A Report on

Development of Efficient Parallel Image Segmentation Algorithms

Submitted by

Rajat Tiwari
IV Sem B.Tech(ECE)

Under the guidance of

Dr.Sathya Peri
CSE Dept. (IIT Hyderabad)

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

CERTIFICATE

This is to certify that the project entitled **“Development of Efficient Parallel Image Segmentation Algorithm”** is a bonafide work carried out as a part of Summer Internship under my guidance by Rajat Tiwari of IV Sem B.Tech (ECE) at the Department of Computer and Science, Indian Institute of Technology, Hyderabad during June 2020.

Date: 4 July 2020

Signature of Instructor

ABSTRACT

Digital Image Processing supports research in fields like Image Segmentation and image recognization used with the various enhancing technologies.Image Segmentation is applied in processing and analyzing digital image partitions into multiple parts.It helps in separating foreground from background. Segmentation is prominent in computer vision and image analysis.Segmentation refers to the process of partitioning a digital image into multiple segments like image objects and sets of pixels. Segmentation can be easily understood as creating an array of the same pattern object and finding regularity for better understanding of images.As we know, a digital image is composed of pixels.

Edge Detection is an image processing technique for finding edges and boundaries of objects within an image. Edge Detection is a very well-developed field in Image Analysis. Edges typically correspond to points in the image where the gray value changes significantly from one pixel to the next. First order and second order methods are like kirsch, sobel, and canny to determine edges.

Watershed method is considered efficient and faster. The image is considered topographic. Pixels having the highest gradient magnitude intensities (GMIs) correspond to watershed lines, which represent the region boundaries. Water placed on any pixel enclosed by a common watershed line flows downhill to a common local intensity minimum (LIM). Pixels draining to a common minimum form a catch basin, which represents a segment.

PDE-based image segmentation techniques coupled level set methods and Fast Marching Methods to quickly and accurately extract boundaries from image data. They do not require prior knowledge about the number or topology of objects in the image data. For this purpose, the active contours and level set methods are commonly used, both exploiting the parametric curve evolution described by PDEs.

ACKNOWLEDGEMENT

A satisfactory completion of any significant task is the outcome of invaluable determination and dedication towards the task. I would therefore take the opportunity to thank and express our gratitude to all those for helping me to complete every task.

We extend our thanks to our Internship Coordinator, Dr.Sathya Peri ,Department of Computer Science and Engineering, IIT Hyderabad for his constant help and guidance to complete my internship. I would like to work more under your guidance for a long term approach.

RAJAT TIWARI

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1. Problem Statement

The aim of this project is to perform a comparative analysis of different types of image segmentation techniques and algorithms of different methods using python in serial programming.

Segmentation can be easily understood as creating an array of the same pattern object and finding regularity for better understanding of images.As we know, a digital image is composed of pixels.

Edge Detection is an image processing technique for finding edges and boundaries of objects within an image. Edge Detection is a very well-developed field in Image Analysis. Edges typically correspond to points in the image where the gray value changes significantly from one pixel to the next. First order and second order methods are like kirsch, sobel, and canny to determine edges.

Watershed method is considered efficient and faster. The image is considered topographic. Pixels having the highest gradient magnitude intensities (GMIs) correspond to watershed lines, which represent the region boundaries. Water placed on any pixel enclosed by a common watershed line flows downhill to a common local intensity minimum (LIM). Pixels draining to a common minimum form a catch basin, which represents a segment.

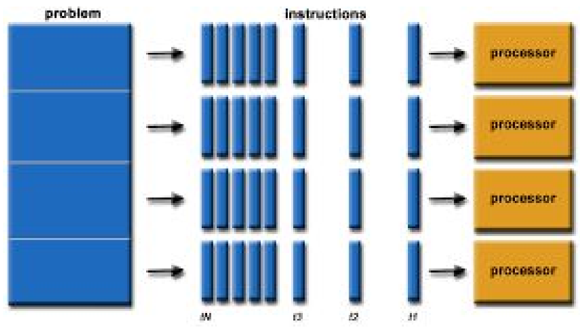
PDE-based image segmentation techniques coupled level set methods and Fast Marching Methods to quickly and accurately extract boundaries from image data. They do not require prior knowledge about the number or topology of objects in the image data. For this purpose, the active contours and level set methods are commonly used, both exploiting the parametric curve evolution described by PDEs.

1.2. Stacked Images

Focus Stacking is also known as focal plane merging, z-stacking and focus blending.It is a digital image processing technique which combine multiple images taken at different focal length to give a resulting image with a greater Depth of Field(DoF).It allows generation of images that are physically impossible.

1.3. Parallel Programming

It is the use of multiple processing elements simultaneously for solving any problem. Problems are broken down into instructions and are solved concurrently as each resource which has been applied to work is working at the same time. A process has many threads within it and only one thread is called master thread.

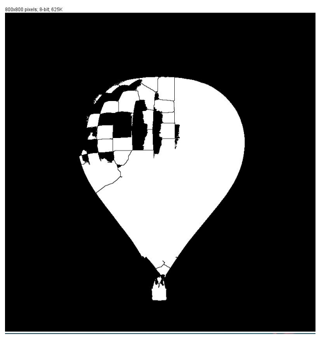


CHAPTER 2

IMAGE SEGMENTATION

In Image Analysis and Computer Vision, Image Segmentation is the process of dividing a digital image into multiple segments(i.e.Pixels).Segmentation means dividing images into different blocks based on certain characteristics like color,intensity,etc.We can convert many stacks into single images.Various algorithm are there to support image segmentation.It cluster pixels into salient image regions.

Image Segmentation is also involved in separating foreground with background. A segmentation could be used for object recognition, age regression, machine vision, medical imaging, video surveillance, image compression, image editing, or image database look-up.



Types of Image Segmentation:

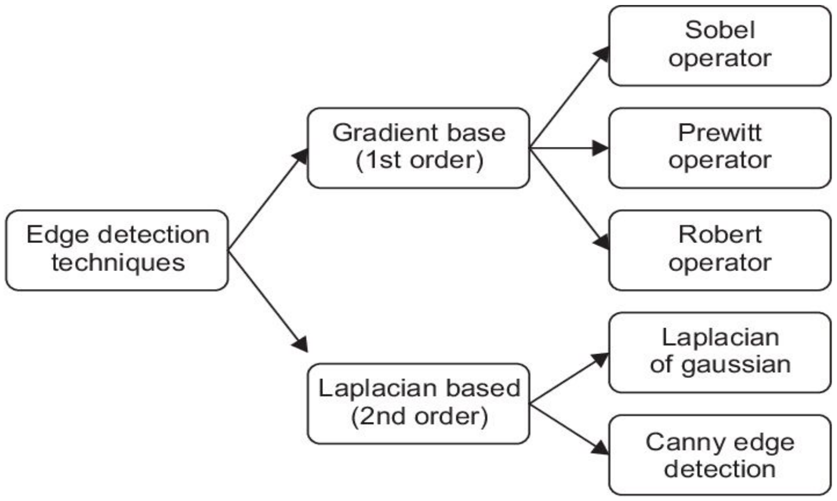
1. Threshold Method
2. Edge Detection Method
3. Region Based Method
4. Watershed Method
5. Clustering Method
6. Partial Differential Equation Based Method
7. Artificial Neural Network Based Method
8. Graph Partitioning Method

CHAPTER 3

EDGE DETECTION BASED

Edge Detection is a very well-developed field in Image Analysis. Edges typically correspond to points in the image where the gray value changes significantly from one pixel to the next. Thus, detecting Edges helps in extracting useful information characteristics of the image where there are abrupt changes.Edges are those points where there is a change in pixel density. Region boundaries and edges are closely related to each other and there is a sharp adjustment in intensity at the boundaries.

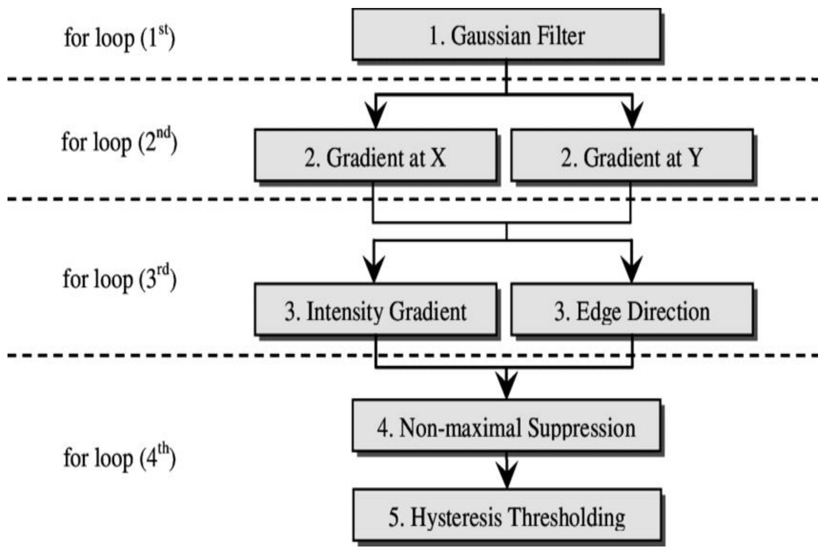
Edge detection is a process of locating an edge of an image. Detection of edges in an image is a very important step towards understanding image features. Edges consist of meaningful features and contain significant information. Since edges often occur at image locations representing object boundaries, edge detection is extensively used in image segmentation when images are divided into areas corresponding to different objects.



To improve visibility of those images, scaling has to be done .For small values, it has to be scaled up by appropriate factors. For large values, it has to be scaled down by appropriate factors.

3.1. Gradient Edge Based

The gradient method detect the edges by looking for the maximum and minimum in the first derivative of the image.



3.1.1. Robert Edge Detection Method

It uses diagonal edge gradients, susceptible to fluctuations. Gives no information about edge orientation and works best with binary images.The Roberts Cross operator performs a simple, quick to compute, 2-D spatial gradient measurement on an image. It thus highlights regions of high spatial frequency which often correspond to edges.

$$\begin{array}{|c|c|}\hline +1 & 0 \\ \hline 0 & -1 \\ \hline \end{array} \qquad \begin{array}{|c|c|}\hline 0 & +1 \\ \hline -1 & 0 \\ \hline \end{array}$$

Gx Gy

In its most common usage, the input to the operator is a grayscale image, as is the output. Pixel values at each point in the output represent the estimated absolute magnitude of the spatial gradient of the input image at that point.

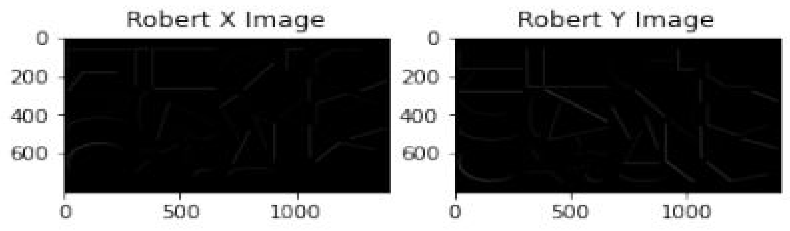
```
[1]: #import important libraries
     import cv2
     import numpy as np
     from matplotlib import pyplot as plt
```

```
[2]: #reading image and resizing and gray scale conversion
     image = cv2.imread('img.jpg',0)
     image = cv2.resize(image,(1400,800))
```

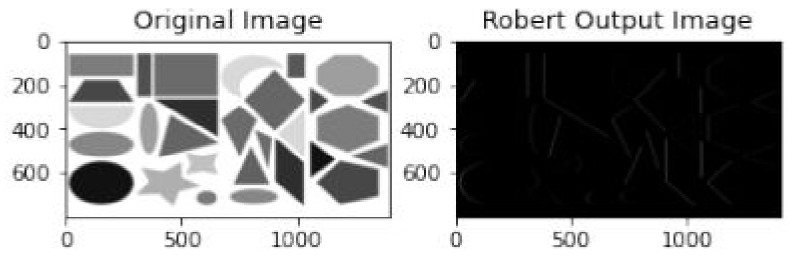
```
[3]:  # Robertsedge operator
     kernel_Roberts_x = np.array([
         [1, 0],
         [0, -1]
         ])
     kernel_Roberts_y = np.array([
         [0, -1],
         [1, 0]
         ])

     average_kernel = (kernel_Roberts_x+kernel_Roberts_y)
```

```
[4]: output_x = cv2.filter2D(image, -1, kernel_Roberts_x)
     output_y = cv2.filter2D(image, -1, kernel_Roberts_y)
```

```
[5]: # Show sharpening effect
     plt.subplot(121),plt.imshow(output_x,cmap = 'gray')
     plt.title('Robert X Image')
     plt.subplot(122),plt.imshow(output_y,cmap = 'gray')
     plt.title('Robert Y Image')
```

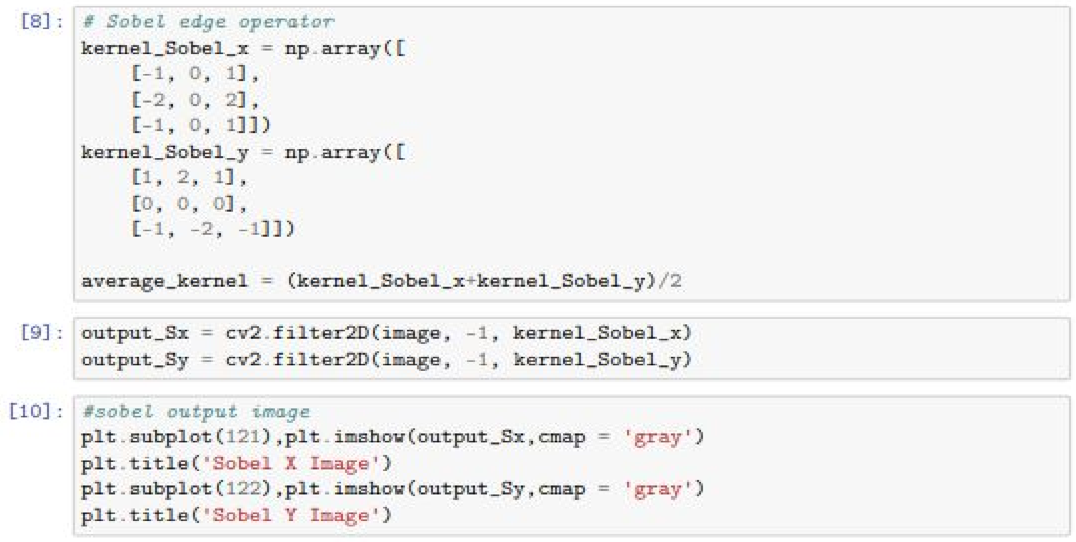


```
[7]: plt.subplot(121),plt.imshow(image,cmap = 'gray')
     plt.title('Original Image')
     plt.subplot(122),plt.imshow(output_Rav,cmap = 'gray')
     plt.title('Robert Output Image')
```



3.1.2. Sobel Edge Detection Method

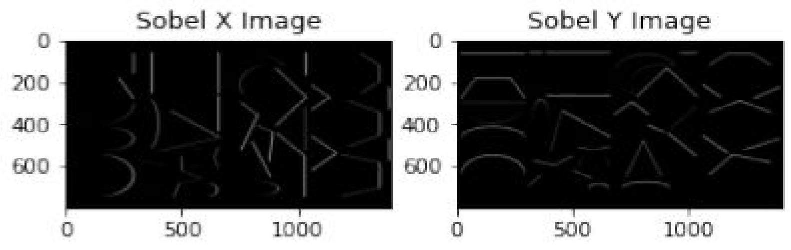
The Sobel operator performs a 2-D spatial gradient measurement on an image and so emphasizes regions of high spatial frequency that correspond to edges. Typically it is used to find the approximate absolute gradient magnitude at each point in an input grayscale image.

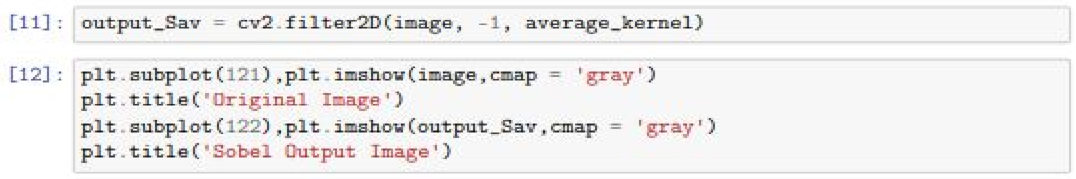


```
[8]:  # Sobel edge operator
      kernel_Sobel_x = np.array([
          [-1, 0, 1],
          [-2, 0, 2],
          [-1, 0, 1]])
      kernel_Sobel_y = np.array([
          [1, 2, 1],
          [0, 0, 0],
          [-1, -2, -1]])

      average_kernel = (kernel_Sobel_x+kernel_Sobel_y)/2
```

```
[9]:  output_Sx = cv2.filter2D(image, -1, kernel_Sobel_x)
      output_Sy = cv2.filter2D(image, -1, kernel_Sobel_y)
```

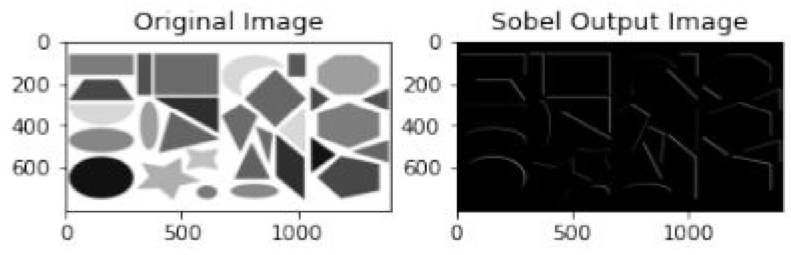
```
[10]: #sobel output image
      plt.subplot(121),plt.imshow(output_Sx,cmap = 'gray')
      plt.title('Sobel X Image')
      plt.subplot(122),plt.imshow(output_Sy,cmap = 'gray')
      plt.title('Sobel Y Image')
```





```
[11]: output_Sav = cv2.filter2D(image, -1, average_kernel)
```

```
[12]: plt.subplot(121),plt.imshow(image,cmap = 'gray')
      plt.title('Original Image')
      plt.subplot(122),plt.imshow(output_Sav,cmap = 'gray')
      plt.title('Sobel Output Image')
```



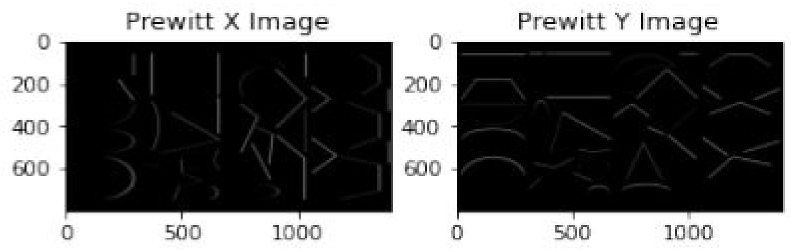
3.1.3. Prewitt Edge Detection Method

Prewitt operator is a method of edge detection in image processing which calculates the maximum response of a set of convolution kernels to find the local edge orientation for each pixel. The Prewitt edge detector is an appropriate way to estimate the magnitude and orientation of an edge.

```
[13]: # Prewitt edge operator
      kernel_Prewitt_x = np.array([
          [-1, 0, 1],
          [-1, 0, 1],
          [-1, 0, 1]])
      kernel_Prewitt_y = np.array([
          [1, 1, 1],
          [0, 0, 0],
          [-1, -1, -1]])

      average_kernel = kernel_Prewitt_x+kernel_Prewitt_y
```

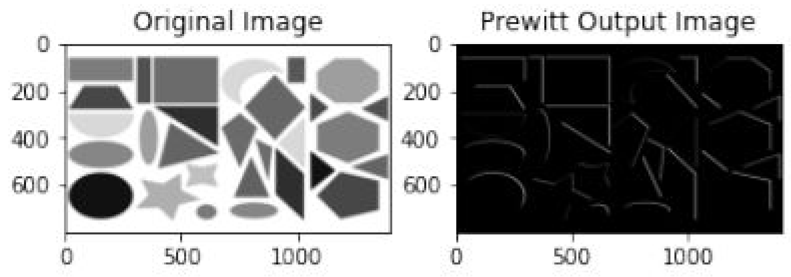
```
[14]: output_Px = cv2.filter2D(image, -1, kernel_Prewitt_x)
      output_Py = cv2.filter2D(image, -1, kernel_Prewitt_y)
```

```
[15]: plt.subplot(121),plt.imshow(output_Px,cmap = 'gray')
      plt.title('Prewitt X Image')
      plt.subplot(122),plt.imshow(output_Py,cmap = 'gray')
      plt.title('Prewitt Y Image')
```



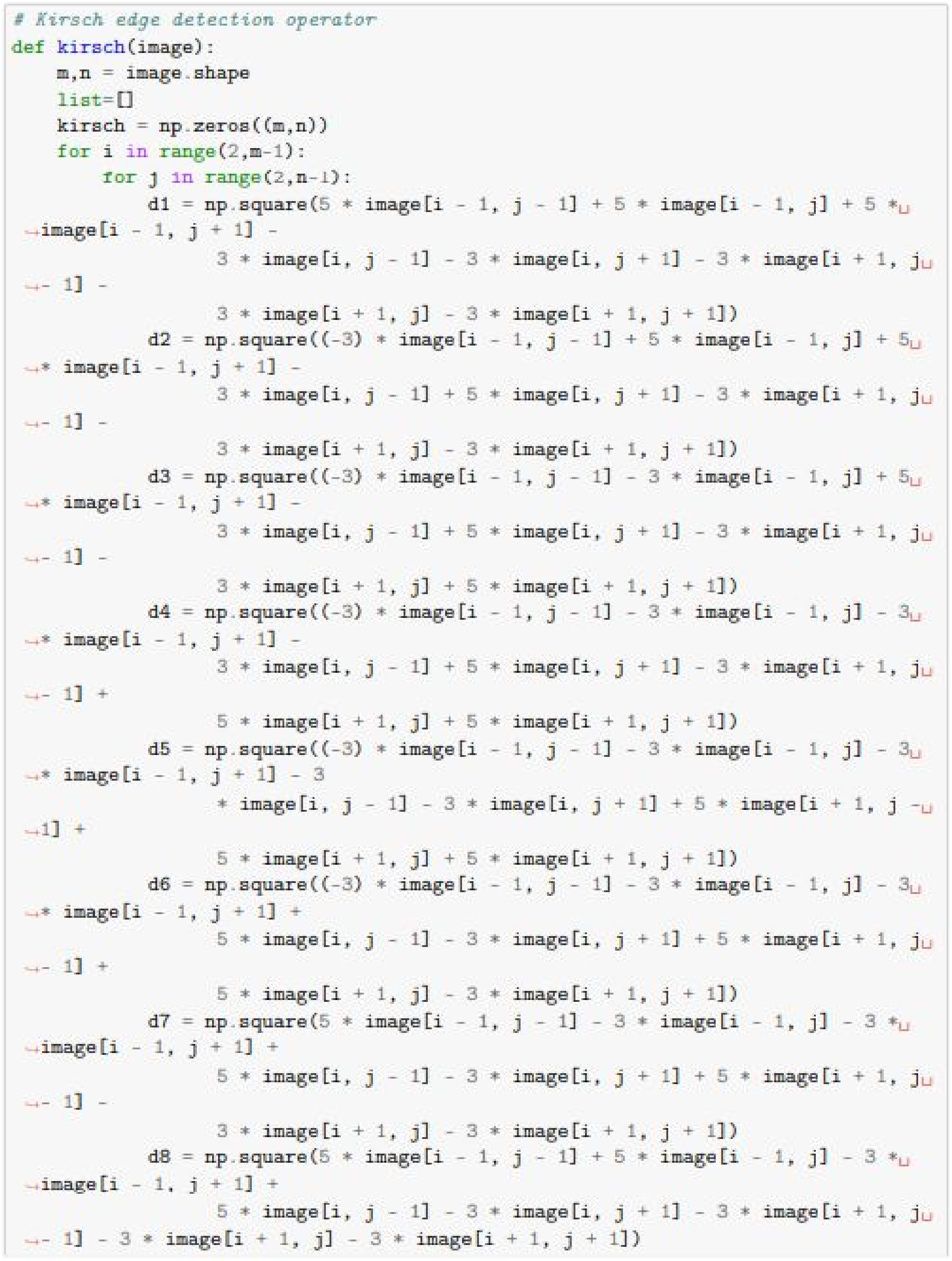
```
[16]: output_Pav = cv2.filter2D(image, -1, average_kernel)
```

```
[17]: plt.subplot(121),plt.imshow(image,cmap = 'gray')
      plt.title('Original Image')
      plt.subplot(122),plt.imshow(output_Pav,cmap = 'gray')
      plt.title('Prewitt Output Image')
```



3.1.4. Kirsch Edge Detection Method

Kirsch edge operators have been proposed for image segmentation of mammographic images.The kirsch edge detector detects edges using eight filters applied to the image with the maximum being retained for the final image. The eight filters are a rotation of a basic compass convolution filter.

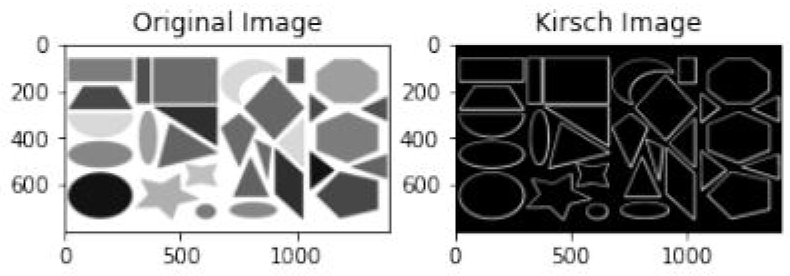


```
# Kirsch edge detection operator
def kirsch(image):
    m,n = image.shape
    list=[]
    kirsch = np.zeros((m,n))
    for i in range(2,m-1):
        for j in range(2,n-1):
            d1 = np.square(5 * image[i - 1, j - 1] + 5 * image[i - 1, j] + 5 *␣
↪image[i - 1, j + 1] -
                  3 * image[i, j - 1] - 3 * image[i, j + 1] - 3 * image[i + 1, j␣
↪- 1] -
                  3 * image[i + 1, j] - 3 * image[i + 1, j + 1])
            d2 = np.square((-3) * image[i - 1, j - 1] + 5 * image[i - 1, j] + 5␣
↪* image[i - 1, j + 1] -
                  3 * image[i, j - 1] + 5 * image[i, j + 1] - 3 * image[i + 1, j␣
↪- 1] -
                  3 * image[i + 1, j] - 3 * image[i + 1, j + 1])
            d3 = np.square((-3) * image[i - 1, j - 1] - 3 * image[i - 1, j] + 5␣
↪* image[i - 1, j + 1] -
                  3 * image[i, j - 1] + 5 * image[i, j + 1] - 3 * image[i + 1, j␣
↪- 1] -
                  3 * image[i + 1, j] + 5 * image[i + 1, j + 1])
            d4 = np.square((-3) * image[i - 1, j - 1] - 3 * image[i - 1, j] - 3␣
↪* image[i - 1, j + 1] -
                  3 * image[i, j - 1] + 5 * image[i, j + 1] - 3 * image[i + 1, j␣
↪- 1] +
                  5 * image[i + 1, j] + 5 * image[i + 1, j + 1])
            d5 = np.square((-3) * image[i - 1, j - 1] - 3 * image[i - 1, j] - 3␣
↪* image[i - 1, j + 1] - 3
                  * image[i, j - 1] - 3 * image[i, j + 1] + 5 * image[i + 1, j -␣
↪1] +
                  5 * image[i + 1, j] + 5 * image[i + 1, j + 1])
            d6 = np.square((-3) * image[i - 1, j - 1] - 3 * image[i - 1, j] - 3␣
↪* image[i - 1, j + 1] +
                  5 * image[i, j - 1] - 3 * image[i, j + 1] + 5 * image[i + 1, j␣
↪- 1] +
                  5 * image[i + 1, j] - 3 * image[i + 1, j + 1])
            d7 = np.square(5 * image[i - 1, j - 1] - 3 * image[i - 1, j] - 3 *␣
↪image[i - 1, j + 1] +
                  5 * image[i, j - 1] - 3 * image[i, j + 1] + 5 * image[i + 1, j␣
↪- 1] -
                  3 * image[i + 1, j] - 3 * image[i + 1, j + 1])
            d8 = np.square(5 * image[i - 1, j - 1] + 5 * image[i - 1, j] - 3 *␣
↪image[i - 1, j + 1] +
                  5 * image[i, j - 1] - 3 * image[i, j + 1] - 3 * image[i + 1, j␣
↪- 1] - 3 * image[i + 1, j] - 3 * image[i + 1, j + 1])
```

```
# : Take the maximum value in each direction, the effect is not good, use␣
↪another method
            list=[d1, d2, d3, d4, d5, d6, d7, d8]
            kirsch[i,j]= int(np.sqrt(max(list)))
# : Rounding the die length in all directions
#kirsch[i, j] =int(np.sqrt(d1+d2+d3+d4+d5+d6+d7+d8))
    for i in range(m):
        for j in range(n):
            if kirsch[i,j]>127:
                kirsch[i,j]=255
            else:
                kirsch[i,j]=0
    return kirsch
```

```
[19]: output_K = kirsch(image)
```

```
[20]: plt.subplot(121),plt.imshow(image,cmap = 'gray')
      plt.title('Original Image')
      plt.subplot(122),plt.imshow(output_K,cmap = 'gray')
      plt.title('Kirsch Image')
```



3.2. Laplacian Edge Detection

The Laplacian method searches for zero crossings in the second derivative of the image to find edges.
Laplacian is a derivative operator; its uses highlight gray level discontinuities in an image and try to de-emphasize regions with slowly varying gray levels. This operation in result produces such images which have grayish edge lines and other discontinuities on a dark background. This produces inward and outward edges in an image.

0	-1	0
-1	4	-1
0	-1	0

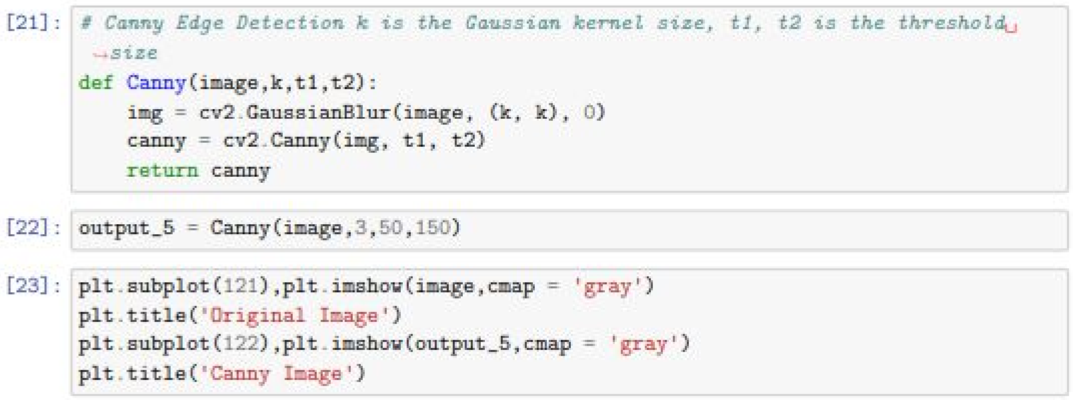
The laplacian operator

-1	-1	-1
-1	8	-1
-1	-1	-1

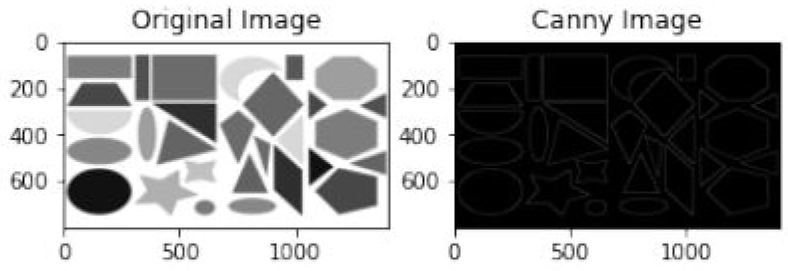
The laplacian operator
(include diagonals)

3.2.1. Canny Edge Detection

Canny edge detection is a multi-step algorithm that can detect edges with noise suppressed at the same time.Canny edge detection is a technique to extract useful structural information from different vision objects and dramatically reduce the amount of data to be processed.
Canny edge detection algorithm is one of the most strictly defined methods that provides good and reliable detection. Owing to its optimality to meet the three criteria for edge detection and the simplicity of process for implementation, it became one of the most popular algorithms for edge detection.

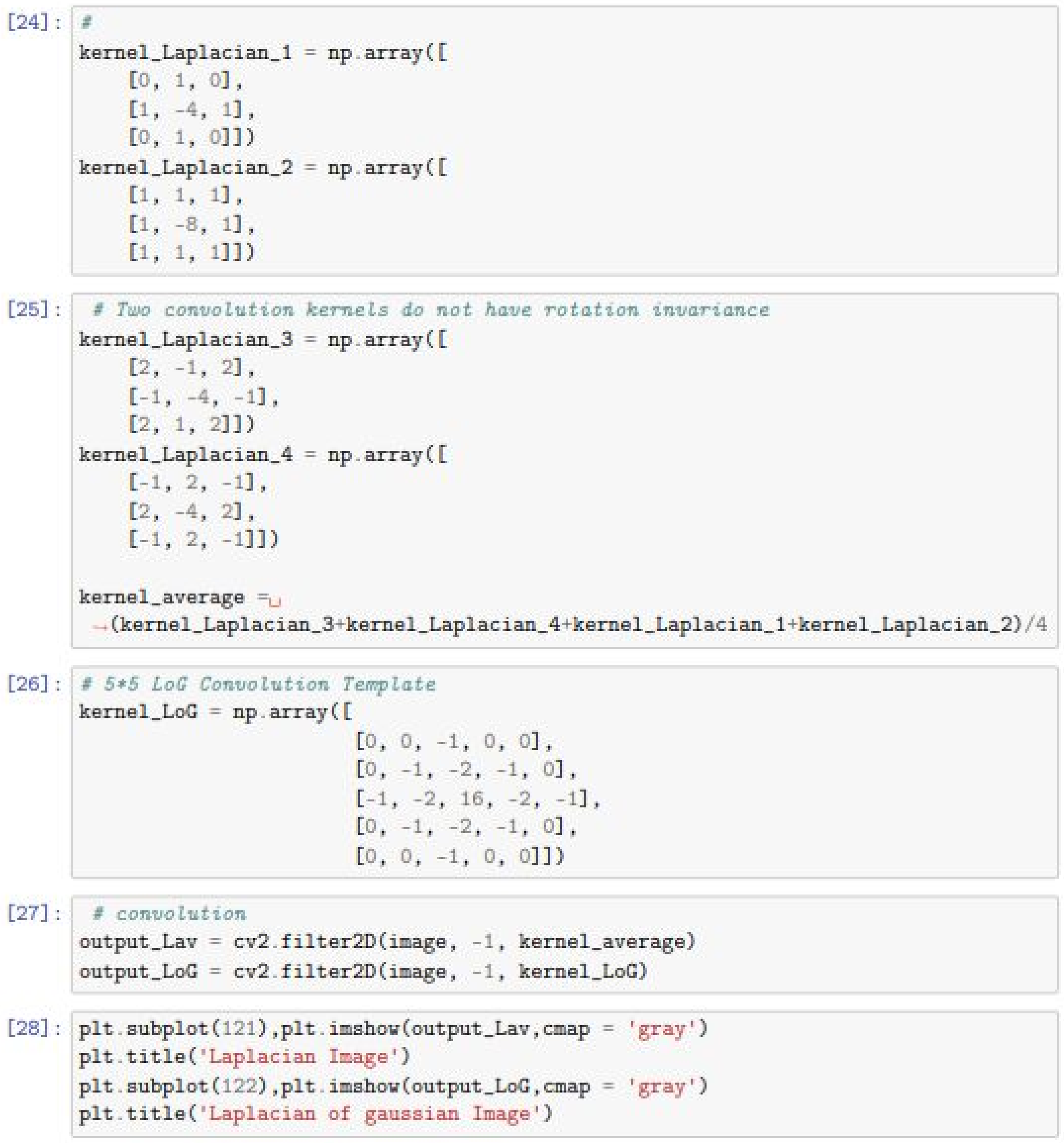


```
[21]: # Canny Edge Detection k is the Gaussian kernel size, t1, t2 is the threshold␣
        ↪size
      def Canny(image,k,t1,t2):
          img = cv2.GaussianBlur(image, (k, k), 0)
          canny = cv2.Canny(img, t1, t2)
          return canny

[22]: output_5 = Canny(image,3,50,150)

[23]: plt.subplot(121),plt.imshow(image,cmap = 'gray')
      plt.title('Original Image')
      plt.subplot(122),plt.imshow(output_5,cmap = 'gray')
      plt.title('Canny Image')
```



3.2.4. Laplacian of Gaussian Edge Detection

The Laplacian edge detector uses only one kernel. It calculates second order derivatives in a single pass.



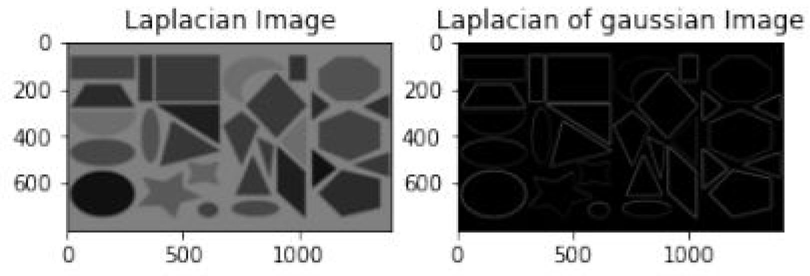
```
[24]: #
kernel_Laplacian_1 = np.array([
    [0, 1, 0],
    [1, -4, 1],
    [0, 1, 0]])
kernel_Laplacian_2 = np.array([
    [1, 1, 1],
    [1, -8, 1],
    [1, 1, 1]])
```

```
[25]:  # Two convolution kernels do not have rotation invariance
kernel_Laplacian_3 = np.array([
    [2, -1, 2],
    [-1, -4, -1],
    [2, 1, 2]])
kernel_Laplacian_4 = np.array([
    [-1, 2, -1],
    [2, -4, 2],
    [-1, 2, -1]])

kernel_average =␣
 →(kernel_Laplacian_3+kernel_Laplacian_4+kernel_Laplacian_1+kernel_Laplacian_2)/4
```

```
[26]: # 5*5 LoG Convolution Template
kernel_LoG = np.array([
                      [0, 0, -1, 0, 0],
                      [0, -1, -2, -1, 0],
                      [-1, -2, 16, -2, -1],
                      [0, -1, -2, -1, 0],
                      [0, 0, -1, 0, 0]])
```

```
[27]:  # convolution
output_Lav = cv2.filter2D(image, -1, kernel_average)
output_LoG = cv2.filter2D(image, -1, kernel_LoG)
```

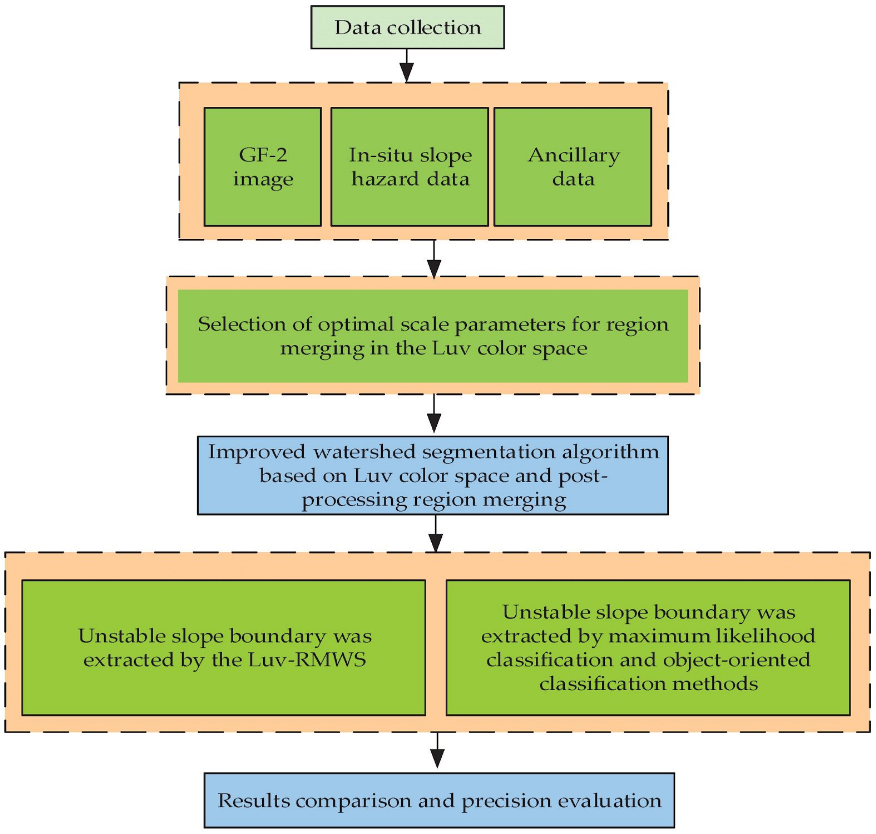
```
[28]: plt.subplot(121),plt.imshow(output_Lav,cmap = 'gray')
plt.title('Laplacian Image')
plt.subplot(122),plt.imshow(output_LoG,cmap = 'gray')
plt.title('Laplacian of gaussian Image')
```



CHAPTER 4

WATERSHED SEGMENTATION

This method is considered efficient and faster.The image is considered as topographic. Pixels having the highest gradient magnitude intensities (GMIs) correspond to watershed lines, which represent the region boundaries. Water placed on any pixel enclosed by a common watershed line flows downhill to a common local intensity minimum (LIM). Pixels draining to a common minimum form a catch basin, which represents a segment. It is appropriate to use this method to segment the high-resolution remote sensing image.


Data collection
GF-2 image
In-situ slope hazard data
Ancillary data
Selection of optimal scale parameters for region merging in the Luv color space
Improved watershed segmentation algorithm based on Luv color space and post-processing region merging
Unstable slope boundary was extracted by the Luv-RMWS
Unstable slope boundary was extracted by maximum likelihood classification and object-oriented classification methods
Results comparison and precision evaluation

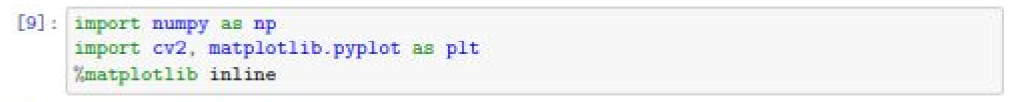
Types of Watershed Method:

1. Distant Transform Method
2. Gradient Method
3. Marker Controlled Method

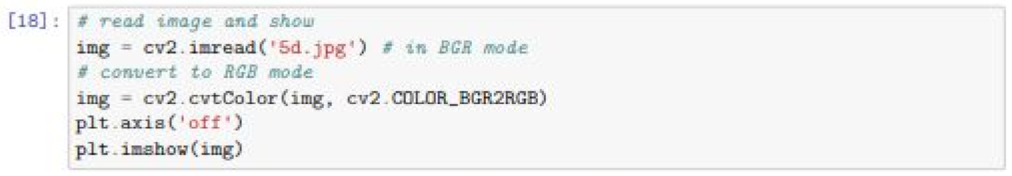
4.1. Distant Transform Approach

A tool used commonly in conjunction with the watershed transform for segmentation is the distance transform. It is the distance from every pixel to the nearest non zero-valued pixel. The distance transform can be computed using toolbox function bwdist, whose calling syntax is

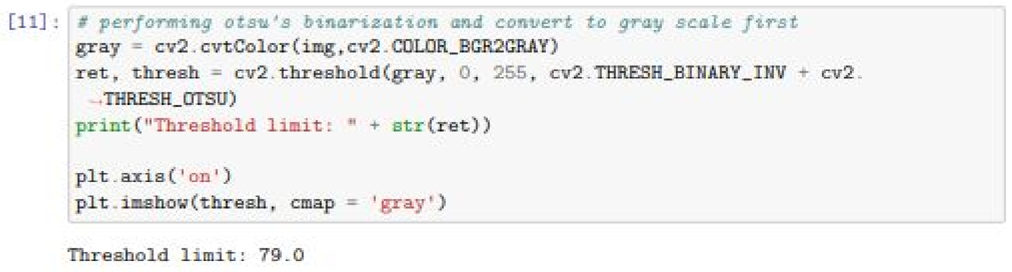
D=bwdist(f)



```
[9]: import numpy as np
     import cv2, matplotlib.pyplot as plt
     %matplotlib inline
```

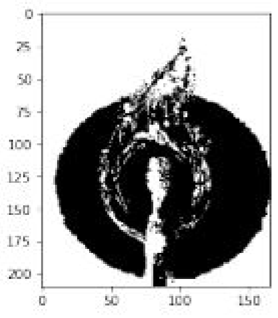


```
[18]: # read image and show
      img = cv2.imread('5d.jpg') # in BGR mode
      # convert to RGB mode
      img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
      plt.axis('off')
      plt.imshow(img)
```



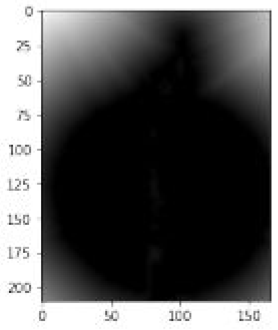
```
[11]: # performing otsu's binarization and convert to gray scale first
      gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
      ret, thresh = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY_INV + cv2.
       →THRESH_OTSU)
      print("Threshold limit: " + str(ret))

      plt.axis('on')
      plt.imshow(thresh, cmap = 'gray')
```

```
Threshold limit: 79.0
```

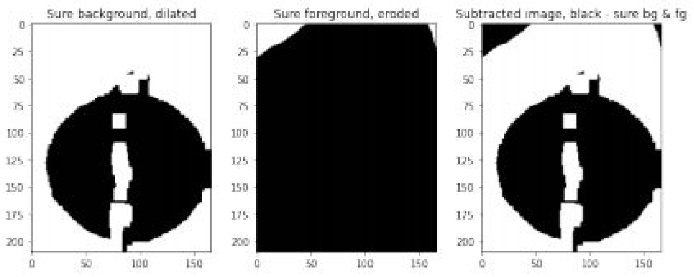


```
[12]: # noise removal
      kernel = np.ones((3, 3), np.uint8)
      opening = cv2.morphologyEx(thresh, cv2.MORPH_OPEN, kernel, iterations = 2)
      # sure background area
      sure_bg = cv2.dilate(opening, kernel, iterations = 3)
      # sure foreground area
      dist_transform = cv2.distanceTransform(opening,cv2.DIST_L2,5)
      ret, sure_fg = cv2.threshold(dist_transform,0.7*dist_transform.max(),255,0)

      # Finding unknown region
      sure_fg = np.uint8(sure_fg)
      unknown = cv2.subtract(sure_bg,sure_fg)
```

```
[13]: plt.imshow(dist_transform, cmap = 'gray')
```

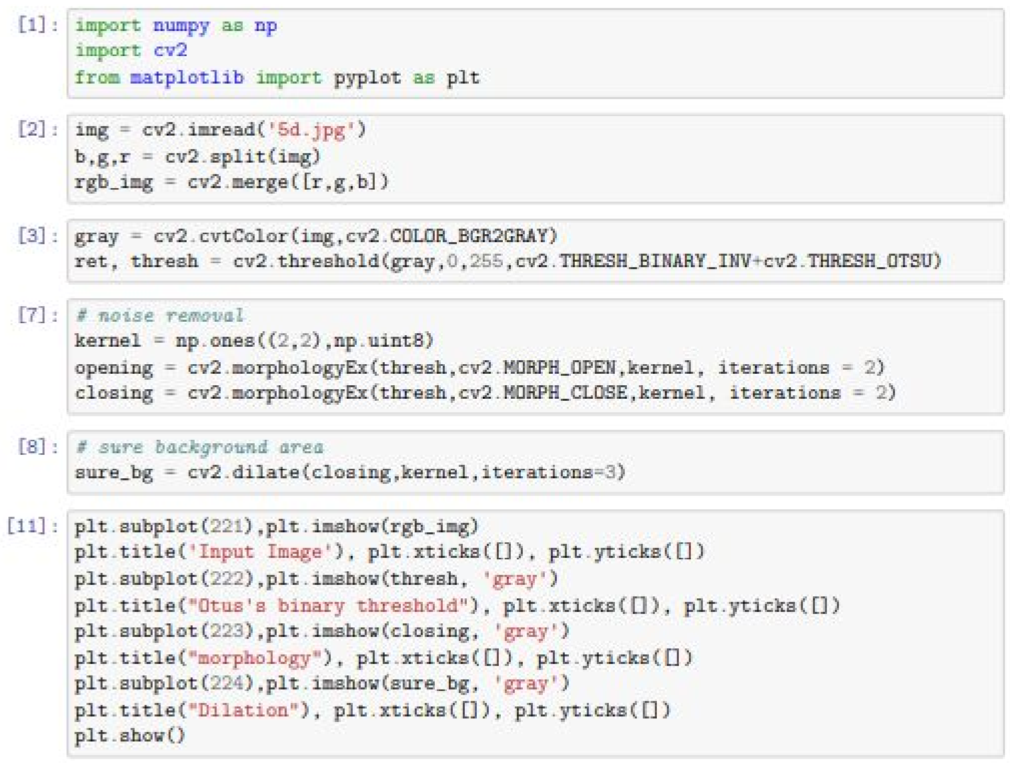


```
[15]: fig = plt.figure(figsize = (10, 5)) # to change figsize
      plt.subplot(131)
      plt.imshow(sure_bg, cmap = 'gray')
      plt.title('Sure background, dilated')

      plt.subplot(132)
      plt.imshow(sure_fg, cmap = 'gray')
      plt.title('Sure foreground, eroded')

      plt.subplot(133)
      plt.imshow(unknown, cmap = 'gray')
      plt.title('Subtracted image, black - sure bg & fg')
      plt.tight_layout()

      # plt.subplots_adjust(wspace = 3)
      # fine tuning
      # f.subplots_adjust(wspace=3)
```

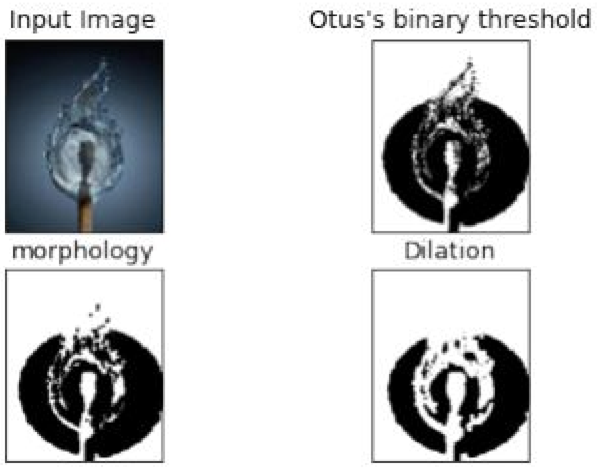


4.2. Gradient Method

The gradient magnitude is used to pre process a gray-scale image prior to using the watershed transform for segmentation. The gradient magnitude image has high pixel values along object edges and low pixel values everywhere else.
It splits the segmentation process into two separate steps: first we detect the main edges of the image processed, and then we compute the watershed of the gradient detected.



```
[1]: import numpy as np
     import cv2
     from matplotlib import pyplot as plt

[2]: img = cv2.imread('5d.jpg')
     b,g,r = cv2.split(img)
     rgb_img = cv2.merge([r,g,b])

[3]: gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
     ret, thresh = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)

[7]: # noise removal
     kernel = np.ones((2,2),np.uint8)
     opening = cv2.morphologyEx(thresh,cv2.MORPH_OPEN,kernel, iterations = 2)
     closing = cv2.morphologyEx(thresh,cv2.MORPH_CLOSE,kernel, iterations = 2)

[8]: # sure background area
     sure_bg = cv2.dilate(closing,kernel,iterations=3)

[11]: plt.subplot(221),plt.imshow(rgb_img)
      plt.title('Input Image'), plt.xticks([]), plt.yticks([])
      plt.subplot(222),plt.imshow(thresh, 'gray')
      plt.title("Otus's binary threshold"), plt.xticks([]), plt.yticks([])
      plt.subplot(223),plt.imshow(closing, 'gray')
      plt.title("morphology"), plt.xticks([]), plt.yticks([])
      plt.subplot(224),plt.imshow(sure_bg, 'gray')
      plt.title("Dilation"), plt.xticks([]), plt.yticks([])
      plt.show()
```



4.3. Marker Controlled Method

The marker-controlled watershed segmentation has been shown to be a robust and flexible method for segmentation of objects with closed contours, where the boundaries are expressed as ridges.
Markers are placed inside an object of interest; internal markers associate with objects of interest, and external markers associate with the background. After segmentation, the boundaries of the watershed regions are arranged on the desired ridges, thus separating each object from its neighbours.

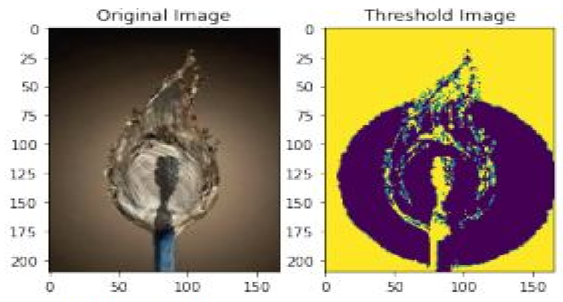
```
[1]: import cv2
     import numpy as np
     from matplotlib import pyplot as plt
```

```
[2]: def showImage(imageForShowing):
         plt.figure(figsize = (2.5, 2.5))
         plt.imshow(imageForShowing)
```

```
[3]: im = cv2.imread("5d.jpg", cv2.IMREAD_GRAYSCALE)
     original = cv2.imread("5d.jpg")
```

```
[4]: _, im = cv2.threshold(im, 72, 255, cv2.THRESH_BINARY_INV)
     #ret, im = cv2.threshold(im, 100, 255,cv2.THRESH_BINARY_INV+cv2.THRESH_OTSU)
     #ret, im = cv2.threshold(im, 100, 255, cv2.THRESH_OTSU)
```

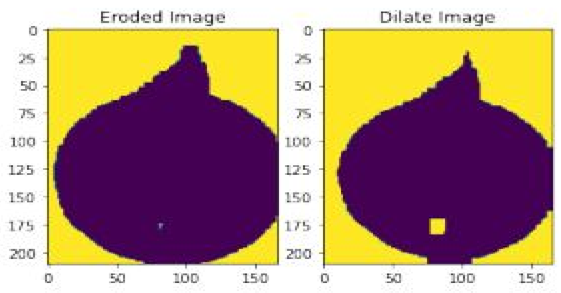
```
[5]: plt.subplot(121),plt.imshow(original)
     plt.title('Original Image')
     plt.subplot(122),plt.imshow(im)
     plt.title('Threshold Image')
```



```
[6]: kernel = np.ones((10,10),np.uint8)
     im = cv2.erode(im, kernel)
```

```
[7]: kernel = np.ones((10,10),np.uint8)
     dilateImage = cv2.dilate(im, kernel, iterations = 1)
```

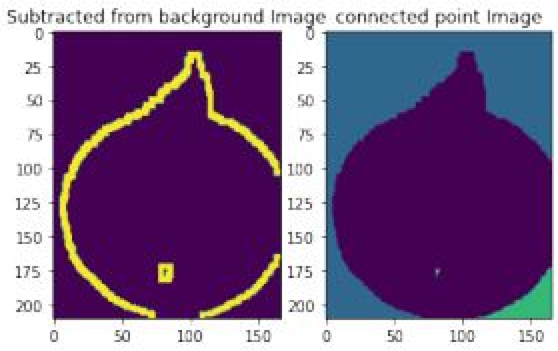
```
[8]: plt.subplot(121),plt.imshow(im)
     plt.title('Eroded Image')
     plt.subplot(122),plt.imshow(dilateImage)
     plt.title('Dilate Image')
```



```
[9]: unknown = cv2.subtract(dilateImage,im)
```

```
[10]: _, markers = cv2.connectedComponents(im)
```

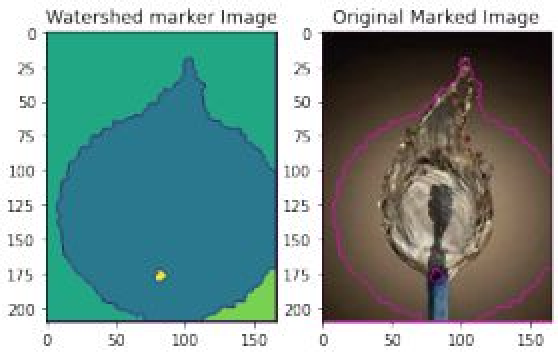
```
[11]: plt.subplot(121),plt.imshow(unknown)
      plt.title('Subtracted from background Image')
      plt.subplot(122),plt.imshow(markers)
      plt.title('connected point Image')
```



```
[12]: markers = markers + 1
      markers[unknown==255] = 0
```

```
[13]: markers = cv2.watershed(original,markers)
      original[markers== -1] = [250, 0, 210]
```

```
[14]: plt.subplot(121),plt.imshow(markers)
      plt.title('Watershed marker Image')
      plt.subplot(122),plt.imshow(original)
      plt.title('Original Marked Image')
```



CHAPTER 5

<u>PARTIAL DIFFERENTIATION EQUATION BASED</u>

PDE-based image segmentation techniques coupled level set methods and Fast Marching Methods to quickly and accurately extract boundaries from image data. They do not require prior knowledge about the number or topology of objects in the image data.

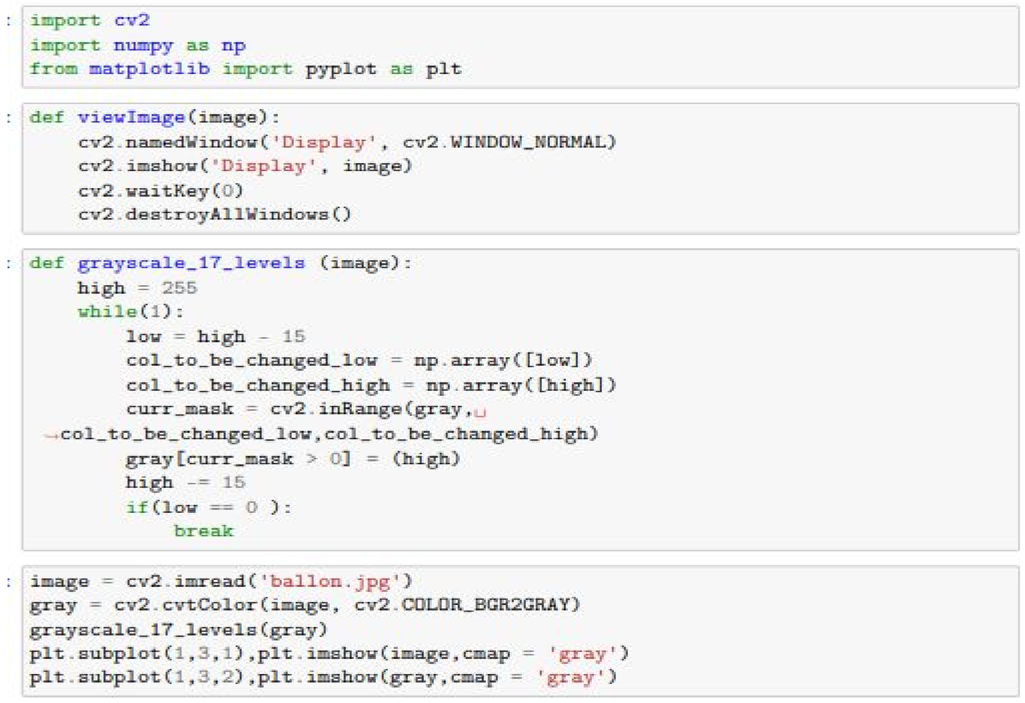
For this purpose, the active contours and level set methods are commonly used, both exploiting the parametric curve evolution described by PDEs. The forces derived from the image cause a curve change according to the segmented objects.

Types of PDE:

1. **Lagrangian Method**
2. **Level set Method**
3. **Active Counter Method**

5.1. Lagrangian Method

These are based on parameterizing the counters according to sampling strategy and evolving each element according to image.

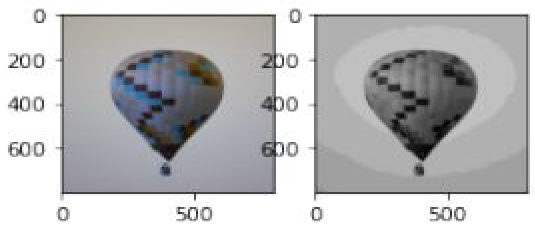


```
: import cv2
  import numpy as np
  from matplotlib import pyplot as plt
```

```
: def viewImage(image):
      cv2.namedWindow('Display', cv2.WINDOW_NORMAL)
      cv2.imshow('Display', image)
      cv2.waitKey(0)
      cv2.destroyAllWindows()
```

```
: def grayscale_17_levels (image):
      high = 255
      while(1):
          low = high - 15
          col_to_be_changed_low = np.array([low])
          col_to_be_changed_high = np.array([high])
          curr_mask = cv2.inRange(gray,␣
   →col_to_be_changed_low,col_to_be_changed_high)
          gray[curr_mask > 0] = (high)
          high -= 15
          if(low == 0 ):
              break
```

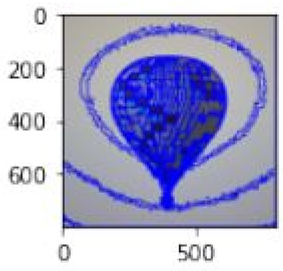
```
: image = cv2.imread('ballon.jpg')
  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
  grayscale_17_levels(gray)
  plt.subplot(1,3,1),plt.imshow(image,cmap = 'gray')
  plt.subplot(1,3,2),plt.imshow(gray,cmap = 'gray')
```



```
: def get_area_of_each_gray_level(im):
## convert image to gray scale (must br done before contouring)
    image = cv2.cvtColor(im, cv2.COLOR_BGR2GRAY)
    output = []
    high = 255
    first = True
    while(1):
      low = high - 15;
      if(first == False):
          to_be_black_again_low = np.array([high])
          to_be_black_again_high = np.array([255])
          curr_mask = cv2.inRange(image, to_be_black_again_low,
          to_be_black_again_high)
          image[curr_mask > 0] = (0)

        # making values of this gray level white so we can calculate
        # it's area
      ret, threshold = cv2.threshold(image, low, 255, 0)
      contours, hirerchy = cv2.findContours(threshold,
      cv2.RETR_LIST, cv2.CHAIN_APPROX_NONE)
      if(len(contours) > 0):
        output.append([cv2.contourArea(contours[0])])
        cv2.drawContours(im, contours, -1, (0,0,255), 3)
      high -= 15
      first = False
      if(low == 0 ):
        break
    return output
```

```
: image = cv2.imread('ballon.jpg')
print(get_area_of_each_gray_level(image))
plt.subplot(1,3,1),plt.imshow(image,cmap = 'gray')
```

```
[[0.0], [0.0], [0.0], [0.0], [0.0], [0.0], [7.0], [2.0], [0.0], [0.0], [0.0],
[0.0], [0.0], [0.0], [0.0], [19.0], [0.0]]
```



5.2. Level Set Method

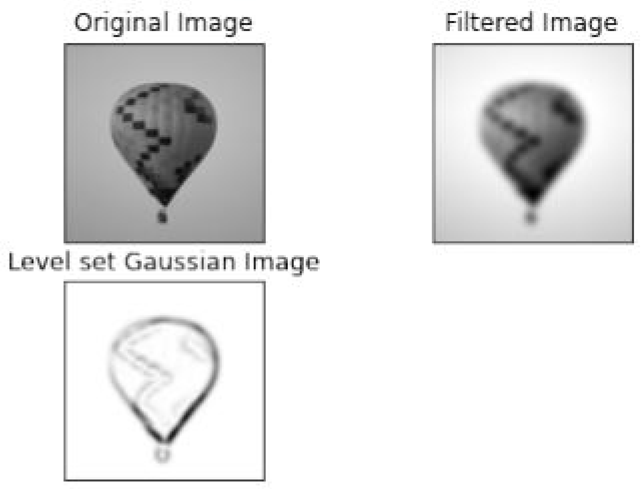
The Level-set method is simple and adaptable for computing and analyzing the motion of an interface in two or three dimensions. In Level-set, curve and surfaces are represented in implicit form as zero level sets of High Definition Continuous Function.

```
import numpy as np
import scipy.ndimage
import scipy.signal
import matplotlib.pyplot as plt
from skimage import color, io
```

```
def grad(x):
    return np.array(np.gradient(x))
def norm(x, axis=0):
    return np.sqrt(np.sum(np.square(x), axis=axis))
def stopping_fun(x):
    return 1. / (1. + norm(grad(x))**2)
```

```
img = io.imread('ballon.jpg')
img = color.rgb2gray(img)
img = img - np.mean(img)

img_smooth = scipy.ndimage.filters.gaussian_filter(img, sigma=15)

F = stopping_fun(img_smooth)
```

```
plt.subplot(221),plt.imshow(img,cmap = 'gray')
plt.title('Original Image'), plt.xticks([]), plt.yticks([])
plt.subplot(222),plt.imshow(img_smooth,cmap = 'gray')
plt.title('Filtered Image'), plt.xticks([]), plt.yticks([])
plt.subplot(223),plt.imshow(F,cmap = 'gray')
plt.title('Level set Gaussian Image'), plt.xticks([]), plt.yticks([])
```



5.3. Active Counter Method

The HAC combines the contour and region-based criteria into one embedded functional that includes regularization terms as well. The PDE describing the level set evolution was obtained using the Euler-Lagrange method.

```
import numpy as np
import matplotlib.pyplot as plt
import skimage.data as data
import skimage.segmentation as seg
import skimage.filters as filters
import skimage.draw as draw
import skimage.color as color
```

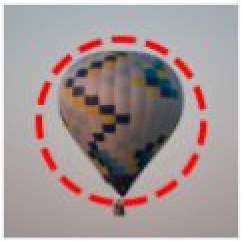
```
def image_show(image, nrows=1, ncols=1, cmap='gray'):
    fig, ax = plt.subplots(nrows=nrows, ncols=ncols, figsize=(2, 2))
    ax.imshow(image, cmap='gray')
    ax.axis('off')
    return fig, ax
```

```
# import the image
from skimage import io
image = io.imread('ballon.jpg')
image_show(image);
```

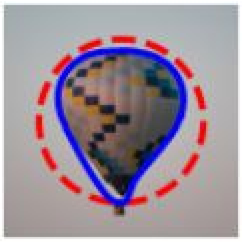
```
image_gray = color.rgb2gray(image)
image_show(image_gray);
```

```
: def circle_points(resolution, center, radius):
      """ Generate points which define a circle on an image.Centre refers to the␣
  ↪centre of the circle """
      radians = np.linspace(0, 2*np.pi, resolution)
      c = center[1] + radius*np.cos(radians)#polar co-ordinates
      r = center[0] + radius*np.sin(radians)

      return np.array([c, r]).T
  # Exclude last point because a closed path should not have duplicate points
  points = circle_points(200, [400, 400], 280)[:-3]
```

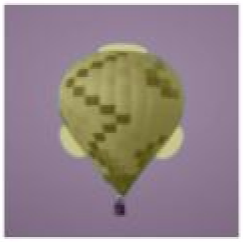
```
: fig, ax = image_show(image)
  ax.plot(points[:, 0], points[:, 1], '--r', lw=3)
```



```
: snake = seg.active_contour(image_gray, points)
  fig, ax = image_show(image)
  ax.plot(points[:, 0], points[:, 1], '--r', lw=3)
  ax.plot(snake[:, 0], snake[:, 1], '-b', lw=3);
```



```
image_segmented = seg.random_walker(image_gray, image_labels, beta = 3000)
# Check our results
fig, ax = image_show(image_gray)
ax.imshow(image_segmented == 1, alpha=0.3);
```



CHAPTER 6

<u>CONCLUSIONS</u>

From Edge Based Segmentation it can be concluded that Canny Method From second order differential equation has good effects but kirsch method is the best method in detecting edges of an object.

In watershed method, Marker controlled method is best for segmentation method. It provides full detail of the image. Distant Approach Method is also comparatively better than the gradient method.

In Partial differential Equation based method, Active counter is the best method of lagrangian method. Level sets work well on certain conditions.