A Study on Two Superpixel Generation Algorithms for Transrectal Ultrasound Images: SLIC, and SEEDS

Yiyou Sun
dept.EEMCS, University of Twente, Enschede, Netherlands
email address: y.sun-4@student.utwente.nl

***Abstract*—This project implements and investigates the performance of two superpixel generation algorithms, namely, Simple Linear Iterative Clustering (SLIC), and Superpixels Extracted via Energy-Driven Sampling (SEEDS), for 3D Transrectal Ultrasound (TRUS) images. Superpixel algorithms play a crucial role in image segmentation and feature extraction tasks. In this research, we compare and evaluate the accuracy, compactness, and computational complexity of these three algorithms in generating superpixels from TRUS images. The results of this study will provide valuable insights into choosing the most suitable algorithm for superpixel-based applications in TRUS image analysis**

***Index Terms*—Superpixel, Supervoxel, SLIC, SEEDS, Transrectal Ultrasound**

I. Introduction

Superpixel generation algorithms play a pivotal role in image processing and computer vision applications, enabling efficient image segmentation and feature extraction [1]–[3].

This study presents a comprehensive investigation and implementation of two prominent superpixel generation algorithms: Simple Linear Iterative Clustering (SLIC) [**?**], and Superpixels Extracted via Energy-Driven Sampling (SEEDS) [4]. The selected algorithms are renowned for their efficiency, simplicity, and ability to produce compact and uniform superpixels. These two methods for 3D input have received relatively little attention. The primary goal of this research is to compare the performance and assess the suitability of these algorithms for superpixel-based applications in the context of TRUS 3D image analysis.

In this implementation using Python, a practical and accessible framework is provided for generating superpixels from TRUS images. By implementing these algorithms in Python, we facilitate easy integration into existing medical image processing pipelines and allow researchers and practitioners to readily apply these techniques to their specific applications.

Through evaluation and comparison of the two algorithms, we seek to identify their strengths and weaknesses in handling TRUS images. Moreover, we will assess the computational complexity and runtime performance of each algorithm to ensure practical feasibility in real-world medical imaging scenarios.

II. Literature Review

A. Superpixel

The concept of superpixel was first introduced in [5] researching a classification model for segmentation. Since then, superpixel generation algorithms are widely explored. Adjacent pixels are combined to form superpixels using information such as gray and color scale, texture changes, and so on. It makes sure that the characteristics of pixels are consistent within one superpixel [6].

B. Transrectal Ultrasound

Transrectal ultrasound (TRUS) is a commonly available and non-invasive technique that employs high-frequency sound waves to generate detailed video images of the prostate gland. This study focus on the brightness mode ultrasound (B-mode) scans. The B-mode technique uses sound wave reflections to identify cancerous tissue exhibiting cellular structure abnormalities, manifesting as hypoechoic nodules on grayscale ultrasound. However, this technique faces inherent challenges in accurately distinguishing between cancerous and non-cancerous tissues, as there is a possibility that cancerous tissues also appear as isoechoic or hyperechoic nodules, just like healthy parenchyma does. And its annotation (ground truth) is gained from biopsy and 3D reconstruction, so segmentation for TRUS is always a challenging task.

III. Methods

This section explains the technical details of SLIC, and SEEDS implementation for 3D grayscale input images. [7] [4]

A. Simple Linear Iterative Clustering (SLIC)

SLIC segments images into supervoxels with comparable color and texture which combines the concept of K-means methods and superpixel. Compared to traditional K-means methods, SLIC limits the search space to reduce the calculation complexity to be linear in the value of pixel numbers N. Moreover, SLIC uses distance measures as dissimilarity functions of voxels containing color and spatial differences. In grayscale images, color differences are simplified to grayscale differences.

By default, only one parameter K is considered which controls the number of equally-sized supervoxels. To start with, the initialization of cluster centers selects k random seeds in input image I. The kth cluster center is described as vector $[r_k, g_k, b_k, x_k, y_k, z_k]$ where first three dimension record the color distribution. In 3D grayscale images, the vector is simplified into $[g_k, x_k, y_k, z_k]$. The grid size of clusters is determined by $\sqrt[3]{N/K}$ where N denotes the voxel volume of the whole input. To make sure cluster centers are not located on the edge or noisy voxel, they are updated in each iteration to the lowest gradient position within a supervoxel. Then, each voxel of the input image is assigned to the nearest cluster using distance measures. The searching area for each voxel is limited to a 2S×2S×2S region, which significantly reduced the calculation complexity of the dissimilarity function. After each voxel has been assigned to the nearest supervoxel cluster, the centers of the clusters are updated by finding the mean vector $[g, x, y, z]$ of all voxels belonging to the cluster.

Four-dimensional Euclidean distance in $gxyz$ space as distance measure will lead to inconsistent clusters. So, one crucial step in measuring the distances is to normalize the grayscale distance and spatial distances dividing their respective maximum distances within clusters.

$$\begin{aligned} d_c &= \sqrt{(g_j - g_i)^2}. \\ d_s &= \sqrt{(x_j - x_i)^2 + (y_j - y_i)^2 + (z_j - z_i)^2}. \\ D &= \sqrt{(\frac{d_c}{m})^2 + (\frac{d_s}{S})^2}. \end{aligned} \tag{1}$$

where m is a constant pre-determined to evaluate the importance between grayscale differences and spatial differences. This controls the compactness of output supervoxels. When m is large, the distance measure is mainly determined by spatial differences resulting supervoxels having similar and regular shapes and boundaries. Otherwise, the generated supervoxels fit better to the ground-truth boundaries where grayscale differences indicate.

B. Superpixels Extracted Via Energy-Driven Sampling (SEEDS)

SEEDS algorithm develops superpixels by exchanging the pixels on existing boundaries between superpixels, instead of developing regions from centers like SLIC. SEEDS superpixel generation is regarded as an energy maximization problem where color distribution and boundary shape of superpixels contribute to the energy term.

By default, SEEDS iterations begin with sampling the original 3D grayscale input image as grids. The grid size of clusters is determined by $\sqrt[3]{N/K}$ where N denotes the voxel volume of the whole input, which is the same as the SLIC algorithm. The cluster A_k is described as $[no, num, h_s, g_s]$ where no is the segment label for the cluster starting from 1, num is the voxel volume within the cluster, and h_s, g_s is the color and boundary histogram distributions of all voxels, respectively. Each voxel only has one unique cluster label. A segment map with the same size as the input image is filled with cluster labels to efficiently assign and modify segment labels for voxels, obtaining easy correspondence between the voxels index and their cluster label. Then, the segment map is used to locate the boundary voxels. For each iteration, multiple new partitioning near the boundary voxels are investigated. If the energy term of the new partition of A_k, A_m is larger than the term of the current partition, the boundary block or voxel will be moved from A_k to A_m. In this case, the segment map and $[no, num, h_s, g_s]$ of A_k, A_m will all be updated in a cost-effective way. The way to find new partitioning is to randomly choose a fixed ratio of all boundary voxels for each iteration in our experiment. The ratio is set as 0.01 because a large ratio makes the algorithm hard to converge to the global maximum while too small a ratio makes the algorithm need more iterations.

Algorithm 1 SLIC supervoxel generation for 3D grayscale images

```
Initialization of centers C_k ← [g_k, x_k, y_k, z_k]^T
Sample voxels to regular grids with size S ← ∛(N/k)
Label l(i) ← −1   for each voxel i
Distance d(i) ← ∞   for each voxel i
residual error E ← ∞
while E > threshold do
  for each cluster center C_k do
    for each voxel i within grid region around C_k  do
      D ← distance between C_k and i
      if D<d(i) then
        d(i) ← D
        l(i) ← k
      end if
    end for
  end for
  Compute new centers
  Compute residual error E
end while
```

Energy term contains two parts by default.

$$E(s) = H(s) + \gamma G(s). \tag{2}$$

where γ control the weights of the two terms.

The term $H(s)$ evaluates the color or grayscale distributions. A supervoxel should be as homogeneous as possible in grayscale. For supervoxel A_k, we have grayscale distribution C_{A_k} computed from binned voxels. C_{A_k} is calculated by discretizing the images into color bins H_j where j denotes the value of binned scale. $\delta()$ helps to calculate the histogram bar for each H_j. The normalization factor for C_{A_k} is given by Z. $H(s)$ as a global grayscale energy is calculated as follows:

$$H(s) = \sum_k \sum_{H_j} \frac{1}{Z} \sum_{i \in A_k} \delta(I(i) \in H_j) \tag{3}$$

In the equation, the normalization factor Z is somehow confusing whether to use the number of voxels within each supervoxel or the voxel volume of the whole input image. In

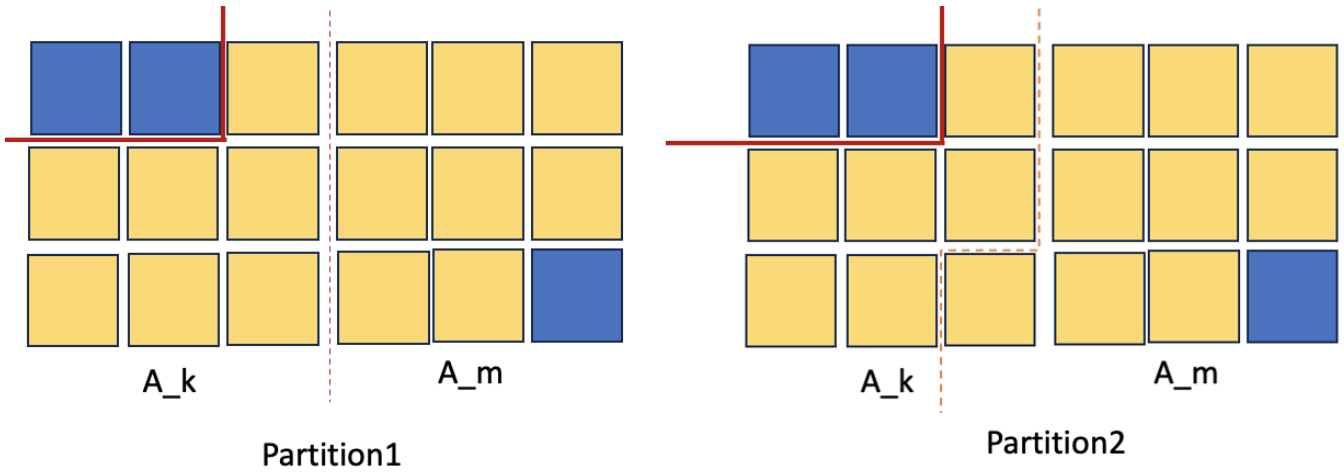


Fig. 1: One simple 2D example to illustrate the normalization factor in SEEDS algorithm. The example explains how we decide if partition 1 or partition 2 is better according to the color term. Blue and yellow show the different color distributions of each pixel. The red solid line indicates where the ground truth boundary is. The red dotted line indicates the boundaries between superpixel segmentation of A_k, A_m.

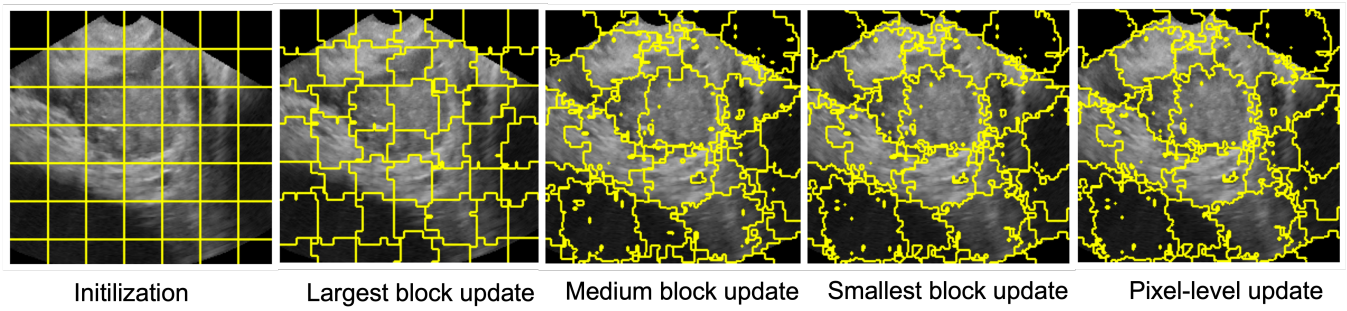


Fig. 2: Block and pixel movements. This shows the different levels of iterations of the SEEDS algorithm. The first image is the initialization as grids, and the following figures show block update from large size to small size to pixel-level upgrade.

Fig. 1 example, if the normalization factor is the voxel volume of each supervoxel.

$$H(s)_{p1} = ((\frac{2}{9})^2 + (\frac{7}{9})^2) + ((\frac{1}{9})^2 + (\frac{8}{9})^2).$$
$$H(s)_{p2} = ((\frac{2}{8})^2 + (\frac{6}{8})^2) + ((\frac{1}{10})^2 + (\frac{9}{10})^2).$$

In order to approach the ground truth boundary, $H(s)_{p2}$ should be larger than $H(s)_{p1}$, however, here in this example $H(s)_{p1} = 1.4567 > H(s)_{p2} = 1.445$. If we consider the voxel volume of the whole input:

$$H(s) = ((\frac{2}{18})^2 + (\frac{7}{18})^2) + ((\frac{1}{18})^2 + (\frac{8}{18})^2).$$
$$H(s) = ((\frac{2}{18})^2 + (\frac{6}{18})^2) + ((\frac{1}{18})^2 + (\frac{9}{18})^2).$$

$H(s)_{p2} = 0.3765 > H(s)_{p1} = 0.3641$. In most partitions comparison, normalizing by voxel volume of each suerpervoxel correctly decides whether to update or not. However, output accuracy will be improved by using the voxel volume of considered two supervoxels as normalizing factor.

The term $G(s)$ evaluates the shape of supervoxel, it penalizes the irregular boundaries. Same with $H(s)$, $G(s)$ should also be evaluated as a global term. Let N_i be the patch around voxel i, when calculating the segment label histogram within a cube region of size $N \times N \times N$.

$$G(s) = \sum_k \sum_k \frac{1}{Z} \sum_{j \in N_i} \delta(j \in A_k) \tag{4}$$

In order to obtain the global boundary term for each update, it is necessary to renew $G(s)$ on every voxel within considered supervoxels. This is time-consuming and sometimes impractical. So, we only apply to compute the boundary term in the pixel-level update. Let $b_{N_i}(k)$ be the histogram bars of supervoxel labels within the patch near voxel i. The boundary term criteria is simplified as:

$$b_{N_i}(n) + 1 \geq b_{N_i}(k) \tag{5}$$

This introduces a weak boundary penalty with least calculation cost, further improvement could be made if an experiment desires more control over the shape of the supervoxels.

The hill-climbing optimization problem is realized by updating the partitions in the direction of the larger energy term. SEEDS introduces unique movements from block level to pixel level. Block-level updates are used for increasing efficiency, as they allow for faster convergence, and help to avoid local

maxima. Fig. 2 shows the SEEDS iteration stages of a 2D slice from 3D data. In the meantime, there is a trick to use

Algorithm 2 SEEDS supervoxel generation for 3D grayscale images

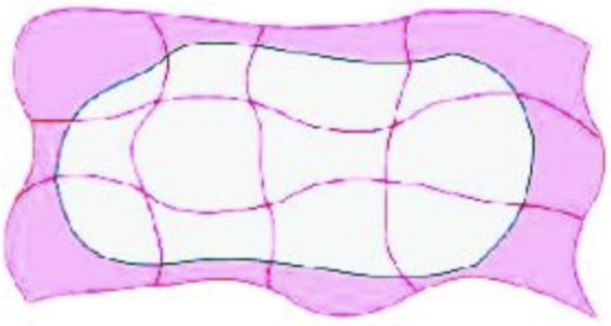
Initialization of clusters as grids with size $S \leftarrow \sqrt[3]{N/k}$
Binned image $h(i) \leftarrow$ binned grayscale for each voxel i
SuperVoxel $A_k \leftarrow [no, num]$
Label $l(i) \leftarrow$ Cluster Number for each voxel i
Color term $A_k[H] \leftarrow$ Histogram of $h(i)$ for k_{th} supervoxel
residual error $Err \leftarrow \infty$
while E > threshold **do**
  for each boundary block $b(i)$ between A_k, A_m **do**
    $E \leftarrow E(A_k[H]) + E(A_m[H]) + \gamma * b_{N_i}(k)$
    $E^{new} \leftarrow E(A_k[H^{new}]) + E(A_m[H^{new}]) + \gamma * b_{N_i}(m) + 1$
    if $E^{new} > E$ **then**
      Update H_k, H_m
      Update A_k, A_m
    end if
  end for
  Compute residual error Err
end while

C. Evaluation Methods

There multiple evaluation methods such as boundary recall (BR), undersegmentation error (UE), achievable segmentation accuracy (ASA) and segmentation speed, are introduced in [7] [8]. However, not all of them can be applied to our TRUS segmentation task. For example, the ground truth of the transrectal b-mode scan is the annotation of malignant and benign tissues, there are no clear and smooth boundaries to be regarded as reference to calculate boundary recall. So we only consider undersegmentation error and compactness.

1) Undersegmentation Error (UE)

UE evaluates the ratio of the number of voxels outside the intersection of the supervoxel A_k and the Ground truth G_i. The less UE is, the better segment outputs fit the boundary and ground truth segmentation. This parameter is not limited to [0,1].



$$UE_G(A) = \frac{\sum_i \sum_{k:A_k \cap G_i \neq \varnothing} |A_k - G_i|}{\sum_i |G_i|} \tag{6}$$

2) Compactness (CO)

In [9], it uses an isoperimetric quotient that relates the area of a given shape to the area of a circle that has the same perimeter to evaluate the compactness of 2D shapes. In our 3D case, we simplify this evaluation as:

$$CO = \frac{\text{length of output boundaries}}{\text{length of initial grids boundaries}} \tag{7}$$

IV. Results

This experiment runs on a single 3D b-mode scan in 3 with spatial size [284, 226, 234] which has a clear prostate boundary for evaluating the supervoxel segmentation output.

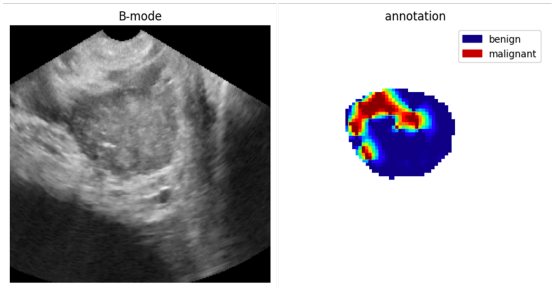


Fig. 3: 2D slice of 3D B-mode scan and its annotation.

A. SLIC

There is a well-established SLIC supervoxel segmentation function embedded in *skimage* library. Compared to by-hand implementation, the function from *skimage* generates faster segmentation and offer more flexible adjustment of hyperparameters. There are three hyperparameters to be decided, k the number of supervoxels, m which controls the compactness of output, and σ which controls the width of the Gaussian smoothing kernel as a pre-processing step. In Fig. 4, when compactness is big, the marked boundaries of clusters fail to follow the structures of the prostate scan. On the contrary, when compactness is too small, the boundaries between clusters are too sensitive to the grayscale changes in the image. There is a trade-off between the high boundary recall and regular shape and the similar size of different supervoxels. The larger the sigma is, the boundaries are smoother. There are speckle disturbances in the ultrasound B-mode image, suitable pre-processing kernel helps to generate more practical shapes of supervoxels.

B. SEEDS

There is no open-sourced implementation of SEEDS in generating supervoxels in 3D grayscale inputs, implementation in Python is added in the appendix section. In Fig. 2, the expanding boundaries vary a lot in different phases. In our experiment, block size [11,5,3] is applied in block-level updates. Between the smallest block update and pixel level update, there are already no distinct boundary changes. In Fig. 5, γ weight of boundary term and N considered patch size make a contribution to the penalty of generated boundaries. When γ and N are larger, the boundaries are smoother. Since

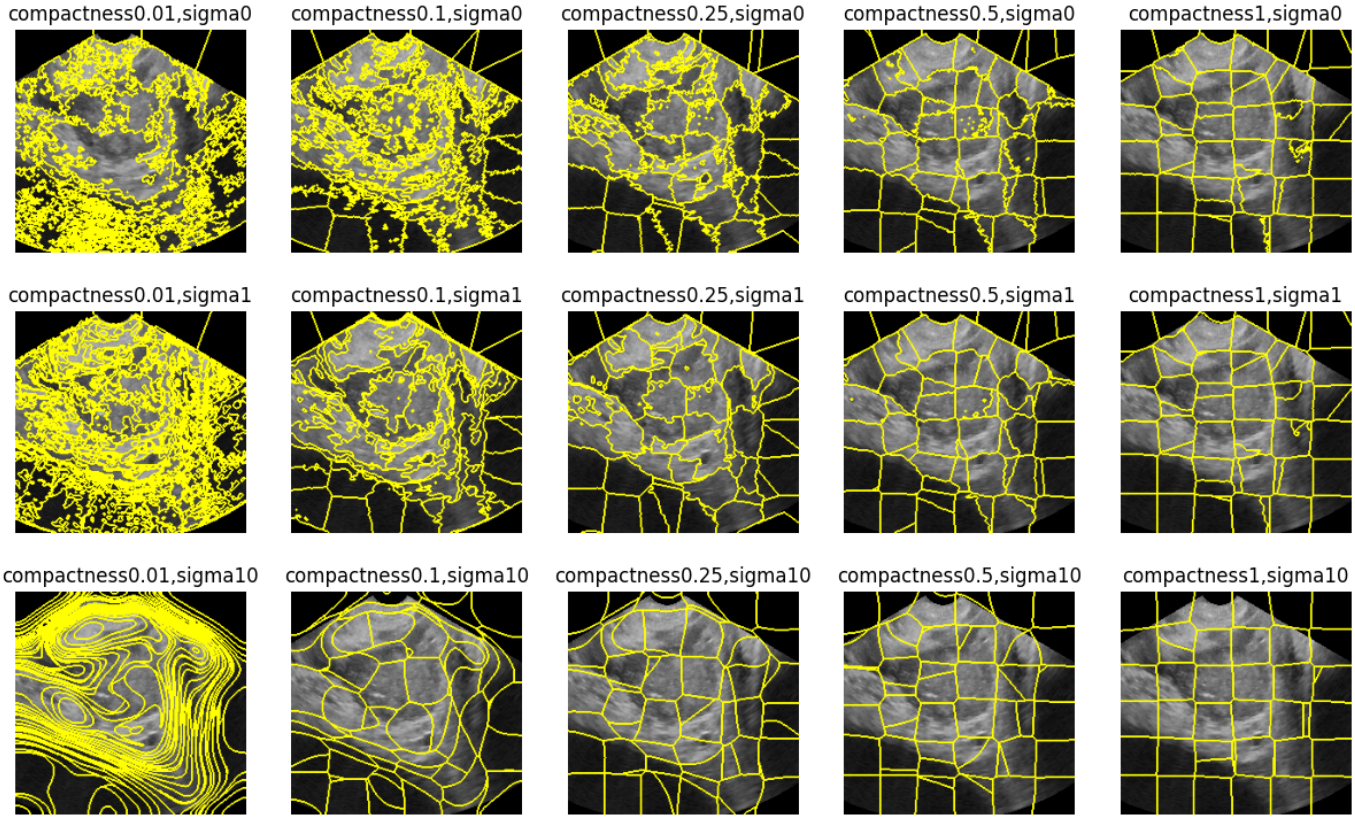


Fig. 4: 2D slice outputs of SLIC supervoxel generation on 3D ultrasound scan using different compactness and sigma.

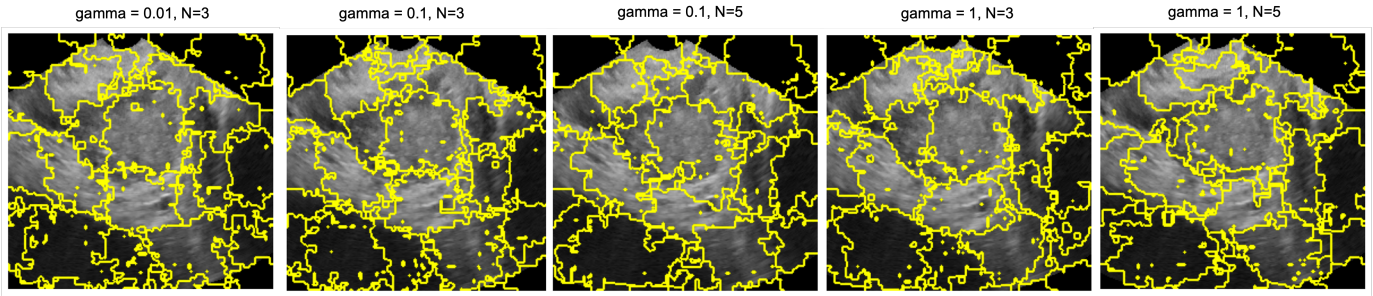


Fig. 5: 2D slice outputs of SEEDS supervoxel generation on 3D ultrasound scan using different gamma and N for boundary penalty.

	SLIC (sigma=1)					SEEDS				
Evaluation	compactness=0.01	0.1	0.25	0.5	1	$\gamma = 0.01, N = 3$	(0.1,3)	(0.1,5)	(1,3)	(1,5)
UE	1.5206	0.4036	0.8924	0.7837	0.9689	0.3985	0.3528	0.4939	0.3589	0.4880
CO	4.2317	1.9958	1.1554	0.9598	0.9549	2.0534	2.1012	1.9064	2.0703	1.9577
Num	197	274	286	288	288	252	252	252	252	252

TABLE I: Evaluation matrix: Undersegmentation Error (UE), Compactness (CO), Supervoxel Number (Num) of SLIC and SEEDS segmentation using different parameters.

the boundary term is a weak control, the outputs of SEEDS do not show great variance among different parameters as SLIC outputs. In general, our by-hand implementation obtains acceptable performance of SEEDS segmentations. However, the boundaries are too sensitive to B-mode speckle changes, some pre-processing and more strict boundary term criteria should be applied for applications in further medical image processing.

C. Summary

In table I, several evaluations are given out. Ground truth segmentation map is obtained from the annotation map shown in Fig. 3. We only consider the prostate area as the whole ground truth not considering separating the malignant and benign tissues. There are too many intersections between two classes of tissues making it impossible to gain a reliable UE.

The table clearly shows that there is a trade-off between the accuracy of fitting ground truth boundaries and structures and regular boundaries. In SLIC $compactness = 0.1, 0.25$, UE decreases with CO increases. In general, our SEEDS segments perform better than SLIC in fitting the prostate area contour while having much more irregular and curry boundaries. It is noticeable that the number of supervoxel does not change with parameters in SEEDS compare to SLIC. Both algorithms adopt the same segment label number 300 set for K. However, in our SEEDS implementation, we use stable grid initialization and there is no discarding and consolidation of supervoxel clusters. The calculation speed is not compared in the table because the two implementations are in different schemes, and it is meaningless to compare computing time. Both two meet requirements of real-time applications, and the whole algorithm ends within minutes.

V. Conclusion

In this study, we conducted a thorough investigation and implementation of two supervoxel generation algorithms, namely SLIC, and SEEDS for Transrectal Ultrasound (TRUS) images. The objective was to compare their performance and assess their suitability for supervoxel-based applications in TRUS image analysis. SLIC offers promising results while SEEDS implementation has a shortcoming of uneven borders. This study still inspires especially beginners to integrate supervoxel-based analyses into their TRUS image processing pipelines.

References

[1] B. Fulkerson, A. Vedaldi, and S. Soatto, "Class Segmentation and Object Localization with Superpixel Neighborhoods," *Proceedings of the IEEE International Conference on Computer Vision*, pp. 670–677, 2009.

[2] S. Wang, H. Lu, F. Yang, and M. H. Yang, "Superpixel tracking," *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1323–1330, 2011.

[3] B. Alexe, T. Deselaers, and V. Ferrari, "Measuring the objectness of image windows," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 11, pp. 2189–2202, 2012.

[4] M. Van den Bergh, X. Boix, G. Roig, and L. Van Gool, "SEEDS: Superpixels Extracted via Energy-Driven Sampling," *International Journal of Computer Vision*, vol. 111, no. 3, pp. 298–314, 9 2013. [Online]. Available: https://arxiv.org/abs/1309.3848v1

[5] X. Ren and J. Malik, "Learning a classification model for segmentation," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 1, pp. 10–17, 2003.

[6] Z. Chen, B. Guo, C. Lib, and H. Liu, "Review on Superpixel Generation Algorithms Based on Clustering," *Proceedings of 2020 IEEE 3rd International Conference on Information Systems and Computer Aided Education, ICISCAE 2020*, pp. 532–537, 9 2020.

[7] D. Stutz, A. Hermans, and B. Leibe, "Superpixels: An Evaluation of the State-of-the-Art," *Computer Vision and Image Understanding*, vol. 166, pp. 1–27, 12 2016. [Online]. Available: http://arxiv.org/abs/1612.01601 http://dx.doi.org/10.1016/j.cviu.2017.03.007

[8] M. Van den Bergh, X. Boix, G. Roig, and L. Van Gool, "SEEDS: Superpixels Extracted via Energy-Driven Sampling," *International Journal of Computer Vision*, vol. 111, no. 3, pp. 298–314, 9 2013. [Online]. Available: https://arxiv.org/abs/1309.3848v1

[9] "Measuring and evaluating the compactness of superpixels | IEEE Conference Publication | IEEE Xplore." [Online]. Available: https://ieeexplore.ieee.org/document/6460287

Appendix

A. SLIC implementation

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

from skimage.data import astronaut
from skimage.color import rgb2gray, gray2rgb
from skimage.filters import sobel
from skimage.segmentation import  slic, watershed
from skimage.segmentation import mark_boundaries, find_boundaries
from skimage.util import img_as_float

import cv2
import numpy as np
from cv2 import ximgproc
from matplotlib import pyplot as plt

img = img_as_float(bmode['data4d'][:,:,:].squeeze())
### SLIC
fig, ax = plt.subplots(3, 5, figsize=(15, 9), sharex=True, sharey=True)
coms = [0.01,0.1,0.25,0.5,1]
sigmas = [0,1,10]
for i,sigma in enumerate(sigmas):
    for j,com in enumerate(coms):
       segments_slic_3d = slic(img, n_segments=60, compactness=com, max_num_iter=10,sigma=sigma,
       ↪  start_label=1,channel_axis=None)
       print(com, sigma,len(np.unique(segments_slic_3d)))
       ax[i,j].imshow(mark_boundaries(img[25,:,:], segments_slic_3d[25,:,:]))
       ax[i,j].set_title('compactness{},sigma{}'.format(com, sigma))
       ax[i,j].axis('off')

### WaterShed
gradient = sobel(img)
segments_watershed_3d = watershed(gradient, markers=250, compactness=0.001)
fig, ax = plt.subplots(1, 5, figsize=(15, 9), sharex=True, sharey=True)
coms = [0.00001,0.0001, 0.001,0.002,0.01]
sigmas = [0,1,10]
i=0
for j,com in enumerate(coms):
    segments_watershed_3d = watershed(gradient, markers=60, compactness=com)
    print(com,len(np.unique(segments_watershed_3d)))
    ax[j].imshow(mark_boundaries(img[25,:,:], segments_watershed_3d[25,:,:]))
    ax[j].set_title('compactness{}'.format(com))
    ax[j].axis('off')
```

B. SEEDS implementation

```
import math
from skimage import io, color
import numpy as np
import matplotlib.pyplot as plt
from tqdm import trange
import os
from skimage.segmentation import mark_boundaries, find_boundaries
import cv2

class Cluster(object):
    cluster_index = 1

    def __init__(self, num_pixels=0, pixels=[]):
        '''
        if num_pixels=0, pixels should be []
        '''
        self.num_pixels = num_pixels

        self.pixels = pixels
        self.color_hist = []
        self.boundary_hist = []
        #self.boundarypixels = []
        self.no = self.cluster_index
        Cluster.cluster_index += 1


class SEEDSProcessor(object):
```

```
def __init__(self, image, K, N, N_L, bin,alpha, gama):
    self.K = K # total number of superpixels
    self.N = N
    self.N_step = int(N/2)
    self.colorbin = bin
    self.alpha = alpha
    self.gama = gama
    self.num_levels = N_L
    self.iter_pixel = 0

    self.data = image
    self.image_height = self.data.shape[0]
    self.image_width = self.data.shape[1]
    self.image_depth = self.data.shape[2]
    self.Size = self.image_height * self.image_width * self.image_depth
    #self.S = int(math.sqrt(self.N / self.K))
    self.S = int(max(self.image_height,self.image_width,self.image_depth)/math.pow(self.K, 1/3)) ##
    ↪  grid size
    self.block_size = []
    a = self.S/3
    for i in range(self.num_levels):
        self.block_size.append(int(a))
        a = a/2

    self.block_size = np.array(self.block_size)
    print("grid size", self.S)
    print("block size", self.block_size)
    self.clusters = []
    self.label = {}
    self.hist = np.zeros((self.image_height, self.image_width, self.image_depth))
    self.Segment = np.zeros((self.image_height, self.image_width, self.image_depth))
    self.boundary = np.ones((self.image_height, self.image_width, self.image_depth))
    self.boundary_energy = np.zeros((self.image_height, self.image_width, self.image_depth))

def make_cluster(self, h,w,d):
    N_step = self.N_step
    img = np.ones((self.image_height,self.image_width,self.image_depth))
    ## init all clusters to create grids..
    pixels = []
    h_ = min(h+self.S,self.image_height)
    w_ = min(w+self.S,self.image_width)
    d_ = min(d+self.S,self.image_depth)

    #print(h,w,d, h_,w_,d_)
    idxs = np.argwhere(img[h:h_, w:w_, d:d_])
    num_pixels = idxs.shape[0]
    pixels = list(idxs+np.array([h,w,d]))
    return Cluster(num_pixels, pixels)

def init_clusters(self):
    ## initialize the grids in 3d
    #num_clusters = (len(a)-1)*(len(b)-1)*(len(c)-1)
    #h, w, d = [N_step, N_step, N_step]
    h, w, d = [0, 0, 0]
    flag= 0
    while h < self.image_height:
        while w < self.image_width:
            while d < self.image_depth:
                self.clusters.append(self.make_cluster(h=h,w=w,d=d))
                d += self.S
                flag +=1
            d = 0
            w += self.S
        w=0
        h += self.S

    self.hist= np.round_(self.colorbin/255 *self.data,0) + 1
    for no, cluster in enumerate(self.clusters):
        for p in cluster.pixels:

            self.Segment[tuple(p)] = no+1
        cluster.color_hist = np.histogram(self.hist[self.Segment==no+1].flatten(),bins=
        ↪  self.colorbin,range=[1,self.colorbin+1])[0]
    self.boundary = find_boundaries(self.Segment)
    #self.iter_pixel = int(np.count_nonzero(self.boundary)*self.alpha)
```

```
def boundary_h(self,h,w,d, N_step):
    #N_step = self.N_step
    block = self.Segment[h-N_step:h+N_step+1,w-N_step:w+N_step+1,d-N_step:d+N_step+1]
    boundary_hist =
    ↪  np.unique(self.Segment[h-N_step:h+N_step+1,w-N_step:w+N_step+1,d-N_step:d+N_step+1].flatten(),return_count
    num_s = np.count_nonzero(find_boundaries(block))

    return boundary_hist[0],boundary_hist[1] ,num_s

def neigh_hist(self, h,w,d, label, N_step):

    block = self.Segment[h-N_step:h+N_step+1,w-N_step:w+N_step+1,d-N_step:d+N_step+1]

    c_neigh = self.hist[h-N_step:h+N_step+1,w-N_step:w+N_step+1,d-N_step:d+N_step+1]
    c_neigh_label = c_neigh[np.where(block==label)]
    num_n = len(c_neigh_label)

    color_hist= np.histogram(c_neigh_label,bins=self.colorbin,range=[1,self.colorbin+1])[0]
    return block, color_hist,num_n

def energy(self, hist, NM):
    '''
    histogram, NM normalized weight
    '''
    hist = np.array(hist)/NM

    return np.sum(hist*hist)


def assignment(self, number_level):
    ## find random boundary pixels, update the superpixel label for them
    N = self.block_size[number_level]
    N_step = int(N/2)
    gama = self.gama
    if N > self.N:
        gama=0

    N_step_B = int(self.N/2)
    num_N = N * N * N

    self.boundary_pixels = np.argwhere(self.boundary)
    pixel_idxs =
    ↪  np.random.choice(range(self.boundary_pixels.shape[0]),int(len(self.boundary_pixels)*self.alpha))

    for h,w,d in self.boundary_pixels[pixel_idxs]:
    #for h,w,d in self.boundary_pixels:
        label = int(self.Segment[h,w,d])

        if label == 0 or h not in range(N_step,self.image_height-N_step) or w not in
        ↪  range(N_step,self.image_width-N_step) or d not in range(N_step,self.image_depth-N_step):
            continue
        #if label == 0 or h not in range(N,self.image_height-N) or w not in range(N,self.image_width-N)
        ↪  or d not in range(N,self.image_depth-N):
        #    continue

        cluster = self.clusters[label-1]
        num_p = cluster.num_pixels

        color_hist= cluster.color_hist
        expo_clusters, boundary_hist ,num_s= self.boundary_h(h,w,d,N_step_B)
        #boundary_hist = boundary_hist/num_b

        num_label = boundary_hist[np.where(expo_clusters==label)]
        block, colorH, num_n= self.neigh_hist(h,w,d,label,N_step)

        for i,expo_cluster in enumerate(expo_clusters):
            expo_cluster =  int(expo_cluster)

            if expo_cluster ==0 or expo_cluster == label:
                continue
            num_exo = boundary_hist[i]

            cluster_index = expo_cluster-1
            c_hist2 = self.clusters[cluster_index].color_hist
            num_p2 = self.clusters[cluster_index].num_pixels

            # calculate current energy value
```

```
                hs1 = self.energy(color_hist, num_p+num_p2)
                hs2 = self.energy(c_hist2,num_p2+num_p)
                Hs = hs1 + hs2
                Hs_ = self.energy(color_hist-colorH, num_p+num_p2) +self.energy(c_hist2 +colorH,
                ↪  num_p+num_p2)

                Es = Hs + num_label*gama
                Es_ = Hs_ + num_exo*gama
                #if Hs_ > Hs :#or Hs_ == Hs:
                if Es_ > Es:
                    #print("updated")
                    self.Segment[h-N_step:h+N_step+1,w-N_step:w+N_step+1,d-N_step:d+N_step+1][block==label]
                    ↪  = expo_cluster
                    #self.boundary_energy[h,w,d] = Gs_

                    cluster.num_pixels -= num_n
                    self.clusters[cluster_index].num_pixels += num_n

                    cluster.color_hist -= colorH
                    self.clusters[cluster_index].color_hist += colorH
                break

        self.boundary = find_boundaries(self.Segment)

    def save_lab_image(self, path, lab_arr,slice):
        """
        save the gray image
        :param path:
        :param lab_arr:
        :return:
        """
        abs_path = os.path.join("outputs_test_0719",path)
        gray_arr = lab_arr[slice,:,:]

        io.imsave(abs_path, gray_arr)


    def iterate_times(self, iter):
        slice = 25
        self.init_clusters()
        print("init cluster number",len(self.clusters))
        print("init seg max",np.max(self.Segment.flatten()))
        plt.figure()
        plt.imshow(mark_boundaries(self.data[slice,:,:],self.Segment[slice,:,:],mode='inner'))
        plt.axis('off')
        name = 'ceus3d_init.png'
        self.save_lab_image(name, self.Segment, slice)
        print("SEEDS initilization completed")
        N_L = self.num_levels
        iters = iter/N_L

        for i in trange(iter):
            print(int(i/iters))
            self.assignment(int(i/iters))

        name = 'a{alpha}L{level}K{k}N{n}B{b}G{g}.png'.format(alpha=self.alpha,level=self.num_levels,
        ↪  k=self.K,n=self.N,b=self.colorbin,g=self.gama)
        self.save_lab_image(name, self.Segment, slice)
        name =
        ↪  'a{alpha}L{level}K{k}N{n}B{b}G{g}_boundary.png'.format(alpha=self.alpha,level=self.num_levels,
        ↪  k=self.K,n=self.N,b=self.colorbin,g=self.gama)
        self.save_lab_image(name, self.boundary, slice)
        plt.figure()
        plt.imshow(mark_boundaries(self.data[slice,:,:],self.Segment[slice,:,:],mode='inner'))
        plt.axis('off')

### img is 3D grayscale input image
p = SEEDSProcessor(image=img,  K=300, N_L=4, N=3, bin=16, alpha=0.01,gama=1)
p.iterate_times(10)
```