```
import cv2
import matplotlib.pyplot as plt
import sys
from google.colab import drive
drive.mount('/content/drive')
from PIL import Image, ImageFilter, ImageOps
import numpy as np
import math
import scipy.sparse as sps
from scipy.signal import convolve2d as conv2d
```

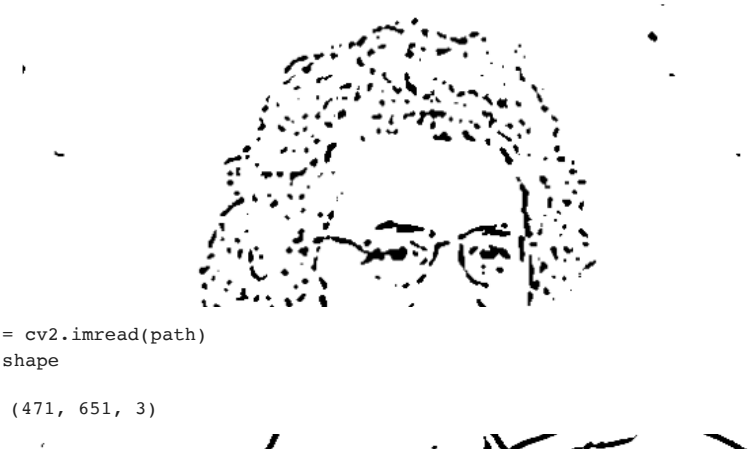
```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=
```

Edge Detection

```
def cartoonify(ImagePath):
    originalmage = cv2.imread(ImagePath)
    originalmage = cv2.cvtColor(originalmage, cv2.COLOR_BGR2RGB)
    shape = originalmage.shape[:2]
    print(shape)
# check if the image is chosen
    if originalmage is None:
        print("Can not find any image. Choose appropriate file")
        sys.exit()
    # ReSized1 = cv2.resize(originalmage, shape)
    ReSized1 = originalmage
    grayScaleImage = cv2.cvtColor(originalmage, cv2.COLOR_BGR2GRAY)
    ReSized2 = grayScaleImage
    # ReSized2 = cv2.resize(grayScaleImage, shape)
#applying median blur to smoothen an image
    smoothGrayScale = cv2.medianBlur(grayScaleImage, 5)
    ReSized3 = smoothGrayScale
    # ReSized3 = cv2.resize(smoothGrayScale, shape)
#retrieving the edges for cartoon effect
    getEdge = cv2.adaptiveThreshold(smoothGrayScale, 255,
      cv2.ADAPTIVE_THRESH_MEAN_C,
      cv2.THRESH_BINARY, 9, 9)
    ReSized4 = cv2.medianBlur(getEdge,3)
    # ReSized4 = getEdge
    # ReSized4 = cv2.resize(getEdge, shape)
#applying bilateral filter to remove noise
    #and keep edge sharp as required
    colorImage = cv2.bilateralFilter(originalmage, 9, 300, 300)
    ReSized5 = colorImage
    # ReSized5 = cv2.resize(colorImage, shape)
    #masking edged image with our "BEAUTIFY" image
    cartoonImage = cv2.bitwise_and(colorImage, colorImage, mask=getEdge)
    ReSized6 = cartoonImage
    # ReSized6 = cv2.resize(cartoonImage, shape)

#Plotting the whole transition
    images=[ReSized1, ReSized2, ReSized3, ReSized4, ReSized5, ReSized6]
    fig, axes = plt.subplots(3,2, figsize=(8,8), subplot_kw={'xticks':[], 'yticks':[]}, gridspec_kw=dict(hspace=0.1, wspace=0.1
    for i, ax in enumerate(axes.flat):
        ax.imshow(images[i], cmap='gray')
    plt.show()

    plt.imshow(ReSized4, cmap='gray')
    plt.show()
    return ReSized4, grayScaleImage



path ='/content/drive/MyDrive/neil.jpeg'
edge, grayScaleImage = cartoonify(path)
```

```
(471, 651)
```


```
# plt.imshow(edge)
fig = plt.figure(frameon=False)
# fig.set_size_inches(shape)
ax = plt.Axes(fig, [0., 0., 1., 1.])
ax.set_axis_off()
fig.add_axes(ax)
ax.imshow(edge, aspect='auto', cmap='gray')
# fig.savefig(fname, dpi)
plt.savefig("/content/drive/MyDrive/neil_edge.jpeg")
```



```
img = cv2.imread(path)
img.shape
```

```
(471, 651, 3)
```

```
plt.imshow(img)
```

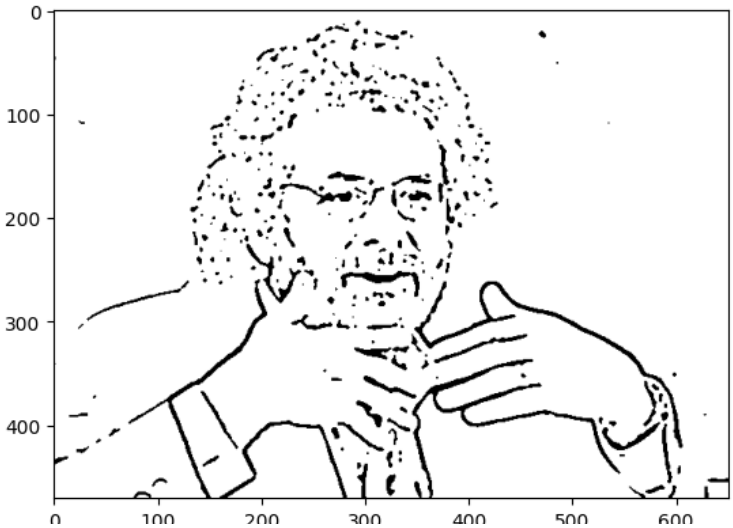
```
<matplotlib.image.AxesImage at 0x7b797754ab00>
```


```
RGB_img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
plt.imshow(RGB_img)
```

```
<matplotlib.image.AxesImage at 0x7b796c8bb370>
```


```
m = cv2.medianBlur(edge,3)
plt.imshow(m, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7b7962b83760>
```




```
edges = cv2.Canny(image=img, threshold1=100, threshold2=200)

# Display Canny Edge Detection Image
fig = plt.figure(frameon=False)
ax = plt.Axes(fig, [0., 0., 1., 1.])
ax.set_axis_off()
fig.add_axes(ax)
plt.imshow(255-edges, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7b7961e89b40>
```

```
# Opening the image (R prefixed to string
# in order to deal with '\' in paths)
image = Image.open(path)

# Converting the image to grayscale, as edge detection
# requires input image to be of mode = Grayscale (L)
image = image.convert("L")

# Detecting Edges on the Image using the argument ImageFilter.FIND_EDGES
image = image.filter(ImageFilter.FIND_EDGES)

image = ImageOps.invert(image)

# # Saving the Image Under the name Edge_Sample.png
# image.save(r"Edge_Sample.png")
```

```
fig = plt.figure(frameon=False)
ax = plt.Axes(fig, [0., 0., 1., 1.])
ax.set_axis_off()
fig.add_axes(ax)
plt.imshow(image, cmap="gray")
```

```
<matplotlib.image.AxesImage at 0x7b79623984f0>
```

```
def sparseMatrix(i, j, Aij, imsize):
    """ Build a sparse matrix containing 2D linear neighborhood operators
    Input:
        Aij = [ni, nj, nc] nc: number of neighborhoods with contraints
        i: row index
        j: column index
        imsize: [nrows ncols]
    Returns:
        A: a sparse matrix. Each row contains one 2D linear operator
    """
    ni, nj, nc = Aij.shape
    nij = ni*nj

    a = np.zeros((nc*nij))
    m = np.zeros((nc*nij))
    n = np.zeros((nc*nij))
    grid_range = np.arange(-(ni-1)/2, 1+(ni-1)/2)
    jj, ii = np.meshgrid(grid_range, grid_range)
    ii = ii.reshape(-1,order='F')
    jj = jj.reshape(-1,order='F')


    k = 0
    for c in range(nc):
        # Get matrix index
        x = (i[c]+ii) + (j[c]+jj)*nrows
        a[k:k+nij] = Aij[:,:,c].reshape(-1,order='F')
        m[k:k+nij] = c
        n[k:k+nij] = x

        k += nij

    m = m.astype(np.int32)
    n = n.astype(np.int32)
    A = sps.csr_matrix((a, (m,  n)))

    return A


# World parameters
theta = 35*math.pi/180
```

```
img = cv2.imread(path) # modified the image names here
img = img[:, :, ::-1].astype(np.float32)

nrows, ncols, colors = img.shape
ground = (np.min(img, axis=2) > 110).astype(np.float32)
print('ground', ground.shape, ground)
foreground = (ground == 0).astype(np.float32)

m = np.mean(img, 2)
kern = np.array([[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]], dtype=np.float32)
dmdx = conv2d(m, kern, 'same')
dmdy = conv2d(m, kern.transpose(), 'same')

mag = np.sqrt(dmdx**2 + dmdy**2)
mag[0, :] = 0
mag[-1, :] = 0
mag[:, 0] = 0
mag[:, -1] = 0

edge_orientation = np.arctan2(dmdx, dmdy)
edges = mag >= 30
edges = edges * foreground

## Occlusion and contact edges
pi = math.pi
vertical_edges = edges*((edge_orientation<115*pi/180)*(edge_orientation>65*pi/180)+(edge_orientation<-65*pi/180)*(edge_orienta
horizontal_edges = edges * (1-vertical_edges)

kern = np.array([[-1, -2, -1], [0, 0, 0], [1, 2, 1]], dtype=np.float32)
horizontal_ground_to_foreground_edges = (conv2d(ground, kern, 'same'))>0;
horizontal_foreground_to_ground_edges = (conv2d(foreground, kern, 'same'))>0;
vertical_ground_to_foreground_edges = vertical_edges*np.abs(conv2d(ground, kern.transpose(), 'same'))>0


occlusion_edges = edges*(vertical_ground_to_foreground_edges + horizontal_ground_to_foreground_edges)
contact_edges   = horizontal_edges*(horizontal_foreground_to_ground_edges);


E = np.concatenate([vertical_edges[:,:,None],
                    horizontal_edges[:,:,None],
                    np.zeros(occlusion_edges.shape)[:,:,None]], 2)


# Plot
plt.figure()
plt.subplot(2,2,1)
plt.imshow(img.astype(np.uint8))
plt.axis('off')
plt.title('Input image')
plt.subplot(2,2,2)
plt.imshow(edges == 0, cmap='gray')
plt.axis('off')
plt.title('Edges')

# Normals
K = 3
ey, ex = np.where(edges[::K, ::K])
ex *= K
ey *= K
plt.figure()
plt.subplot(2,2,3)
plt.imshow(np.max(mag)-mag, cmap='gray')
dxe = dmdx[::K, ::K][edges[::K, ::K] > 0]
dye = dmdy[::K, ::K][edges[::K, ::K] > 0]
n = np.sqrt(dxe**2 + dye**2)
dxe = dxe/n
dye = dye/n
plt.quiver(ex, ey, dxe, -dye, color='r')
plt.axis('off')
plt.title('Normals')


plt.subplot(2,2,4)
plt.imshow(np.max(mag)-mag, cmap='gray')
# Recreate the normals plot using sin and cos
```

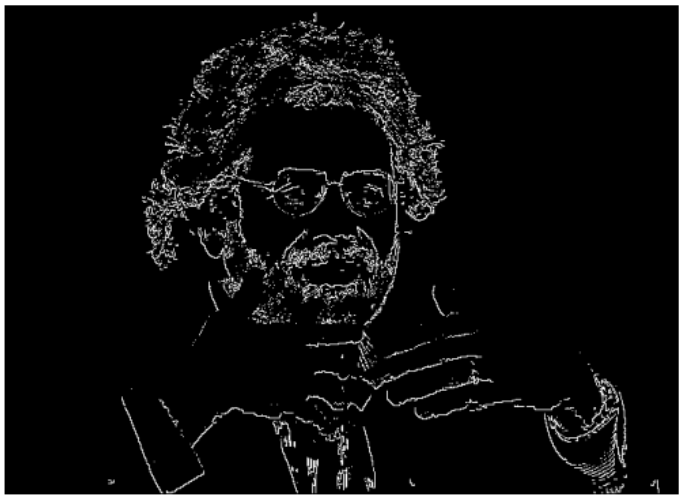
```
# Note: -dye_mod  used in plot because 0 is upper right corner here
dxe_mod = nx = np.sin(edge_orientation[::K, ::K][edges[::K, ::K] > 0])
dye_mod = ny = np.cos(edge_orientation[::K, ::K][edges[::K, ::K] > 0])
plt.quiver(ex, ey, dxe_mod, -dye_mod, color='r') # at ex, ey location, plot dx, -dy
plt.axis('off')
plt.title('Recreated Normals')
plt.show()


# Edges and boundaries
plt.figure()
plt.subplot(2,2,1)
plt.imshow(img.astype(np.uint8))
plt.axis('off')
plt.title('Input image')


plt.subplot(2,2,2)
plt.imshow(E+(edges == 0)[:, :, None])
plt.axis('off')
plt.title('Edges')


plt.subplot(2,2,3)
plt.imshow(1-(occlusion_edges>0), cmap='gray')
plt.axis('off')
plt.title('Occlusion boundaries')

plt.subplot(2,2,4)
plt.imshow(1-contact_edges, cmap='gray')
plt.axis('off')
plt.title('Contact boundaries')
```

```
ground (471, 651) [[0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 [0. 0. 0. ... 0. 0. 0.]
 ...
 [0. 0. 0. ... 1. 1. 1.]
 [0. 0. 0. ... 1. 1. 1.]
 [0. 0. 0. ... 1. 1. 1.]]
```


```
plt.plot()
plt.imshow(1-(occlusion_edges>0), cmap='gray')
# plt.title('Occlusion boundaries')
plt.axis('off')
plt.savefig("/content/drive/MyDrive/neil_ocllusion.jpeg")
```

```
plt.plot()
plt.imshow(occlusion_edges, cmap='gray')
plt.axis('off')
```

```
(-0.5, 650.5, 470.5, -0.5)
```



Poisson Disc Sampling

```
path ='/content/drive/MyDrive/neil.jpeg'
image = cv2.imread(path)
image.shape
```

```
(471, 651, 3)
```

```
def get_cell_coords(pt):
    """Get the coordinates of the cell that pt = (x,y) falls in."""
```

```
    return int(pt[0] // a), int(pt[1] // a)

def get_neighbours(coords):
    """Return the indexes of points in cells neighbouring cell at coords.

    For the cell at coords = (x,y), return the indexes of points in the cells
    with neighbouring coordinates illustrated below: ie those cells that could
    contain points closer than r.

                                     ooo
                                    ooooo
                                    ooXoo
                                    ooooo
                                     ooo

    """

    dxdy = [(-1,-2),(0,-2),(1,-2),(-2,-1),(-1,-1),(0,-1),(1,-1),(2,-1),
            (-2,0),(-1,0),(1,0),(2,0),(-2,1),(-1,1),(0,1),(1,1),(2,1),
            (-1,2),(0,2),(1,2),(0,0)]
    neighbours = []
    for dx, dy in dxdy:
        neighbour_coords = coords[0] + dx, coords[1] + dy
        if not (0 <= neighbour_coords[0] < nx and
                0 <= neighbour_coords[1] < ny):
            # We're off the grid: no neighbours here.
            continue
        neighbour_cell = cells[neighbour_coords]
        if neighbour_cell is not None:
            # This cell is occupied: store this index of the contained point.
            neighbours.append(neighbour_cell)
    return neighbours

def point_valid(pt):
    """Is pt a valid point to emit as a sample?

    It must be no closer than r from any other point: check the cells in its
    immediate neighbourhood.

    """

    cell_coords = get_cell_coords(pt)
    for idx in get_neighbours(cell_coords):
        nearby_pt = samples[idx]
        # Squared distance between or candidate point, pt, and this nearby_pt.
        distance2 = (nearby_pt[0]-pt[0])**2 + (nearby_pt[1]-pt[1])**2
        if distance2 < r**2:
            # The points are too close, so pt is not a candidate.
            return False
    # All points tested: if we're here, pt is valid
    return True

def get_point(k, refpt):
    """Try to find a candidate point relative to refpt to emit in the sample.

    We draw up to k points from the annulus of inner radius r, outer radius 2r
    around the reference point, refpt. If none of them are suitable (because
    they're too close to existing points in the sample), return False.
    Otherwise, return the pt.

    """
    i = 0
    while i < k:
        i += 1
        rho = np.sqrt(np.random.uniform(r**2, 4 * r**2))
        theta = np.random.uniform(0, 2*np.pi)
        pt = refpt[0] + rho*np.cos(theta), refpt[1] + rho*np.sin(theta)
        if not (0 <= pt[0] < width and 0 <= pt[1] < height):
            # This point falls outside the domain, so try again.
            continue
        if point_valid(pt):
            return pt

    # We failed to find a suitable point in the vicinity of refpt.
```

```
    return False


# Choose up to k points around each reference point as candidates for a new
# sample point
k = 30

# Minimum distance between samples
r = 5

height, width = image.shape[:2]
print(f'width is {width}, height is {height}')

# Cell side length
a = r/np.sqrt(2)
# Number of cells in the x- and y-directions of the grid
nx, ny = int(width / a) + 1, int(height / a) + 1

# A list of coordinates in the grid of cells
coords_list = [(ix, iy) for ix in range(nx) for iy in range(ny)]
# Initilalize the dictionary of cells: each key is a cell's coordinates, the
# corresponding value is the index of that cell's point's coordinates in the
# samples list (or None if the cell is empty).
cells = {coords: None for coords in coords_list}

# Pick a random point to start with.
pt = (np.random.uniform(0, width), np.random.uniform(0, height))
samples = [pt]
# Our first sample is indexed at 0 in the samples list...
cells[get_cell_coords(pt)] = 0
# ... and it is active, in the sense that we're going to look for more points
# in its neighbourhood.
active = [0]

nsamples = 1
# As long as there are points in the active list, keep trying to find samples.
while active:
    # choose a random "reference" point from the active list.
    idx = np.random.choice(active)
    refpt = samples[idx]
    # Try to pick a new point relative to the reference point.
    pt = get_point(k, refpt)
    if pt:
        # Point pt is valid: add it to the samples list and mark it as active
        samples.append(pt)
        nsamples += 1
        active.append(len(samples)-1)
        cells[get_cell_coords(pt)] = len(samples) - 1
    else:
        # We had to give up looking for valid points near refpt, so remove it
        # from the list of "active" points.
        active.remove(idx)

# plt.scatter(*zip(*samples), color='r', alpha=0.6, lw=0)
# plt.xlim(0, width)
# plt.ylim(0, height)
# # plt.axis('off')
# plt.show()
```

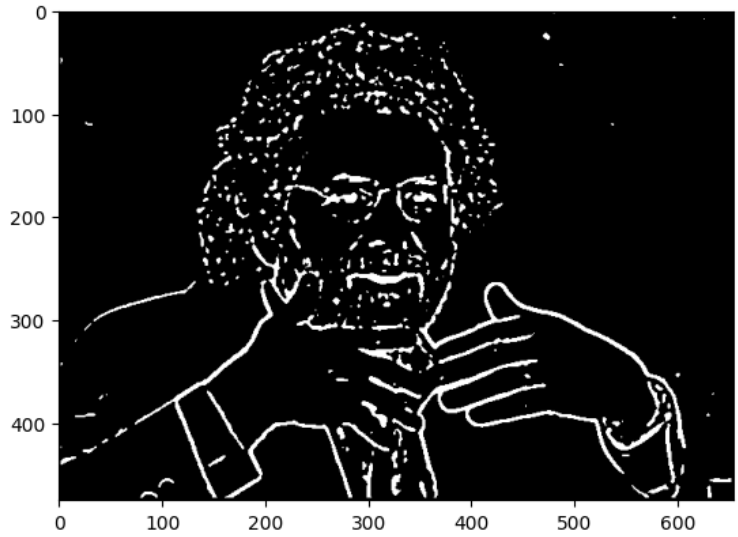
```
 width is 651, height is 471
```

```
edge_bit = 1.0-(edge/255.0 >= 0.5)*1.0


plt.imshow(edge_bit, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7b78b83679a0>
```


```
w, h = image.shape[:2]
edge_bit_large = np.zeros((w+4, h+4))
edge_bit_large[2:2+w,2:2+h] = edge_bit
plt.imshow(edge_bit_large, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7b796cf8b310>
```




```
len(samples)
```

```
7652
```

```
area = []
xs = []
ys = []

for i in range(len(samples)):
    x, y = samples[i]
    x_round = round(x)+2
    y_round = round(y)+2
    sum = np.sum(edge_bit_large[height-y_round-2:height-y_round+3,x_round-2:x_round+3])
    if sum > 0:
        xs.append(x)
        ys.append(y)
        area.append(sum/5)

print(len(area))
del samples
```

```
1333
```

```
area = [i*1.5 for i in area]


plt.scatter(xs,ys, color='k', alpha=1, lw=0, s=area)
plt.xlim(0, width)
plt.ylim(0, height)
plt.axis('off')
plt.show()
```

```
image = cv2.imread(path)
plt.figure(figsize=(12.5*2,9*2))

# ax = [plt.subplot(2,2,i+1) for i in range(4)]

# for a in ax:
#     a.set_xticklabels([])
#     a.set_yticklabels([])

plt.subplots_adjust(wspace=0, hspace=0)

plt.subplot(3,3,1)
plt.imshow(image.astype(np.uint8))
plt.axis('off')

plt.subplot(3,3,2)
plt.imshow(grayScaleImage, cmap='gray')
plt.axis('off')

plt.subplot(3,3,3)
plt.imshow(edge, cmap='gray')
plt.axis('off')

plt.subplot(3,3,5)
plt.scatter(xs,ys, color='k', alpha=1, lw=0, s=area)
plt.axis('off')

plt.subplot(3,3,4)
plt.imshow(edge_bit, cmap='gray')
plt.axis('off')

plt.subplot(3,3,6)
plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.axis('off')

plt.subplot(3,3,7)
plt.imshow(edge, cmap='gray')
plt.axis('off')

plt.subplot(3,3,8)
plt.imshow(255-edges, cmap='gray')
plt.axis('off')

plt.subplot(3,3,9)
plt.imshow(np.max(mag)-mag, cmap='gray')
dxe_mod = nx = np.sin(edge_orientation[::K, ::K][edges[::K, ::K] > 0])
dye_mod = ny = np.cos(edge_orientation[::K, ::K][edges[::K, ::K] > 0])
plt.quiver(ex, ey, dxe_mod, -dye_mod, color='r')
plt.axis('off')

plt.show()
```

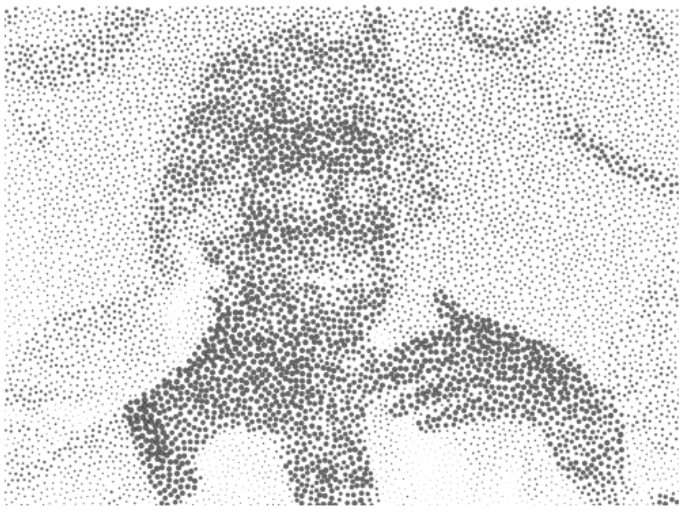
```
ls_points = []
for i in range(len(xs)):
    ls_points.append((round(xs[i],2), round(ys[i],2), round(area[i],2)))


with open("/content/drive/MyDrive/file.txt", "w") as output:
    output.write(str(ls_points))


# grayScaleImage
w, h = image.shape[:2]
grayScaleImage_large = np.zeros((w+4, h+4))
grayScaleImage_large[2:2+w,2:2+h] = grayScaleImage
plt.imshow(grayScaleImage_large, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f3da37693c0>
```


```
area = []
xs = []
ys = []

for i in range(len(samples)):
    x, y = samples[i]
    x_round = round(x)+2
    y_round = round(y)+2
    sum = np.sum(grayScaleImage_large[height-y_round-2:height-y_round+3,x_round-2:x_round+3])
    if sum > 0:
        xs.append(x)
        ys.append(y)
        area.append(sum/5)

print(len(area))
# del samples
```

```
7531
```

```
area_new = [i/100 for i in area]


plt.scatter(xs,ys, color='k', alpha=0.6, lw=0, s=area_new)
plt.xlim(0, width)
plt.ylim(0, height)
plt.axis('off')
plt.savefig("/content/drive/MyDrive/scatter_inverse.jpg")
plt.show()
```



```
scatter_inv = cv2.imread("/content/drive/MyDrive/scatter_inverse.jpg")
plt.imshow(255-scatter_inv)
```

```
<matplotlib.image.AxesImage at 0x7f3da79fb610>
```

Grayscale + Edge + Poisson


```
import cv2
import matplotlib.pyplot as plt
import sys
import numpy as np

google_colab = True

if google_colab:
    from google.colab import drive
    drive.mount('/content/drive')
```

```
 Mounted at /content/drive
```

```
def halftone(image_path, thresh=3, radius=10, savefig = True, halftone_path='halftone.jpeg', BGR=True, dotsize=30):
    image = cv2.imread(image_path)
    if BGR:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
    # image = cv2.resize(image, (0, 0), fx = 0.5, fy = 0.5)

    if image is None:
        print("Can not find any image. Choose appropriate file")
        sys.exit()
    k = 30
    r = radius
    height, width = image.shape[:2]

    a = r/np.sqrt(2)
    nx, ny = int(width / a) + 1, int(height / a) + 1

    coords_list = [(ix, iy) for ix in range(nx) for iy in range(ny)]
    cells = {coords: None for coords in coords_list}
    def get_cell_coords(pt):
        return int(pt[0] // a), int(pt[1] // a)

    def get_neighbours(coords):
        dxdy = [(-1,-2),(0,-2),(1,-2),(-2,-1),(-1,-1),(0,-1),(1,-1),(2,-1),
                (-2,0),(-1,0),(1,0),(2,0),(-2,1),(-1,1),(0,1),(1,1),(2,1),
                (-1,2),(0,2),(1,2),(0,0)]
        neighbours = []
        for dx, dy in dxdy:
            neighbour_coords = coords[0] + dx, coords[1] + dy
            if not (0 <= neighbour_coords[0] < nx and
                    0 <= neighbour_coords[1] < ny):
                # We're off the grid: no neighbours here.
                continue
            neighbour_cell = cells[neighbour_coords]
            if neighbour_cell is not None:
                # This cell is occupied: store this index of the contained point.
                neighbours.append(neighbour_cell)
        return neighbours

    def point_valid(pt):
        cell_coords = get_cell_coords(pt)
        for idx in get_neighbours(cell_coords):
            nearby_pt = samples[idx]
            # Squared distance between or candidate point, pt, and this nearby_pt.
            distance2 = (nearby_pt[0]-pt[0])**2 + (nearby_pt[1]-pt[1])**2
            if distance2 < r**2:
                # The points are too close, so pt is not a candidate.
                return False
        # All points tested: if we're here, pt is valid
        return True

    def get_point(k, refpt):
```

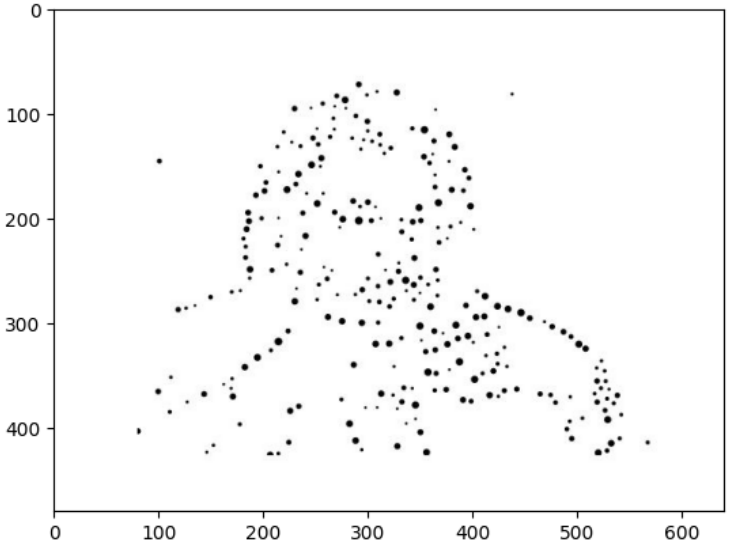
```
    i = 0
    while i < k:
        i += 1
        rho = np.sqrt(np.random.uniform(r**2, 4 * r**2))
        theta = np.random.uniform(0, 2*np.pi)
        pt = refpt[0] + rho*np.cos(theta), refpt[1] + rho*np.sin(theta)
        if not (0 <= pt[0] < width and 0 <= pt[1] < height):
            # This point falls outside the domain, so try again.
            continue
        if point_valid(pt):
            return pt

    # We failed to find a suitable point in the vicinity of refpt.
    return False
################################################
### Processs image to grayscale and get edges
### output: edge (think edge image)
################################################
grayScaleImage = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
smoothGrayScale = cv2.medianBlur(grayScaleImage, 5)
getEdge = cv2.adaptiveThreshold(smoothGrayScale, 255,
  cv2.ADAPTIVE_THRESH_MEAN_C,
  cv2.THRESH_BINARY, 9, 9)
edge = cv2.medianBlur(getEdge,3)

################################################
### Make Poisson Disc Sampling
### output: samples
################################################
pt = (np.random.uniform(0, width), np.random.uniform(0, height))
samples = [pt]
cells[get_cell_coords(pt)] = 0
active = [0]

nsamples = 1
while active:
    idx = np.random.choice(active)
    refpt = samples[idx]
    pt = get_point(k, refpt)
    if pt:
        samples.append(pt)
        nsamples += 1
        active.append(len(samples)-1)
        cells[get_cell_coords(pt)] = len(samples) - 1
    else:
        active.remove(idx)

################################################
### halftone
################################################
edge_bit = 1.0-(edge/255.0 >= 0.5)*1.0
w, h = image.shape[:2]
edge_bit_large = np.zeros((w+4, h+4))
edge_bit_large[2:2+w,2:2+h] = edge_bit

area = []
xs = []
ys = []

for i in range(len(samples)):
    x, y = samples[i]
    x_round = round(x)+2
    y_round = round(y)+2
    sum = np.sum(edge_bit_large[height-y_round-2:height-y_round+3,x_round-2:x_round+3])
    if sum > thresh:
        xs.append(x)
        ys.append(y)
        area.append(sum/25)

area = [i*dotsize for i in area]

plt.scatter(xs,ys, color='k', alpha=1, lw=0, s=area)
plt.xlim(0, width)
plt.ylim(0, height)
plt.axis('off')
plt.savefig(halftone_path)
```

```
    # if savefig:
    #     if halftone_path == '':
    #         plt.show()
    #         print("Please provide a halftone_path, not saving image")
    #     else:
    #         plt.savefig(halftone_path)
    #         plt.show()
    # else:
    #     plt.show()
    img = cv2.imread(halftone_path)

    ls_points = []
    for i in range(len(xs)):
        ls_points.append([round(xs[i],2), round(ys[i],2), round(area[i],2)])
    print(f"{len(ls_points)} points are generated.")

    return img, ls_points


path ='/content/drive/MyDrive/neil.jpeg'
img, ls = halftone(path, thresh = 2, radius = 10)
```

```
271 points are generated.
```

```
plt.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7f2871e74f40>
```




```
ls
```

```
path ='/content/drive/MyDrive/neil_2.jpeg'
ls = halftone(path, thresh = 2, radius = 5)
```

```
    442 points are generated.
```

```
path ='/content/drive/MyDrive/neil_3.jpeg'
ls = halftone(path, thresh = 2, radius = 8, BGR=False)
```

```
    187 points are generated.
```

```
path ='/content/drive/MyDrive/neil_4.jpeg'
ls = halftone(path, thresh = 0, radius = 5, BGR=True)
```

Line Algorithm

```
def cartoonify(ImagePath):
    originalmage = cv2.imread(ImagePath)
    originalmage = cv2.cvtColor(originalmage, cv2.COLOR_BGR2RGB)
    shape = originalmage.shape[:2]
    print(shape)
    if originalmage is None:
        print("Can not find any image. Choose appropriate file")
        sys.exit()
    ReSized1 = originalmage
    grayScaleImage = cv2.cvtColor(originalmage, cv2.COLOR_BGR2GRAY)
    ReSized2 = grayScaleImage
    smoothGrayScale = cv2.medianBlur(grayScaleImage, 5)
    ReSized3 = smoothGrayScale
    getEdge = cv2.adaptiveThreshold(smoothGrayScale, 255,
      cv2.ADAPTIVE_THRESH_MEAN_C,
      cv2.THRESH_BINARY, 9, 9)
    ReSized4 = cv2.medianBlur(getEdge,3)

    return ReSized4, grayScaleImage, ReSized3


path ='/content/drive/MyDrive/neil.jpeg'
edge, gray, blur = cartoonify(path)
```

```
(471, 651)
```

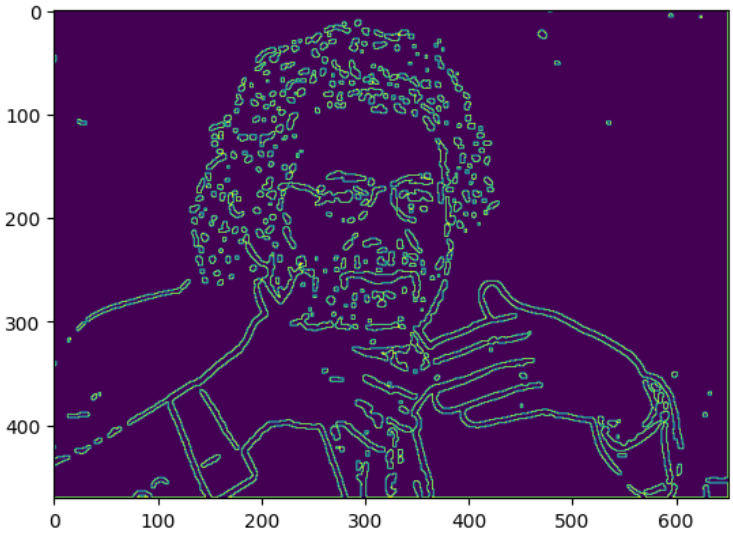
```
edge_bit = (edge > 255/2) * 1.0
plt.imshow(edge_bit, cmap = 'gray')
# edge_bit
```

```
<matplotlib.image.AxesImage at 0x7f288649cfa0>
```


```
width, height = edge_bit.shape


# for i in range(width):
#     for j in range(height):



from PIL import Image, ImageFilter
from matplotlib import cm

contour = Image.fromarray(np.uint8(cm.gist_earth(edge_bit)*255))
```

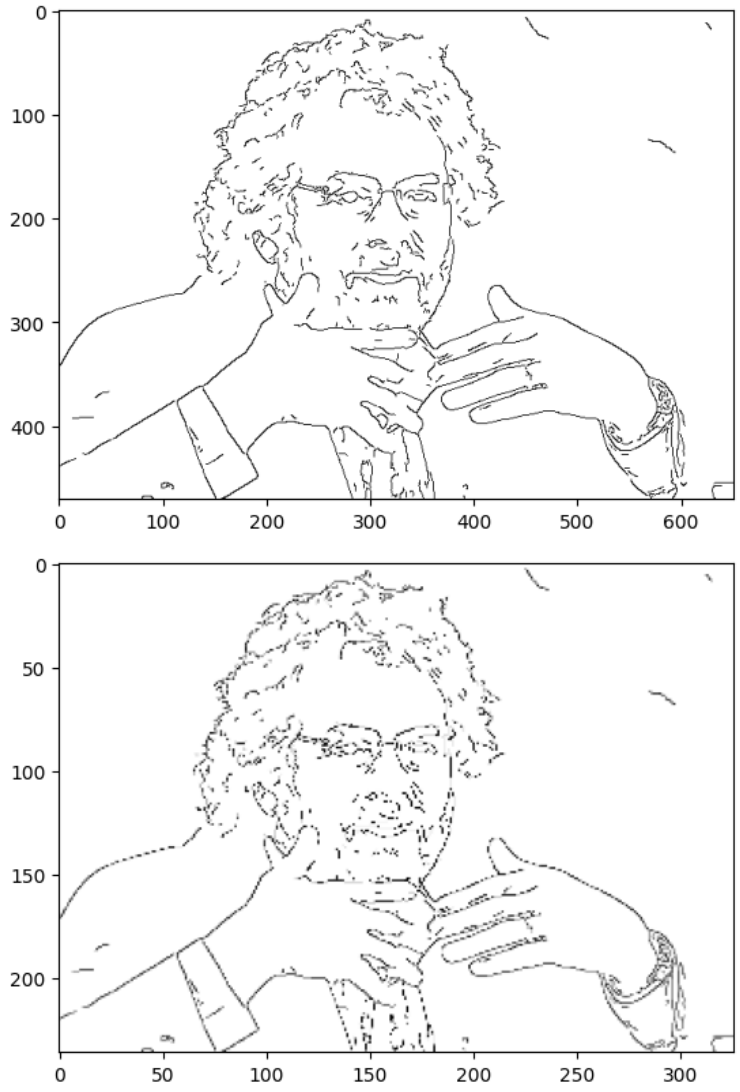
```
# Opening the image (R prefixed to string
# in order to deal with '\' in paths)
# image = Image.open(r"Sample.png")

# Converting the image to grayscale, as edge detection
# requires input image to be of mode = Grayscale (L)
contour = image.convert("L")

# Detecting Edges on the Image using the argument ImageFilter.FIND_EDGES
contour = image.filter(ImageFilter.FIND_EDGES)

# Saving the Image Under the name Edge_Sample.png
plt.imshow(contour)
# image.save(r"Edge_Sample.png")
```

```
<matplotlib.image.AxesImage at 0x7b796dd30be0>
```




```
edges = cv2.Canny(image=blur, threshold1=100, threshold2=140)

# Display Canny Edge Detection Image
fig = plt.figure(frameon=False)
ax = plt.Axes(fig, [0., 0., 1., 1.])
ax.set_axis_off()
fig.add_axes(ax)
plt.imshow(255-edges, cmap='gray')

line_only = ((255-edges) > 255/2) * 1.0
```

```
plt.imshow(line_only, cmap = "gray")
plt.show()
plt.imshow(cv2.resize(line_only, (round(line_only.shape[1]/2), round(line_only.shape[0]/2))), cmap = "gray")
plt.show()
# print(round(line_only.shape[0]/2), round(line_only.shape[1]/2))
# print(round(line_only.shape[0]), round(line_only.shape[1]))
```




```
######## path finding algorithm

paths = []
line_removing = line_only.copy()
w, h = line_only.shape

def find_contour(img, start_x, start_y, path):
    path.append((start_x, start_y))
    img[start_x, start_y] = 0
    ls = [[-1,-1],[-1,0],[-1,1],[0,-1],[0,1],[1,-1],[1,0],[1,1]]
    for pt in ls:
        i = pt[0]
        j = pt[1]
        if start_x+i>=0 and start_x+i<w and start_y+j>=0 and start_y+j<h:
            if img[start_x+i, start_y+j] == 1.0:
                path, img = find_contour(img, i, j, path)
                break
    return path, img

for i in range(w):
    for j in range(h):
        if line_removing[i][j] == 1.0:
```

```
            path, line_removing = find_contour(line_removing, i, j, [])
            paths.append(path)


paths
```

```
    []
```

```
def is_valid_move(matrix, x, y, visited):
    rows, cols = len(matrix), len(matrix[0])
    return 0 <= x < rows and 0 <= y < cols and matrix[x][y] == 1 and not visited[x][y]

def dfs(matrix, x, y, visited, path):
    visited[x][y] = True
    path.append((x, y))

    # Define possible moves (up, down, left, right)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for move in moves:
        new_x, new_y = x + move[0], y + move[1]
        if is_valid_move(matrix, new_x, new_y, visited):
            dfs(matrix, new_x, new_y, visited, path)

def find_paths(matrix):
    rows, cols = len(matrix), len(matrix[0])
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    paths = []

    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == 1 and not visited[i][j]:
                path = []
                dfs(matrix, i, j, visited, path)
                if path:
                    paths.append(path)

    return paths

# Example usage:
image_matrix = [
    [1, 0, 1, 0, 1],
    [1, 1, 1, 0, 0],
    [0, 0, 0, 1, 1],
    [1, 0, 1, 0, 1],
]

result = find_paths(image_matrix)
print(result)
```

```
    [[(0, 0), (1, 0), (1, 1), (1, 2), (0, 2)], [(0, 4)], [(2, 3), (2, 4), (3, 4)], [(3, 0)], [(3, 2)]]
```

```
find_paths(line_only)
```

```
def is_valid_move(matrix, x, y, visited):
    rows, cols = len(matrix), len(matrix[0])
    return 0 <= x < rows and 0 <= y < cols and matrix[x][y] == 1 and not visited[x][y]

def find_paths(matrix):
    rows, cols = len(matrix), len(matrix[0])
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    paths = []

    # Define possible moves (up, down, left, right)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == 1 and not visited[i][j]:
                path = []
                stack = [(i, j)]

                while stack:
                    x, y = stack.pop()
```

```
                    if not visited[x][y]:
                        visited[x][y] = True
                        path.append((x, y))

                        for move in moves:
                            new_x, new_y = x + move[0], y + move[1]
                            if is_valid_move(matrix, new_x, new_y, visited):
                                stack.append((new_x, new_y))

                if path:
                    paths.append(path)

    return paths

# Example usage:
image_matrix = [
    [1, 0, 1, 0, 1],
    [1, 1, 1, 0, 0],
    [0, 0, 0, 1, 1],
    [1, 0, 1, 0, 1],
]

result = find_paths(image_matrix)
print(result)
```

```
[[(0, 0), (1, 0), (1, 1), (1, 2), (0, 2)], [(0, 4)], [(2, 3), (2, 4), (3, 4)], [(3, 0)], [(3, 2)]]
```

```
result = find_paths(line_only)
```

```
len(result)
```

```
18
```

```
flattened_list = [element for sublist in result for element in sublist]
len(flattened_list)
```

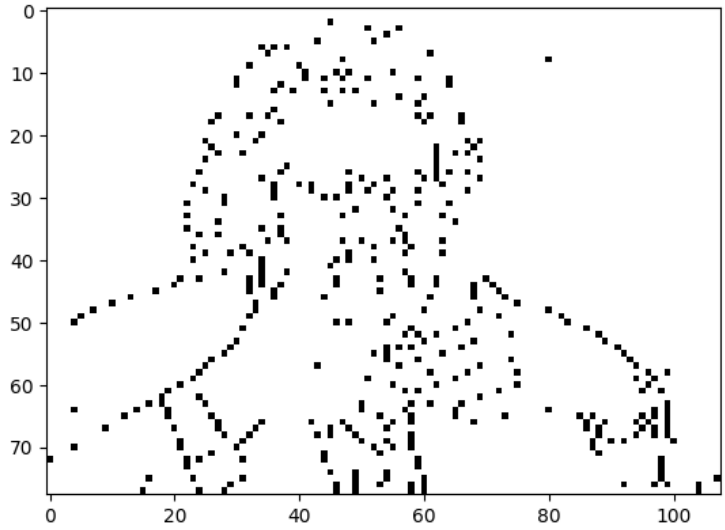
```
297210
```

line algo put together

```
def process(ImagePath):
    originalmage = cv2.imread(ImagePath)
    originalmage = cv2.cvtColor(originalmage, cv2.COLOR_BGR2RGB)
    shape = originalmage.shape[:2]
    print(shape)
    if originalmage is None:
        print("Can not find any image. Choose appropriate file")
        sys.exit()
    ReSized1 = originalmage
    grayScaleImage = cv2.cvtColor(originalmage, cv2.COLOR_BGR2GRAY)
    ReSized2 = grayScaleImage
    smoothGrayScale = cv2.medianBlur(grayScaleImage, 5)
    ReSized3 = smoothGrayScale
    getEdge = cv2.adaptiveThreshold(smoothGrayScale, 255,
      cv2.ADAPTIVE_THRESH_MEAN_C,
      cv2.THRESH_BINARY, 9, 9)
    ReSized4 = cv2.medianBlur(getEdge,3)

    return ReSized4, grayScaleImage, ReSized3

path ='/content/drive/MyDrive/neil.jpeg'
edge, gray, blur = process(path)
edge_bit = (edge > 255/2) * 1.0
plt.imshow(edge_bit, cmap = 'gray')
```

```
edge_bit_old = edge_bit


w, h = edge_bit.shape
edge_bit_new = cv2.resize(edge_bit, (round(h/6),round(w/6)))
w, h = edge_bit_new.shape


edge_bit_new = (edge_bit_new>0.5)*1.0


thin_edge = edge_bit_new.copy()
for i in range(w):
    for j in range(h-1):
        if thin_edge[i,j]==0 and thin_edge[i,j+1]==0:
            thin_edge[i,j]=1

plt.imshow(thin_edge, cmap='gray')
```

```
<matplotlib.image.AxesImage at 0x7f2886502bf0>
```




```
w*h-np.sum(thin_edge)
# np.sum(edge_bit)
```

```
382.0
```

```
def is_valid_move(matrix, x, y, visited):
    rows, cols = len(matrix), len(matrix[0])
    return 0 <= x < rows and 0 <= y < cols and matrix[x][y] == 1 and not visited[x][y]

def dfs(matrix, x, y, visited, path):
    visited[x][y] = True
    path.append([x, y])

    # Define possible moves (up, down, left, right)
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]

    for move in moves:
        new_x, new_y = x + move[0], y + move[1]
        if is_valid_move(matrix, new_x, new_y, visited):
            dfs(matrix, new_x, new_y, visited, path)

def find_paths(matrix):
    rows, cols = len(matrix), len(matrix[0])
    visited = [[False for _ in range(cols)] for _ in range(rows)]
    paths = []

    for i in range(rows):
        for j in range(cols):
            if matrix[i][j] == 1 and not visited[i][j]:
                path = []
                dfs(matrix, i, j, visited, path)
                if path:
                    paths.append(path)
```

```
    return paths

result = find_paths(1-thin_edge)
print(result)
```

```
[[[2, 45]], [[3, 51]], [[3, 56]], [[4, 54]], [[5, 43]], [[5, 52]], [[6, 34], [7, 35], [6, 36]], [[6, 38]], [[7, 61]], [[8,
```

```
len(result)
```

```
228
```

```
flattened_list = []
for sublist in result:
    flattened_list.append([sublist[0][0],sublist[0][1],0])
    flattened_list.append([sublist[0][0],sublist[0][1],1])
    if len(sublist) > 0:
        for i, element in enumerate(sublist[:-1]):
            flattened_list.append([sublist[i][0],sublist[i][1],1])

flattened_list
# len(flattened_list)
```

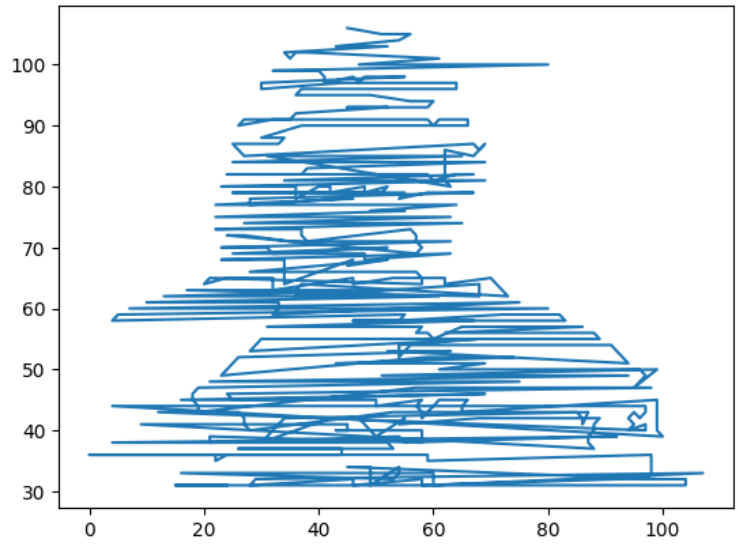
```
import cv2
import matplotlib.pyplot as plt

def line_alg(path):
    def process(ImagePath):
        originalmage = cv2.imread(ImagePath)
        originalmage = cv2.cvtColor(originalmage, cv2.COLOR_BGR2RGB)
        shape = originalmage.shape[:2]
        print(shape)
        if originalmage is None:
            print("Can not find any image. Choose appropriate file")
            sys.exit()
        ReSized1 = originalmage
        grayScaleImage = cv2.cvtColor(originalmage, cv2.COLOR_BGR2GRAY)
        ReSized2 = grayScaleImage
        smoothGrayScale = cv2.medianBlur(grayScaleImage, 5)
        ReSized3 = smoothGrayScale
        getEdge = cv2.adaptiveThreshold(smoothGrayScale, 255,
        cv2.ADAPTIVE_THRESH_MEAN_C,
        cv2.THRESH_BINARY, 9, 9)
        ReSized4 = cv2.medianBlur(getEdge,3)

        return ReSized4, grayScaleImage, ReSized3

    edge, gray, blur = process(path)
    edge_bit = (edge > 255/2) * 1.0
    plt.imshow(edge_bit, cmap = 'gray')

    w, h = edge_bit.shape
    edge_bit_new = cv2.resize(edge_bit, (round(h/6),round(w/6)))
    w, h = edge_bit_new.shape

    edge_bit_new = (edge_bit_new>0.5)*1.0

    thin_edge = edge_bit_new.copy()
    for i in range(w):
        for j in range(h-1):
            if thin_edge[i,j]==0 and thin_edge[i,j+1]==0:
                thin_edge[i,j]=1

    plt.imshow(thin_edge, cmap='gray')

    def is_valid_move(matrix, x, y, visited):
        rows, cols = len(matrix), len(matrix[0])
        return 0 <= x < rows and 0 <= y < cols and matrix[x][y] == 1 and not visited[x][y]

    def dfs(matrix, x, y, visited, path):
        visited[x][y] = True
```

```
        path.append([x, y])

        # Define possible moves (up, down, left, right)
        moves = [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1), (1, 1)]

        for move in moves:
            new_x, new_y = x + move[0], y + move[1]
            if is_valid_move(matrix, new_x, new_y, visited):
                dfs(matrix, new_x, new_y, visited, path)

    def find_paths(matrix):
        rows, cols = len(matrix), len(matrix[0])
        visited = [[False for _ in range(cols)] for _ in range(rows)]
        paths = []

        for i in range(rows):
            for j in range(cols):
                if matrix[i][j] == 1 and not visited[i][j]:
                    path = []
                    dfs(matrix, i, j, visited, path)
                    if path:
                        paths.append(path)

        return paths

    result = find_paths(1-thin_edge)
    # print(result)

    flattened_list = []
    for sublist in result:
        flattened_list.append([sublist[0][0],sublist[0][1],0])
        flattened_list.append([sublist[0][0],sublist[0][1],1])
        if len(sublist) > 0:
            for i, element in enumerate(sublist[:-1]):
                flattened_list.append([sublist[i][0],sublist[i][1],1])
    return thin_edge, flattened_list


path ='/content/drive/MyDrive/neil.jpeg'
line_alg(path)
```

```
(471, 651)
(array([[1., 1., 1., ..., 1., 1., 1.],
        [1., 1., 1., ..., 1., 1., 1.],
        [1., 1., 1., ..., 1., 1., 1.],
        ...,
        [1., 1., 1., ..., 1., 1., 0.],
        [1., 1., 1., ..., 1., 1., 1.],
        [1., 1., 1., ..., 1., 1., 1.]]),
 [[2, 45, 0],
  [2, 45, 1],
  [3, 51, 0],
  [3, 51, 1],
  [3, 56, 0],
  [3, 56, 1],
  [4, 54, 0],
  [4, 54, 1],
  [5, 43, 0],
  [5, 43, 1],
  [5, 52, 0],
  [5, 52, 1],
  [6, 34, 0],
  [6, 34, 1],
  [6, 34, 1],
  [7, 35, 1],
  [6, 38, 0],
  [6, 38, 1],
  [7, 61, 0],
  [7, 61, 1],
  [8, 47, 0],
  [8, 47, 1],
  [8, 80, 0],
  [8, 80, 1],
  [9, 32, 0],
  [9, 32, 1],
  [9, 40, 0],
  [9, 40, 1],
  [9, 40, 1],
  [10, 41, 1],
  [10, 46, 0],
  [10, 46, 1],
  [10, 46, 1],
  [11, 47, 1],
  [10, 55, 0],
  [10, 55, 1],
  [11, 30, 0],
  [11, 30, 1],
  [11, 30, 1],
  [11, 44, 0],
  [11, 44, 1],
  [11, 51, 0],
  [11, 51, 1],
  [11, 59, 0],
  [11, 59, 1],
  [11, 64, 0],
  [11, 64, 1],
  [11, 64, 1],
  [12, 37, 0],
```

```
x,y = zip(*flattened_list)
plt.plot(tuple(i for i in y),tuple(h-j for j in x))
```

```
[<matplotlib.lines.Line2D at 0x7b78b7a6dd20>]
```




```
for ls in result:
    if len(ls)>1:
        print(ls)
        plt.plot(ls)
plt.show()
```

[(6, 62), (7, 62)]
[(7, 71), (8, 71), (9, 71)]
[(13, 71), (14, 71), (15, 71)]
[(16, 61), (17, 61)]
[(17, 66), (18, 66)]
[(17, 88), (18, 88)]
[(18, 44), (19, 44)]
[(19, 58), (20, 58)]
[(24, 86), (25, 86)]
[(25, 56), (26, 56), (27, 56), (28, 56)]
[(27, 89), (28, 89)]
[(30, 44), (31, 44)]
[(30, 49), (31, 49), (32, 49)]
[(31, 96), (32, 96)]
[(32, 92), (33, 92)]
[(34, 39), (35, 39)]
[(34, 42), (35, 42)]
[(34, 93), (35, 93), (36, 93)]
[(35, 97), (36, 97)]
[(38, 93), (39, 93), (40, 93), (41, 93)]
[(42, 34), (43, 34)]
[(42, 92), (43, 92), (44, 92), (45, 92), (46, 92)]
[(43, 76), (44, 76), (45, 76)]
[(44, 68), (45, 68)]
[(44, 81), (45, 81), (46, 81), (47, 81)]
[(44, 85), (45, 85)]
[(44, 98), (45, 98)]
[(45, 41), (46, 41)]
[(47, 85), (48, 85)]
[(47, 94), (48, 94), (49, 94), (50, 94)]
[(48, 42), (49, 42)]
[(50, 55), (51, 55)]
[(51, 40), (52, 40)]
[(53, 48), (54, 48)]
[(53, 55), (54, 55)]
[(53, 84), (54, 84)]
[(54, 35), (55, 35)]
[(55, 33), (56, 33)]
[(55, 85), (56, 85)]
[(56, 82), (57, 82)]
[(57, 46), (58, 46)]
[(59, 53), (60, 53)]
[(59, 80), (60, 80), (61, 80)]
[(60, 34), (61, 34)]
[(60, 72), (61, 72)]
[(60, 92), (61, 92)]
[(62, 51), (63, 51)]
[(64, 86), (65, 86), (66, 86), (67, 86), (68, 86), (69, 86)]
[(65, 50), (66, 50)]
[(65, 56), (66, 56)]
[(65, 62), (66, 62)]
[(65, 69), (66, 69)]
[(65, 93), (66, 93)]
[(67, 48), (68, 48), (69, 48), (70, 48)]
[(67, 102), (68, 102), (69, 102), (70, 102), (71, 102)]
[(67, 107), (68, 107)]
[(68, 54), (69, 54)]