != 4:
        continue

  # Converted image should fit into the original size
    board_size = max(imgw, imgh)
    if imgx + board_size >= w or imgy + board_size >= h:
        continue
  # Points should not be too close to each other
  # (use euclidian distance)
    if cv.norm(frm[0][0] - frm[1][0], cv.NORM_L2) < 0.1*peri or \
        cv.norm(frm[2][0] - frm[1][0], cv.NORM_L2) < 0.1*peri or \
        cv.norm(frm[3][0] - frm[1][0], cv.NORM_L2) < 0.1*peri or \
        cv.norm(frm[3][0] - frm[2][0], cv.NORM_L2) < 0.1*peri:
        continue
```

We iterate for every contour point until we have 4 points.

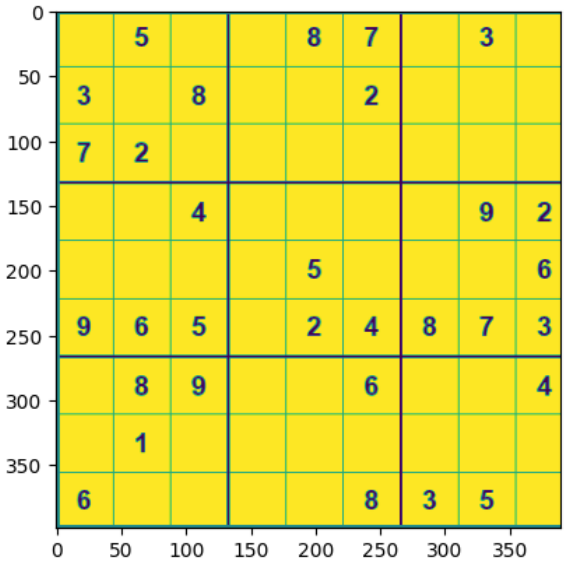
We calculate ecludiana distance between each contour point, as we need points that are far away (the 4 most far away points in this case). We measure the distance of the contour with arclength and get the approximation of the shape based on the contours with arclength.



The Sudoku board contour

We crop the image based on the 4 points that compose the edges of the board.

```
croped_image = img_out[frm[0][0][1]:frm[1][0][1],frm[1][0][0]:frm[2][0][1]]
croped_image = cv.cvtColor(croped_image, cv.COLOR_BGR2GRAY)
```

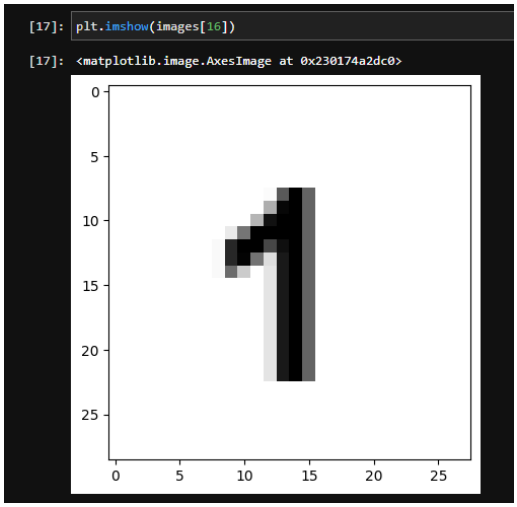


Cropped image

We know have the sudoku board. We know there are 81 squares of the same size, so we iterate over the board and crop the individual squares.

```
images = []
cell_w, cell_h = imagenprueba.shape[1]//9,imagenprueba.shape[0]//9
for x in range(9):
    for y in range(9):
        x1, y1 = x*cell_w, y*cell_h
        x2, y2 = (x + 1)*cell_w, (y + 1)*cell_h
        cx, cy = (x1 + x2)//2, (y1 + y2)//2
        w2, h2 = cell_w, cell_h
        crop = imagenprueba[y1+10:y2-5, x1+10:x2-5]
        images.append(np.concatenate((np.expand_dims(crop, axis = 2),np.expand_dims(crop, axis = 2),np.expand_dims(crop, axis = 2)),axis = 2))
```

Note the sum and subtract of pixels to each individual square. This is to fix some variation I had with each square which ended up with a difunctional image of each square.



A picture of an individual square.

Second step: OCR to identify digits in each individual Square

This step is straightforward. We pass each image of the square to an OCR to identify the digit and append it to a vector. I used EasyOCR (https://github.com/JaidedAI/EasyOCR).

```python
def get_digits(images):
    "get digit as string from images list and append them as integers into another list using an OCR"
    digits = []
    SudokuVector = []
    for i in range (81):
        print(f"Getting digit {i + 1} of 81")
        up_width = 400
        up_height = 400
        up_points = (up_width, up_height)
        resized_up = cv.resize(images[i], up_points, interpolation= cv.INTER_LINEAR)
        cv.imwrite("ActualPicture.JPG",resized_up)
        #Use OCR to extract digit from digit image
        result = reader.readtext('ActualPicture.JPG')



        if len(result)> 0:
            digits.append(int(result[0][1]))
        else:
            digits.append(0)

    #remove temporary JPG

    os.remove("ActualPicture.JPG")

    #transform digits list to the sudoku vector for backtracking algorithm

    for i in range (9):
        for j in range (9):
            SudokuVector.append(digits[(j * 9) + i])

    return SudokuVector
```

In the red arrow I rescale the individual images. This is because the OCR couldn't identify the digits correctly with the original image size. It is worth noting white spaces are replaced wit ha 0 for the backtracking algorithm. In the blue arrow I append each digit to a vector. The images are listed in a vertical way, so I need to apply some logic to order them horizontally.

Third step: Backtracking Algorithm:

```
[12]: vector

[12]: [3,
       9,
       1,
       5,
       6,
       1,
       0,
       4,
       7,
       8,
       7,
       0,
       9,
```

Vector of an example Sudoku

```python
def backtrackingSudoku (vector,vector0,step,soluciones):
    if ValidateNumber(step,vector) == False:
        backtrackingSudoku (vector,vector0,step + 1,soluciones)
    elif ValidateNumber(step,vector) == True:
        for i in range (1,10):
            if ValidateRow(i,step,vector) and  ValidateColumn(i,step,vector) and Validate_box(i,step,vector):
                vector[step] = i
                if 0 not in vector:
                    if solucion(vector):
                        soluciones.append(vector.copy())
                else:
                    backtrackingSudoku (vector,vector0,step + 1,soluciones)
                vector[step] = 0
```

Backtracking Algorithm

The algorithm is simple. The sudoku is transformed into a vector (ordered horizontally). A 0 means that it's a space we can occupy, otherwise is a number already provided in the puzzle. Each step of the backtracking is the next index of the vector. If Vector[step] ! = 0, then we call the algorithm again. If it's a 0 then we try digits 1 to 9, validating row, column and quadrant (9x9 box), and call the algorithm recursively. To validate if the number can be put there, we use multiples of 3 and 9 to identify positions that need to be verified. For example, in Vector[step], we would verify if all the vector[x . 9] have the number we want to try, to validate column.

```
[10]: print("Original Sudoku:", end = "\n")
      printSudoku(vector0)
      print("Solution for Sudoku:", end = "\n")
      for i in soluciones:
          printSudoku(i)
          print(solucion(i))
```

```
Original Sudoku:
---------------------------------
[3, 9, 1] | [5, 6, 0] | [0, 4, 7]
[8, 7, 0] | [9, 4, 0] | [1, 2, 5]
[0, 0, 2] | [1, 8, 0] | [0, 0, 0]
---------------------------------
[0, 0, 0] | [2, 0, 9] | [0, 7, 8]
[9, 0, 0] | [0, 0, 4] | [6, 0, 1]
[0, 0, 3] | [7, 0, 0] | [0, 0, 0]
---------------------------------
[0, 3, 4] | [0, 0, 0] | [7, 0, 2]
[0, 0, 0] | [0, 0, 1] | [0, 6, 0]
[7, 6, 0] | [4, 0, 0] | [0, 0, 0]
---------------------------------
Solution for Sudoku:
---------------------------------
[3, 9, 1] | [5, 6, 2] | [8, 4, 7]
[8, 7, 6] | [9, 4, 3] | [1, 2, 5]
[5, 4, 2] | [1, 8, 7] | [3, 9, 6]
---------------------------------
[6, 1, 5] | [2, 3, 9] | [4, 7, 8]
[9, 2, 7] | [8, 5, 4] | [6, 3, 1]
[4, 8, 3] | [7, 1, 6] | [2, 5, 9]
---------------------------------
[1, 3, 4] | [6, 9, 5] | [7, 8, 2]
[2, 5, 8] | [3, 7, 1] | [9, 6, 4]
[7, 6, 9] | [4, 2, 8] | [5, 1, 3]
---------------------------------
True
```

Example of a Solved Sudoku

[5]:
```
def printSudoku(vector):
    matrix =  get_matrix(vector)
    multiplesOf3= multiples(3,10)
    count = 0
    for i in multiplesOf3[:-1]:
        pre = i
        if (count) % 3 == 0:
            print("-"* 33)
        print(matrix[pre], "|", matrix[pre+1],"|", matrix[pre+2])
        count += 1
    print("-"* 33)
```

[6]:
```
def ValidateNumber(step,vector):
    result = True
    if vector[step] != 0:
        result = False
    return result




def ValidateRow(number,step,vector):
    pre = 0
    result = True
    for i in multiples(9,10):
        if step > pre and step < i:
            if number in vector[pre:i]:
                result = False
        pre = i
    return result


def ValidateColumn(number,step,vector):
    result = True
    for i in multiples(9,9):
        if number == vector[step - i]:
            result = False
    return result
```

Example of validating functions