Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

Магістерська робота

освітній ступінь – магістр

на тему: **«Поля Ґіббса та їх застосування в сегментації зображень»**

Виконала: студентка 2-го року навчання

освітньо-наукової програми «Системний аналіз»,

спеціальності 124 Системний аналіз

Левченко Іларія Сергіївна

Керівник: Чорней Р.К.,

кандидат фіз.-мат. наук, доцент

Рецензент___Тригуб Олександр_____

Кваліфікаційна робота захищена з оцінкою _____

Секретар ЕК _____

«____» _____ 20____ р.

Київ – 2022

Міністерство освіти і науки України
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав.кафедри математики,
проф., д.ф-м.н.
_____Б. В. Олійник
(підпис)
„____”_____2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на дипломну роботу

студенту 2-го курсу магістратури, факультету інформатики__

Левченко Іларії Сергіївні

Розробити модель сегментації зображень на основі Рандомних Полів Гібса

Вихідні дані:

- Image Analysis, Random Fields and Dynamic Monte Carlo Methods A Mathematical Introduction by Gerhard Winkler”

Зміст ТЧ до магістерської роботи:

Зміст
Анотація
Вступ
1 Gibbs random fields for image segmentation
2 Model description and implementation
3 Results and their analysis
Висновки
Список літератури
Додатки

Дата видачі „___” _____ 2021 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Тема: _____ Поля Ґіббса та їх застосування в сегментації зображень

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на дипломну роботу.	01.10.2021	
2.	Огляд технічної літератури за темою роботи.	01.01.2021	
3.	Розробка моделі сегментації зображень	31.01.2022	
3.	Програмування моделі	30.03.2022	
5.	Застосування розробленої програми до масиву зображень	30.04.2022	
7.	Написання пояснювальної роботи.	20.05.2022	
8.	Створення слайдів для доповіді та написання доповіді.	22.06.2022	
8.	Аналіз отриманих результатів з керівником, та попередній захист магістерської роботи.	27.06.2022	
10.	Корегування роботи за результатами попереднього захисту.	30.06.2022	
11.	Остаточне оформлення слайдів.	30.06.2022	
12.	Захист магістерської роботи (проекту)	08.07.2022	

Студент **_____Левченко І.С._____**

Керівник **_____Чорней Р.К._____**

“___01___”_жовтня 2021_____

Зміст

АННОТАЦІЯ

This paper addresses a Gibbs random field (GRF) model for image segmentation of variety of images without the usage of machine learning, training data or supervision. Images represented in four colourspaces were, including greyscale, RGB, L*a*b* and OHTA. An optimisation is achieved through the Simulated Annealing algorithm. Results of segmentation are analysed for limitations and usefulness for certain types of images.

Key words: Gibbs Random Fields, Segmentation, Simulated Annealing (SA)

У цій статті розглядається модель Рандомних Полів Гіббса (GRF) для сегментації різноманітних зображень без використання машинного навчання, навчальних даних або додаткового контролю. Зображення, представлені в чотирьох колірних просторах, у тому числі чорнобілому, RGB, L*a*b* і OHTA. Оптимізація досягається за допомогою алгоритму Simulated Annealing. Результати сегментації аналізуються на предмет обмежень і корисності для певних типів зображень.

Ключові слова: Рандомні поля Гіббса, сегментація, Simulated Annealing (SA)

INTRODUCTION

Image segmentation is important and relevant problem, which still has a lot to be researched. Understanding of segment may vary from source to source, so here segment is used to mean maximally homogenous regions, meaning that some property remains similar for the whole area. Such property for image may be colour, lightness, etc. It is relevant for vide variety of spheres, including image recognition software, which by itself has huge application, including smart vity solutions, medical diagnosis, and much more. The problem is addressed using so many algorithms and models, with machine learning methods becoming more popular; methods becoming more complex and more advanced. Gibbs random fields (GRF) is very useful model in representing images, and using global energy function for futher processing.

And while usage of GRFs for image segmentation is not new notion, most research focuses either on application for specific cases and images, for example, medical scans; demonstrates results on very simplistic images, or are accompanied by more advanced methods of machine learning, training data or supervision. The question arise, how effective GRF model is for wide variety of images, without more complex algorithm, or training.

Therefore, the aim of this paper was to develop and programme GRF model, develop programme for it; and main task is to then analyse results and limitations of it for image segmentation using developed model.

An object of study is Gibbs random fields model, and subject of the study is application of GRF-based model for image segmentation.

The structure of the paper includes three chapters.

First one, the GRFs are defined as well as their relation and usefulness for image segmentation based on the literature study.

Second chapter, defines model developed for image segmentation as well as its application through programme.

In third chapter, the results of model application (segmented images) are analysed and conclusions regarding usage and limitations of the model are drawn.

Scientific novelty of the paper includes, conclusions drawn on the usage of the GRF model for image segmentation, and when it may or may not be applied.

1. Gibbs random fields for image segmentation

On Gibbs random fields and Markov random Fields

At first, MRFs were used in statistical dynamics to model particles on two- or three-dimensional lattices. In 1984, Geman and Geman introduced the usage of MRFs in image segmentation and currently, they are often used in statistical image processing. For image processing using MRFs, pictures are presented as data organised in lattices where pixels play the part of particles. [1]

Other names used for MRFs are Markov networks or undirected graphical models.

The main concept of Markov Random Fields is that if we are in a location we are more likely to be influenced by what is nearby than information that is further. Therefore for MRF models, the concept of neighbourhood is defined, which encompasses points of data which impact and everything that is not included is negligible. In image processing, this idea is applicable as the regions of images often have similar properties, such as colour, brightness, etc. and are homogenous.

Now let us give the formal definition of the MRFs.

Markov random fields are defined on an undirected graph model that expresses conditional dependence between nodes. MRF can be defined through neighbourhood relationships. All nodes in S are related to each other through the neighbourhood system $\aleph = \{\aleph(\mathrm{i}), \mathrm{i} \in \mathrm{S}\}$, where $\aleph(\mathrm{i})$ is a subset of S, which consists of neighbouring to i, nodes. A node cannot be a neighbour to itself. For a finite set of nodes $S = \{1, \dots, N\}$, MRF is a family of random variables $X_i, i \in S$, which probability function, with relation to the neighbourhood system $\aleph$ satisfies the following conditions [1]:

1) $P[X = x] > 0$ – positivity property, which ensures that all possible values of X have a positive probability of occurring;
2) $P\left[X_i = x_i | X_j = x_i, j \neq i\right] = P[X_i = x_i | X_j = x_j, j \in \aleph(\mathrm{i})]$ – Markov property.

The drawback of representing MRFs through local conditional probabilities is that complete system representation is represented through joint probability, and there is no direct method to derive joint probability $P[X_1, \dots, X_N]$ from conditional $P[X_i | X_j, j \in \aleph(\mathrm{i})]$. However, it is combated due to the Hammersley-Clifford theorem, which was firstly presented in an unpublished paper in 1971 by Hammersley and Clifford [2].

Next, let us move to Gibbs random fields.

From [3] Abstract, Gibbs random fields were mostly used in statistical physics and quantum field theory and were later formalized by mathematics as they play important role in many applications of probability theory.

The beginning of the definition is the same as MRFs.

Gibbs random field a set of random variables $X = \{X_i, i \in S\}$ defined on finite set $= \{1, \dots, N\}$, with respect to neighbourhood $\aleph = \{\aleph(\mathrm{i}), \mathrm{i} \in \mathrm{S}\}$, which obeys Gibbs distribution (also term Gibbs measure is is used):

$$P(X = x) = \frac{1}{Z} * \exp\{-\frac{1}{T}E(x)\}$$

where Z is a constant used to normalise distribution: $Z = \sum_{x \in X} \exp\{-\frac{1}{T}E(x)\}$. T is temperature constant which is generally assumed to be equal to 1, and therefore in many articles and model descriptions element $\frac{1}{T}$ is not even displayed. And $E(x)$ is energy function which is equal to the sum of clique potentials.

$$E(x) = \sum_{C \in C} \phi_C(x_C)$$

Clique $C \subseteq S$ is a subset of graph, where each node in the subset is connected with all other nodes of the subset. In case of images, it means that all pairs of pixels in subset are neighbours. Clique which consists of n nodes is called clique of nth order C_n. The set of cliques of the model is union of all subsets of cliqueth of all possible orders $C = C_1 \cup \dots \cup C_n$.

Most MRF vision models are assumed to be homogeneous, for Gibbs random fields homogeneity is established if the energy function independent of the relative position of the cliques. [4]

An MRF is characterized by its Markov property which is a local property, while Gibbs random fields are characterized by the Gibbs distribution, which is a global property. The Hammersley-Clifford theorem allows us to equate these properties. Hammersley-Clifford theorem states the equivalence of Markov Random Field and Gibbs Random Field defined on the same graph, with the same set of random variables F set on the set S, with respect to the same neighbourhood system $\aleph$.[4]

Given the theorem, Gibbs fields can be redefined taking into account the properties of both Gibbs and Markov fields. Such definition, with variations, is most common in modern literature, especially focused on practical application and image processing.

Gibbs random fields are models for a set of random variables $X = (x_1, x_2, \dots, x_n)$ on an undirected graph, where these variables are organized in cliques X_c, where C

is a set of cliques; and for each clique the compatibility function $\Psi_C(X_C)$ is defined, such that the probability distribution p over random variables, has the following form:

$$P(X = x) = \frac{1}{Z}\prod_{c \in C} \Psi_c(X_c), \qquad (1)$$

The probability of x is taken with the respect to the joint distribution of the X_c [5]

A clique is a subset of nodes which all are connected with each node in a subset.

Z is a normalization function to ensure probability distribution, called partition function and can be expressed as follows:

$$Z = \sum_{x \in X} \prod_{c \in C} \Psi_c(X_c)$$

From this probability distribution is positive for all domains. In this case, it can be represented as a Gibbs measure.

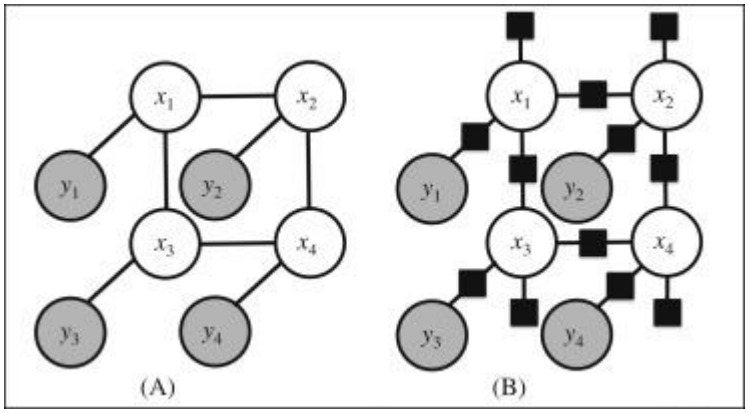
$$P(X = x) = \frac{1}{Z} * \exp\{-\frac{1}{T}E(x)\}$$

$$E(x) = \sum_{c \in C} \phi_C(x_C)$$

$$\phi_C(x_C) = -\log(\Psi_C(x_C)) - potential\ on\ clique\ C$$

On such graphs, the Markov property is ensured.

Figure 1. MRF model visualisation



Given the model commonly used for image processing (Figure 1) [6].

On figure 1 we can see GRF graph (A) and its factor graph (B), which allow to specify function used to create model.

The special case of Markov Random fields are Conditional random fields. CRFs are probabilistic undericted graphical model, in which nodes are separated into

exactly two separate subsets: observations Y and states X, where the model itself is conditional distribustion $P[X|Y]$. [7]

On image segmentation

Segmentation is an important process in digital image processing with extensive application in different areas, including processing medical images, content-based image retrieval, computer vision, etc. Segmentation usually has two main ststeps1) identifying a set of features which would allow identifying regions of the same content and would be different for areas of different content and 2) segmentation method itself which is used on the identified features to identify segments of the image [8]. Feature extraction is a complicated topic and highly depends on what types of images are being processed. Currently, machine learning is often used to identify and specify such features. The task itself is easier if image processing consists of what to be later identified, as a segment of the same nature, for example, faces, or medical images of the same type. Other types of image processing activities include location of element on the image, super resolution, identification, semantic labeling, depth estimation, etc. These activities may utilize segmentation but are not the same as segmentation. There is a huge variety of segmentation methods currently used, including the clustering method, edge-detection, etc.

One of the ways segmentation is approached is as a pixel labelling task.

For each pixel in the picture $s \in S$ the feature vector is defined. For the picture in general it can be represented as $f = \{\vec{f_s}: s \in S\}$. The features most commonly include different colour characteristics and brightness. The set of labels is defined, and for each pixel, a label is assigned $x = \{x_s: s \in S\}$, $x_s \in \Lambda$. Which results in a $|\Lambda|^{NM}$ variants of labelling, where N *M is the size of the image.

Gibbs and Markov random fields have been used to model images in various image processing applications.

By defining local properties, through transitivity, a global model can be constructed, however, it does not ensure the usefulness of such model.

GRF is convenient in modelling, as the energy function of random variables is enough to define the whole model. While the disadvantages include, not always good performance or speed.

In Gibbs random field model representing an image, nodes are usually pixels

In [8] with reference to [9] three advantages of using MRF-based approaches to segmentation are named:

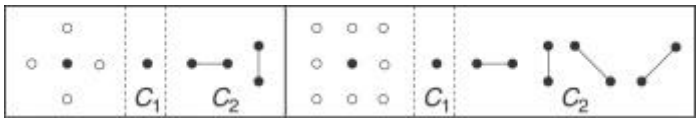
1) The spatial relationship introduced by MRF is easy to use in segmentation procedures;

2) As MRF are similar to Bayesian networks in how the dependencies are represented, MRD segmentation models can utilize the Bayesian framework, which allows utilizing various image features;
3) The label distribution can be obtained by maximizing the MRF model probability

A set of cliques can be also expressed as a set of edges of the graph.

Figure 2, presents two common neighbourhood systems and their cliques for an image. [10]

Figure 2. Neighborhood system



Using GRF for segmentation, means to find optimal labeling from the set of all possible labelings, by maximasing posteriory probability of getting label given feature.

$$x^{MAP} = \arg\max_{x \in \Lambda} P(x|f) = \arg\min_{x \in \Lambda} E(x)$$

2. Model description and implementation

For this thesis the goal was to develop image segmentation programme based on Gibbs Random Fields to be applied to variety images of not of any specific type, without application of machine learning and training of model. As in all studies literature MRF models were accompanied by training on some set of data, developed for very specific type of images, such as medical images of similar type, etc. or used for other purposes; demonstrated on only few images or simplistic ones used specifically for demonstration (few segments of different structure) it remained unclear of how effective MRF model would be by itself without application of other more complex tools. Therefore, in this work I applied a number of variations of MRF model to variety of images to analyse it effect and limitations.

Problem formulation

Let X be coloured image we observe and which we consider a representation of random field. The problem, is to find true labeling field for this image, where each of the labels correspond to one of the pixels.

Model

Two layer model was used: one layer of pixels, which represented by value using colourspace; and second is label layer.

Model parameters

Pixel labels are represented through Gaussian distribution, where mean and variance are estimated through empirical means.

$$P(f_s|x_s) = \frac{1}{\sqrt{2\pi}\sigma_{x_s}} \exp\left(-\frac{\left(f_s - \mu_{x_s}\right)^2}{2\sigma_{x_s}^2}\right)$$

$$\forall x \in \Lambda:$$

$$\mu_x = \frac{1}{|S_x|}\sum_{s\in S_x} f_s$$

$$\sigma_{x_s}^2 = \frac{1}{|S_x|}\sum_{s\in S_x} (f_s - \mu_x)^2$$

S_x is a set of pixels.

Energy function

As previously defined, ϕ_C denotes clique potential of clique C with labeling x_C. Here, as in most literature, we focus only on cliques of two sizes, assigning for all

other potential functions value zero, which is considered satisfactory to model spatial dependencies: singleton (c_0) – node itself, and doubleton $\{c_1, c_2, c_3, c_4\}$ – cliques of two nodes (node and it's neighbour).

$$E(x) = \sum_{c \in C} \phi_c(x) = \sum_{i \in C_1} \phi_{C_1}(x_i) + \sum_{(i,j) \in C_2} \phi_{C_2}(x_i, x_j) + \cdots$$

Singleton here is proportional to the probability of features given label: log(P(f | x)).

Doubleton prefers smoothness, meaning that neighbouring pixels belong to the same label.

$$\phi_{C_2}(x_i, x_j) = \beta\delta(x_i, x_j) = \begin{cases} -\beta & if \quad x_i = x_j \\ +\beta & if \quad x_i \neq x_j \end{cases}$$

We can now define

$$E(x) = \sum_{s} \left(\sqrt{2\pi}\sigma_{x_s} + \frac{(f_s - \mu_{x_s})^2}{2\sigma_{x_s}^2} \right) + \sum_{s,w} \beta\delta(x_s, x_w)$$

Beside this energy function, some variations were used following the same principles.

Colour model

Four colour models were studied.

RGB | All images are colour images and usually can be presented by RGB colour space, which is most commonly used to represent colours of images in computer graphics, and therefore any image is easy to present in such way without any transformation required. It is coordinate system represented by three primary colours (Red, Green, Blue) and each colour is represented by a vector stating intensity of each of the values (from 0 to 255). Given it was the one of the colour models used.

However, while RGB colour space is useful in displaying colour, is not perfect. Disadvantages of RGB colour space include:

- Differently displayed on different devices,
- not perceptually uniform,
- the difference between colours is not linear.

Which means that while human eye would perceive some colours similarly, the numerical representation in RGB can show significant difference between colours and vice verca.

CIELAB | Another colour space, created to combat the problem and commonly used with different applications including editing and colour analysis, is CIELAB color space, also referred to as L*a*b*. This colour space also present colours in vectors of size three, however coordinates have different meaning: L indicates lightness of the colour, while a and b representing two scales from red to green and from blue to yellow. While it is not perfect it was created to be while not trully perceptualy uniform and one of the most commonly used.

Ohta | And another colourspace used was Ohta's colour space also known as $I_1 I_2 I_3$, which is less known but was developed for segmentation and is easy to transform to from RGB system [11].

$$\begin{bmatrix} I_1 \\ I_2 \\ I_3 \end{bmatrix} = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} & \frac{1}{3} \\ \frac{1}{2} & 0 & -\frac{1}{2} \\ -\frac{1}{4} & \frac{1}{2} & -\frac{1}{4} \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Greyscale | And additionally Greyscale colourspace was used. It stores much less information (only intensity or lightness), however it also means it is much easier to process.

Algorithm

So the general algorithm of the developed programme is as follows

1) Initialise
2) Optimize
3) Reconstruct image (to show labeling) and save segmented image

1. ***Initialise***

So first part is to set all initial parameters, initialize image as a matrix and

Initial parameters

Beta is a parameter used within energy function, set to one. Temperature and coolrate are parameters used for optimization algorithm explained bellow.

BETA = 1

TEMPERATURE = 5.0

COOLRATE = 0.95

Neighbours are defining positions of the neighbours in the neighbourhood system.

NEIGHBORHOOD= [(-1,0) , (1,0) , (0,-1) , (0,1)]

Next are some coeficients used for modified energy functions.

beta_variances, beta_neighbours, coefcol, coefbrightnes, diffort=dt

And last, to run the algorithm, a number of segments and iterations have to be defined.

segmentss=[2,3,4,5,6,10]

iterationss=[100000,1000000,2000000,3000000,5000000]

Image: The image need to be read/converted in numerical format (matrix), corresponding to one of the colourspaces. For greyscale (black-white), the image is presented in a form of 2-dimensional matrix $N \times M$, where N and M are height and width of the image in pixels, with values from 0 to 225, representing lightness of the pixel. For three other, an image is a 3-dimensional matrix $N \times M \times 3$, with each pixel being represented by a vector of size 3.

```
def input_bw (imagepath):
    original = cv2.imread(imagepath)
    return cv2.cvtColor(original,cv2.COLOR_BGR2GRAY)

def input_RGB (imagepath):
    return cv2.imread(imagepath)

def input_LAB (imagepath):
    original = cv2.imread(imagepath)
    return cv2.cvtColor(original.astype(np.float32)/255, cv2.COLOR_RGB2Lab)

def input_OHTA (imagepath):
    original=cv2.imread(imagepath)
    for i in original:
        for j in i:
            R=j[0]
            G=j[1]
            B=j[2]
            i1=float(R*(1/3)+G*(1/3)+B*(1/3))
            i2=float(R*(1/2)+B*(-1/2))
            i3=float(R*(-1/4)+G*(1/2)+B*(-1/4))
            j=[i1,i2,i3]
    return original
```

For testing of the programme a different images were chosen, which vary in subject, colour scheme, composition, etc.

Initialize energy function

Energy function require us to know the variance and the means of the current configuration. Recalculating it for the whole image each time can be time consuming process. Therefore, on this stage we initialize labels matrix, and calculate sums of the parameters within each label, squares and number of the parameters within each. Which are then used to calculate variance. This way during each iteration we can easily recalculate numbers, sums, and squares by

simply adding and substracting one, value and value squared of the pixel, label for which was changed, from label it is in new configuration and label it was previously in. Having this updated information, it is easy to recalculate variance.

In case of coulors and vector for value of the pixel, three sums and squared sums are calculated for each of the values. For variance, their sum is used or sum by each of the dimensions.

```
def initialize_bw(img):
    labels = np.zeros(shape=img.shape ,dtype=np.uint8)
    nos = [0.0]*SEGMENTS
    sums = [0.0]*SEGMENTS
    squares = [0.0]*SEGMENTS
    for i in range(len(img)):
        for j in range(len(img[0])):
            #print(labels)
            l = randint(0,SEGMENTS-1)
            #print(img[i][j])
            sums[l] += img[i][j]
            squares[l] += img[i][j]**2
            nos[l] += 1.0
            labels[i][j] = l
    return (sums,squares,nos,labels)
```

```
def initialize_colour(img):
    labels = np.zeros(shape=(img.shape[0],img.shape[1]) ,dtype=np.uint8)
    nos = [0.0]*SEGMENTS
    sums0 = [0.0]*SEGMENTS
    squares0 = [0.0]*SEGMENTS
    sums1 = [0.0]*SEGMENTS
    squares1 = [0.0]*SEGMENTS
    sums2 = [0.0]*SEGMENTS
    squares2 = [0.0]*SEGMENTS
    for i in range(len(img)):
        for j in range(len(img[0])):
            #print(labels)
            l = randint(0,SEGMENTS-1)
            #print(img[i][j])
            sums0[l] += img[i][j][0]
            sums1[l] += img[i][j][1]
            sums2[l] += img[i][j][2]
            squares0[l] += img[i][j][0]**2
            squares1[l] += img[i][j][1]**2
            squares2[l] += img[i][j][2]**2
            nos[l] += 1.0
            labels[i][j] = l
    return (sums0,squares0,sums1,squares1,sums2,squares2,nos,labels)
```

```
def variance(sums1,squares1,nos1):
    t=(squares1-((sums1/nos1)**2))/nos1
    return t
```

Checks and other functions

Then there is function for doubleton, which returns beta if labels are the same and minus beta otherwise.

```
def delta(i,l):
    if(i==l):
        return -BETA
    return BETA
```

Is safe is function used to check that when we take neighbours we stay within the size of the image/matrix.

```
def isSafe(a,b,x,y):
    return x>=0 and x<a and y>=0 and y<b
```

2. ***Optimize***

For optimization of labeling random field, or minimization of energy function, simulated annealing algorithm was chosen, as it often used for image processing using MRFs and in theory it always work, meaning, minimization of the function occur and allow to receive MAP estimate.

The idea of simulated annealing is that system is "melting" – changing, while the temperature is high, which bit by bit getting lower. When the temperature lowers no changes can occur and we freeze the system [11].

The simulated algorithm steps are as follows [11]:

1. Initiate the Temperature *T*.
2. Compute the energy of the current state of the system.
3. Introduce slight changes to the current state of system.
4. Compute the new energy of the new state and compare it with energy of previous state.
5. If energy improved (is lower for new state), accept new state of the system. Else, accept new state of the system with some small probability

For current model, the following was used:
prob = np.exp((energy-newenergy)/temp)
if prob >= (randint(0,1000)+0.0)/1000:
change = True

6. Decrease the temperature.
7. Repeat from step 3, till temperature decreased sufficiently (or the set number of iterations are used).

Three variations of energy function are used:

Basic (bw and colour): is the standard energy function described above.

```
def calculateEnergy_bw(img,variances,labels):
    energy = 0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
            energy += math.log(math.sqrt(2*math.pi*variances[l]))+(((img[i][j]-(sums[l]/nos[l]))**2)/(2*variances[l]))
            for (p,q) in NEIGHBORS:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    energy += (delta(l,labels[i+p][j+q]))
    return energy
```

```
def calculateEnergy_colour(img,variances,labels):
    energy = 0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
            energy += math.log(math.sqrt(2*math.pi*variances[l]))
            energy += ((((img[i][j][0]-(sums0[l]/nos[l]))**2)+
                       ((img[i][j][1]-(sums1[l]/nos[l]))**2)+((img[i][j][2]-(sums2[l]/nos[l]))**2))/(2*variances[l]))
            for (p,q) in NEIGHBORS:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    energy += (delta(l,labels[i+p][j+q])/2.0)
    return energy
```

Clr: weighted function, where sum of the singletons and doubletons is weighted function.

```
def calculateEnergy_clr(img,variances,labels):
    energy = 0.0
    energy2=0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
            energy += math.log(math.sqrt(2*math.pi*variances[l]))
            energy += ((((img[i][j][0]-(sums0[l]/nos[l]))**2)+
                       ((img[i][j][1]-(sums1[l]/nos[l]))**2)+((img[i][j][2]-(sums2[l]/nos[l]))**2))/(2*variances[l]))
            for (p,q) in NEIGHBORS:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    energy2 += (delta(l,labels[i+p][j+q])/2.0)
    return beta_variances*energy+beta_neighbours*energy2
```

Clr2:is modification of clr, where we change doubleton part of the energy, to compare not neighbouring labels but values of the pixels within one label.

```
def calculateEnergy_clr2(img,variances,labels):
    energy = 0.0
    energy2=0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
            energy += math.log(math.sqrt(2*math.pi*variances[l]))
            energy += ((((img[i][j][0]-(sums0[l]/nos[l]))**2)+
                       ((img[i][j][1]-(sums1[l]/nos[l]))**2)+((img[i][j][2]-(sums2[l]/nos[l]))**2))/(2*variances[l]))
            for (p,q) in NEIGHBORS:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    t=0
                    if img[i][j][0]-img[i+p][j+q][0]<coef and img[i][j][0]-img[i+p][j+q][0]>coef:
                        t=t+1
                    if img[i][j][1]-img[i+p][j+q][1]<coef and img[i][j][1]-img[i+p][j+q][1]>coef:
                        t=t+1
                    if img[i][j][2]-img[i+p][j+q][2]<coefbrightnes and img[i][j][2]-img[i+p][j+q][2]>coefbrightnes:
                        t=t+1
                    delt=delta(l,labels[i+p][j+q])
                    if t>=diffort and l==labels[i+p][j+q]:
                        energy2 += ((-BETA))
                    else:
                        energy2 += ((BETA))


    return beta_variances*energy+beta_neighbours*energy2
```

Clr22:is modification of clr2, where we improve energy not only by having similar pixels nearby within one label, but also significant difference in colour if the labeling is different.

```
def calculateEnergy_clr22(img,variances,labels):
    energy = 0.0
    energy2=0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
            energy += math.log(math.sqrt(2*math.pi*variances[l]))
            energy += ((((img[i][j][0]-(sums0[l]/number[l]))**2)+
                       ((img[i][j][1]-(sums1[l]/number[l]))**2)+((img[i][j][2]-(sums2[l]/number[l]))**2))/(2*variances[l]))
            for (p,q) in NEIGHBORHOOD:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    t=0
                    tb=0
                    temp=float(img[i][j][0])-float(img[i+p][j+q][0])
                    if (temp<coefcol0 and temp>-coefcol0) :
                        t=t+1
                    if (temp>difbord0 and temp<-difbord0) :
                        tb=tb+1
                    temp=float(img[i][j][1])-float(img[i+p][j+q][1])
                    if (temp<coefcol1 and temp>-coefcol1) :
                        t=t+1
                    if (temp>difbord1 and temp<-difbord1) :
                        tb=tb+1
                    temp=float(img[i][j][2])-float(img[i+p][j+q][2])
                    if (temp<coefcol2 and temp>-coefcol2) :
                        t=t+1
                    if (temp>difbord2 and temp<-difbord2) :
                        tb=tb+1

                    if (t>=diffort and l==labels[i+p][j+q]) or (tb>=diffort and l!=labels[i+p][j+q]):
                        energy2 += ((-BETA))
                    else:
                        energy2 += ((BETA))


    return beta_variances*energy+beta_neighbours*energy2
```

3. ***Reconstruct and save***

Labels are converted in greyscale, so the difference in colour between segments would be maximal.

```
def reconstruct(labs):
    labels = labs
    for i in range(len(labels)):
        for j in range(len(labels[0])):
            labels[i][j] = (labels[i][j]*255)/(SEGMENTS-1)
    return labels
```

Reconstructed segmented image then saved with a name indicating all indicators used for its construction.

```
def save (colour, energy_function, neighbours):
    string=""
    #create string for the name with information on the parameters used#
    plt.imshow(reconstruct(labels) , interpolation='nearest',cmap='gray')
    cv2.imwrite(string+'.jpg',labels)
    print(string)
```

3. Results and their analysis

Results of model and its variations application vary, from rather good segmentation to poor noise. The results will be compared for main factors that varied in model, such as colourspace and energy function, based on example of few images, as to display and analyse all possible combinations would take too much time. The whole set of generated segmented images as well as originals can be found in attached annex in form of zip file.

Colourspace

As become obvious from received results and example shown bellow, coulourspace do not play significant role for model. While initial assumption was that coloured pictures would hold more information and therefore would allow better results, it was not confired. An attampts to develop more complicated model which would take into account nuinces of clours was unsuccessful and produced just noise.

	Original

	Colourspace - greyscale Segments – 2 Energy function – basic (calculateEnergy_bw) Iterations 1000000 T-4.0
	Colourspace – RGB Segments – 2 Energy function – calculateEnergy_colour Iterations - 1000000 T-4.0

	Colourspace – L*a*b* Segments – 2 Energy function – calculateEnergy_colour Iterations - 1000000 T-4.0
	Colourspace – L*a*b* Segments – 2 Energy function – calculateEnergy_colour without utilizing lightness parameter Iterations - 1000000 T-4.0

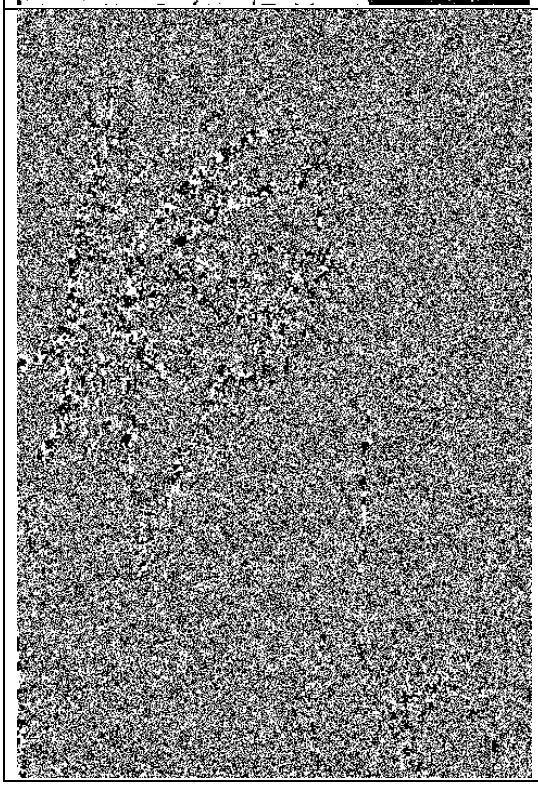
	Colourspace – Ohta Segments – 2 Energy function – calculateEnergy_clr (with coefficients bv-0.25_bn2-0.75) Iterations - 1000000

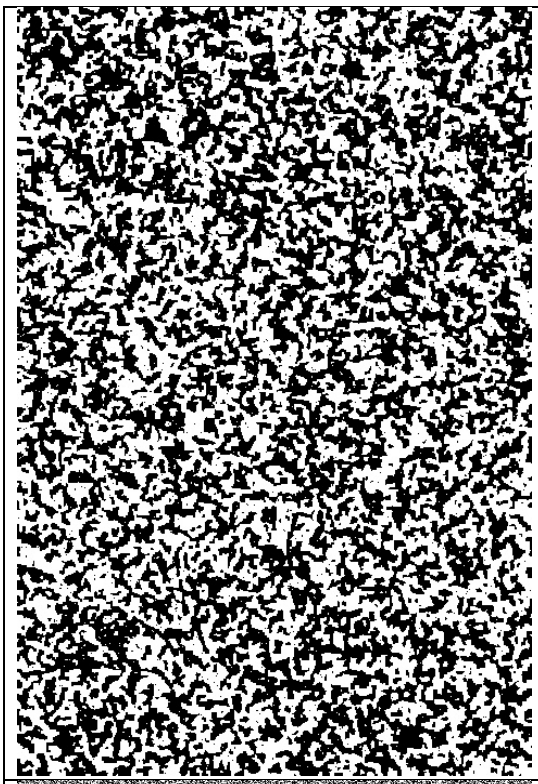
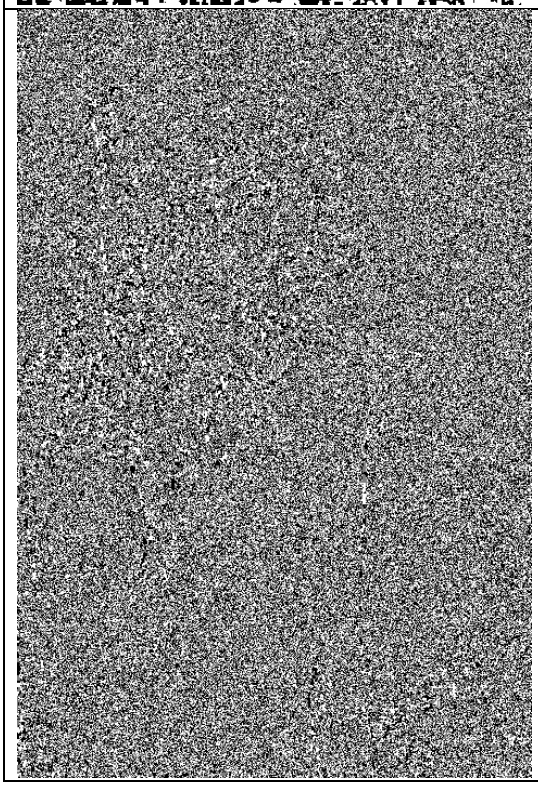
Energy function

Basic function demonstrated better results (both in grey scale and colour) as well as weighted energy function with non-zero coefficients demonstrated better results then other proposed variations of the energy functions variations.

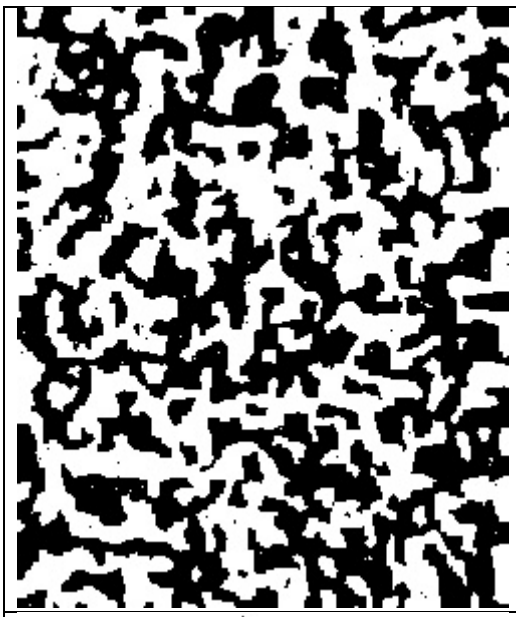
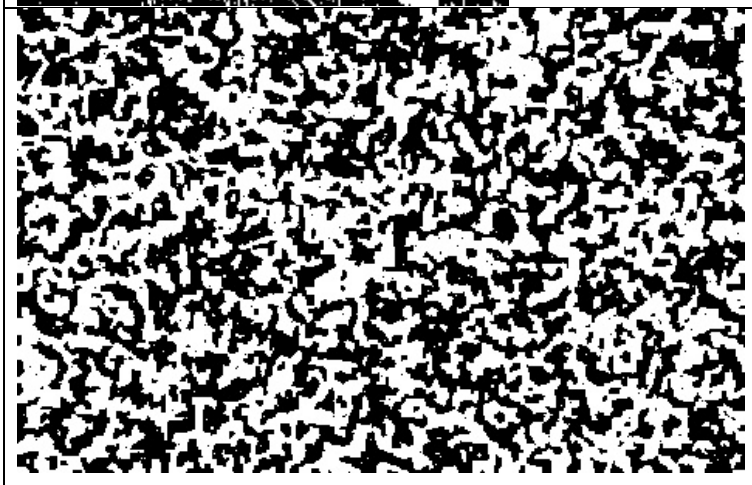
	Original

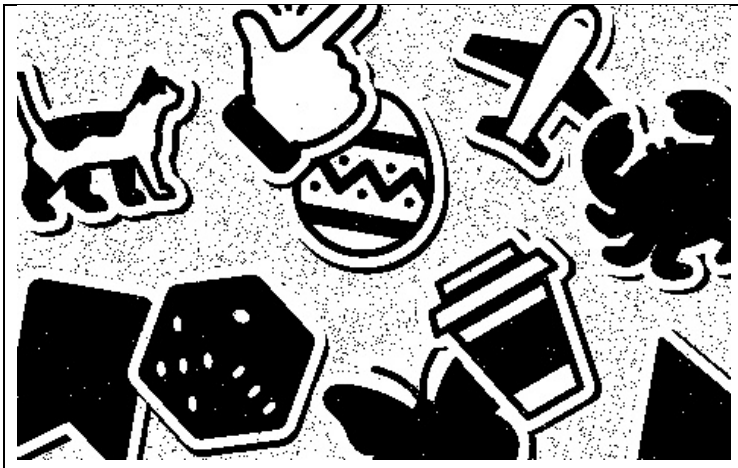
	Colourspace - greyscale Segments – 2 Energy function – basic (calculateEnergy_bw) Iterations - 2000000 T-10.0
	Colourspace – L*a*b* Segments – 2 Energy function – calculateEnergy_clr (with coefficients bv-0.25_bn2-0.75) Iterations - 2000000 T - 5

	Colourspace – OHTA Segments – 2 Energy function – calculateEnergy_clr (with coefficients bv-0.25_bn2-0.75) Iterations - 2000000 T - 5
	Colourspace – OHTA Segments – 2 Energy function – calculateEnergy_clr2 (with coefficients coefcol-0.1_coefb-0.1_dif-2_bv-0.25_bn2-0.75) Iterations - 2000000 T - 5

	Colourspace – L*a*b* Segments – 3 Energy function – calculateEnergy_clr22 (with coefficients coefcol-0.01_dif1-1_difborders-0.1-_bv-0.5_bn2-0.5) Iterations - 2000000 T - 5
	Colourspace – L*a*b* Segments – 3 Energy function – calculateEnergy_clr22 (with coefficients coefcol-0.01dif1-1_difborders-0.1_bv-0.5_bn2-0.5) Iterations - 2000000 T - 5

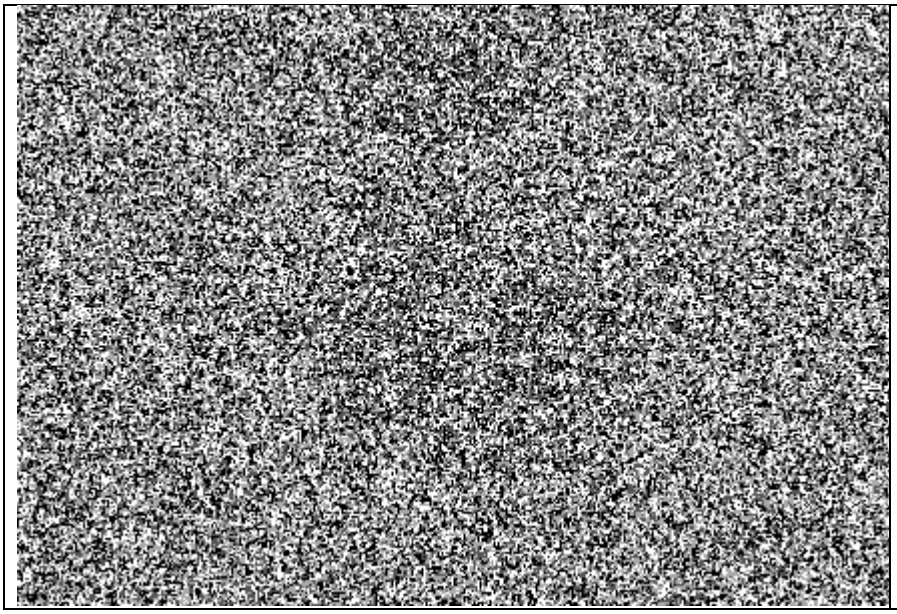
Considering weighted energy function, in function, relying just on doubleton results are insufficient, as such segmetntation produces just noise. While relying just on singletone, produced great results, and able to more accurately detect smaller details.

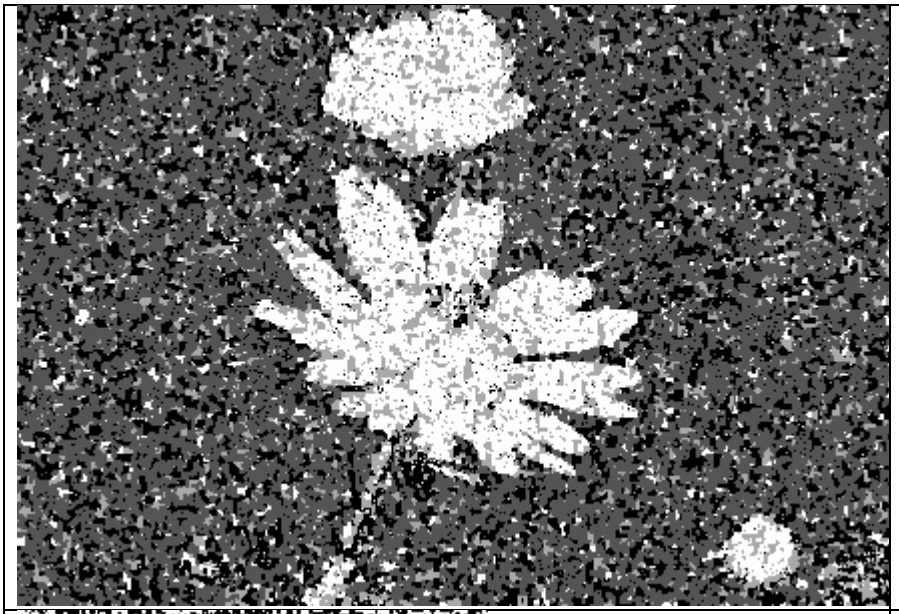
	Colourspace – L*a*b* Segments – 3 Energy function – calculateEnergy_clr22 (with coefficients coefcol-10_coefb-20_dif-2) Coefficient for first sum - bv-0 Coeficient for second sum in energy function - bn2-1 Iterations - 1000000 T - 4
	Colourspace – L*a*b* Segments – 3 Energy function – calculateEnergy_clr22 (with coefficients coefcol-10_coefb-20_dif-2) Coefficient for first sum - bv-1 Coeficient for second sum in energy function - bn2-0 Iterations - 1000000 T - 4
	Colourspace – OHTA Segments – 2 Energy function – calculateEnergy_clr (with coefficients coefcol-10_coefb-20_dif-2_bv-0_bn2-1) Coefficient for first sum - bv-0 Coeficient for second sum in energy function - bn2-1 Iterations - 2000000 T - 4

	Colourspace – OHTA Segments – 2 Energy function – calculateEnergy_clr (with coefficients coefcol-10_coefb-20_dif-2_bv-1_bn2-0) Coefficient for first sum - bv-1 Coeficient for second sum in energy function - bn2-0 Iterations - 2000000 T - 4

Number of iterations

There has to be sufficient enough number of iterations to get rid of all noise. If generally the segmentation was achieved, but some “dots” remained, by increasing number of iterations, the segmentation will improve. For most images, for which segmentation by used model was successful, number iteration in lower hundred thoughthands (100 000, 200 000, 300 000) for two segments, was enough to achive sufficient result. Increasing number of segments in model require also increase in iterations. For images for which some form of segmentation was achieved, meaning, that segmented image is not just noise, no conclusion on a sufficient number of iterations was achieved.

	Colourspace – RGB Segments – 4 Energy function – calculateEnergy_colour Iterations - 100000 T – 4

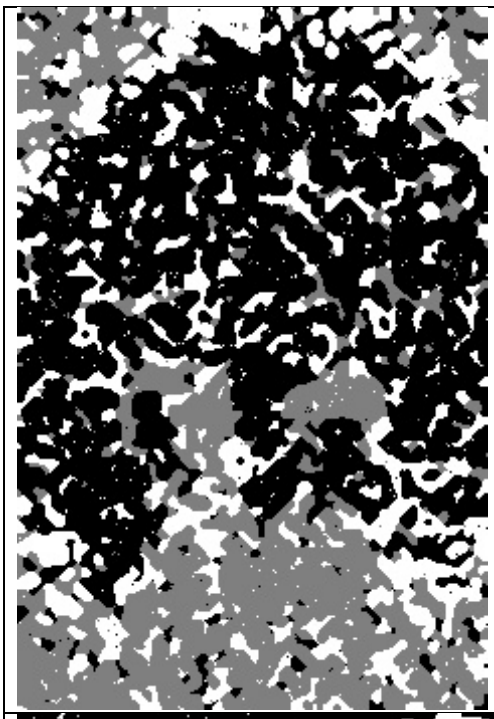
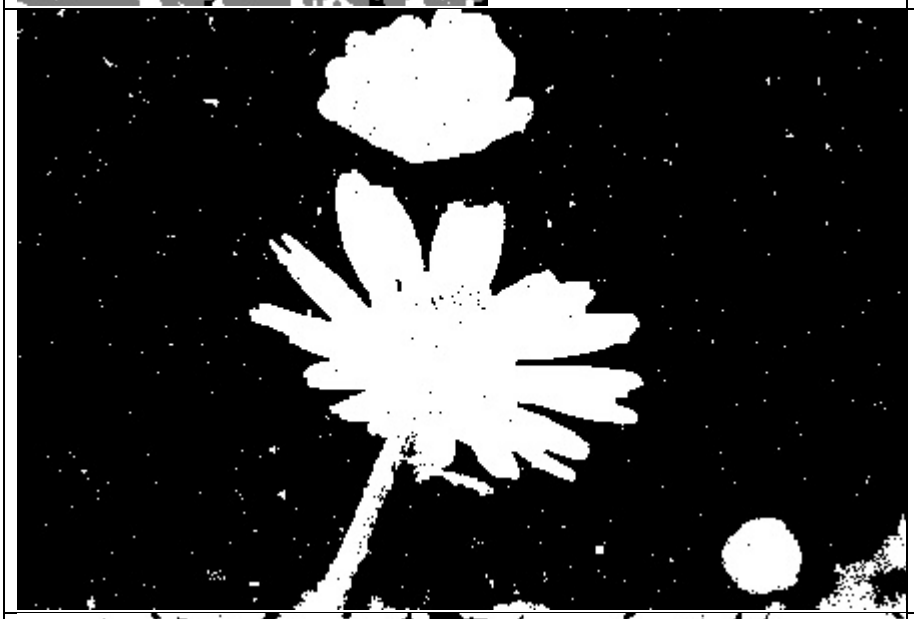
	Colourspace – RGB Segments – 4 Energy function – calculateEnergy_colour Iterations - 1000000 T – 4
	Colourspace – L*a*b* Segments – 2 Energy function – calculateEnergy_colour Iterations - 100000 T - 4

	Colourspace – L*a*b* Segments – 2 Energy function – calculateEnergy_colour Iterations - 1000000 T - 4

Neighborhood

Extending neighborhood, to include also diagonal elements, did not improve model.

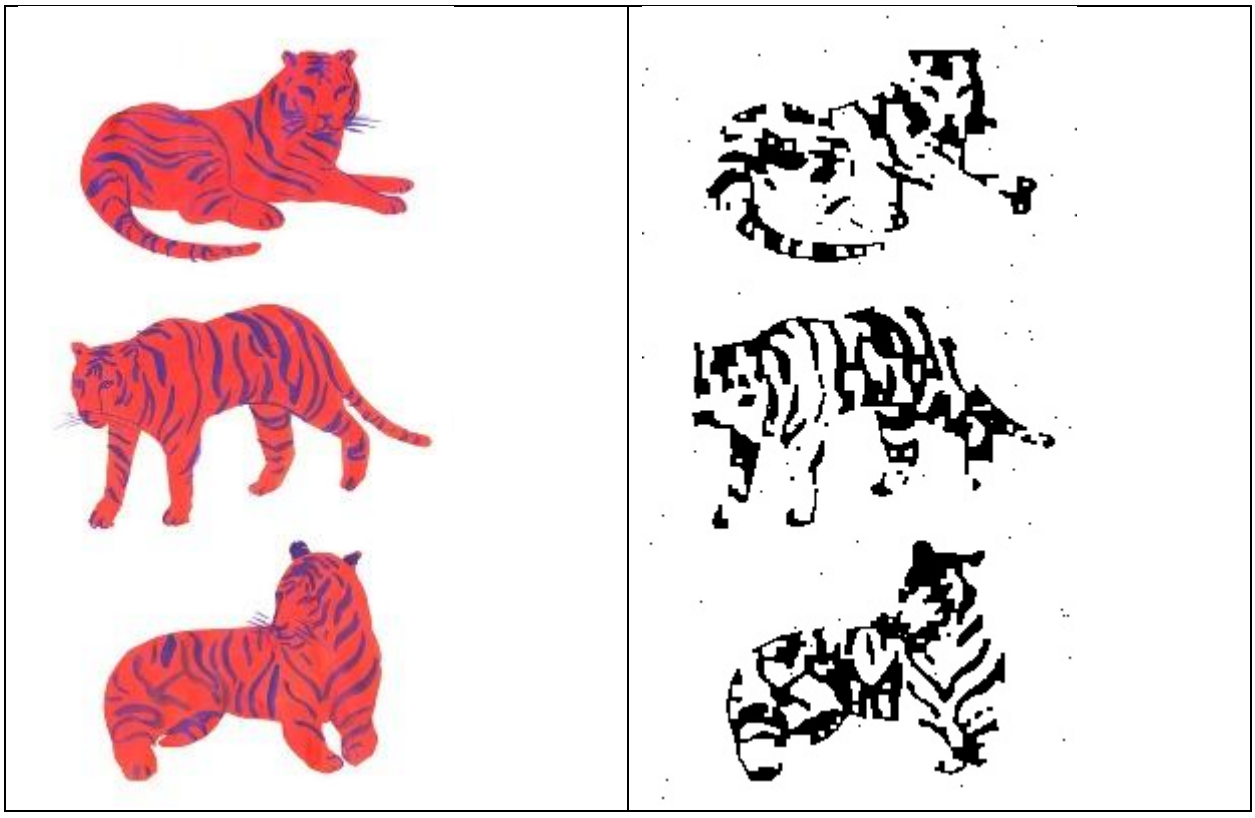
	Colourspace - greyscale Segments – 3 Energy function – basic (calculateEnergy_bw) Iterations - 1000000 T-4.0 Neighborhood includes four pixels

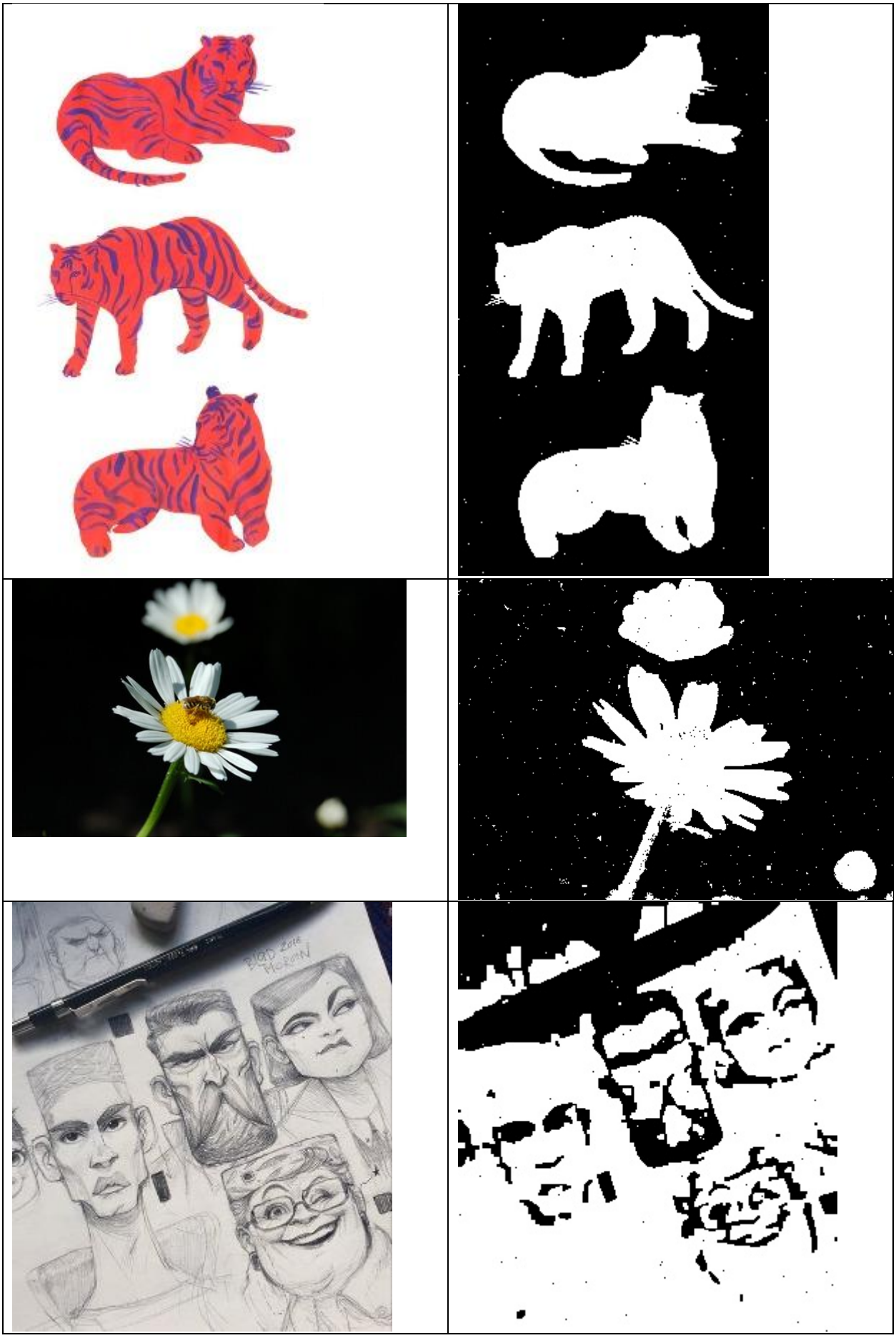
	Colourspace - greyscale Segments – 3 Energy function – basic (calculateEnergy_bw) Iterations - 1000000 T-4.0 Neighborhood includes eight pixels
	Colourspace - greyscale Segments – 2 Energy function – basic (calculateEnergy_bw) Iterations - 1000000 T-4.0 Neighborhood includes four pixels
	Colourspace - greyscale Segments – 2 Energy function – basic (calculateEnergy_bw) Iterations - 1000000 T-4.0 Neighborhood includes eight pixels

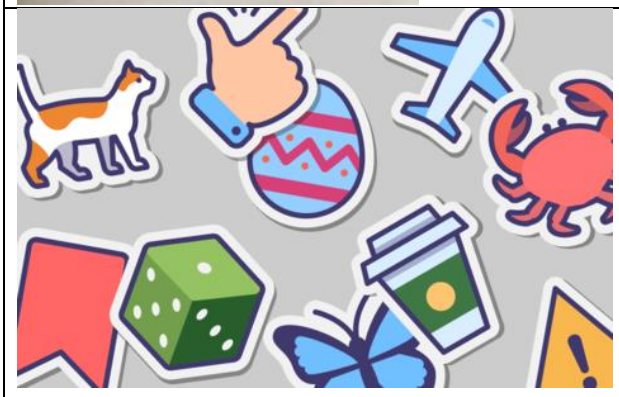
General conclusions

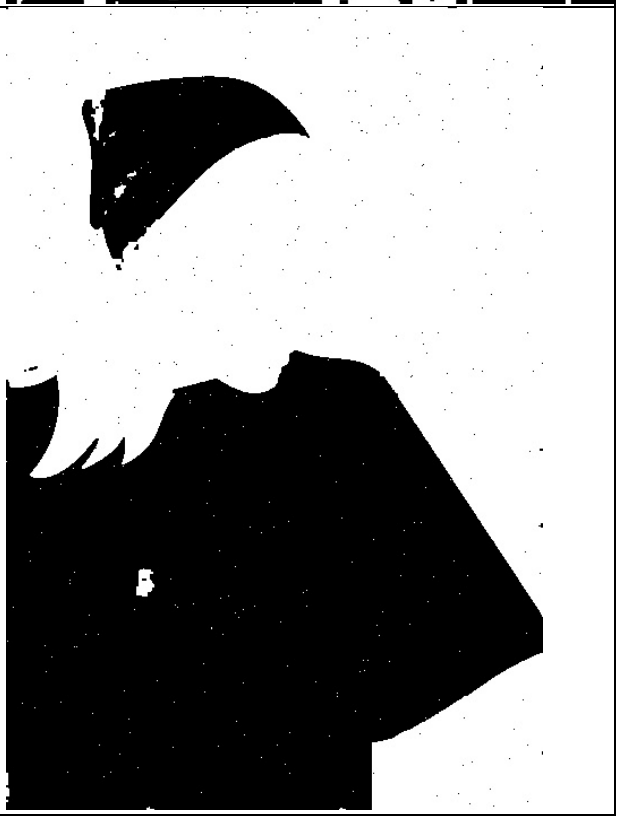
By itself, not attuned to specific type of pictures, MRFs have limited use.

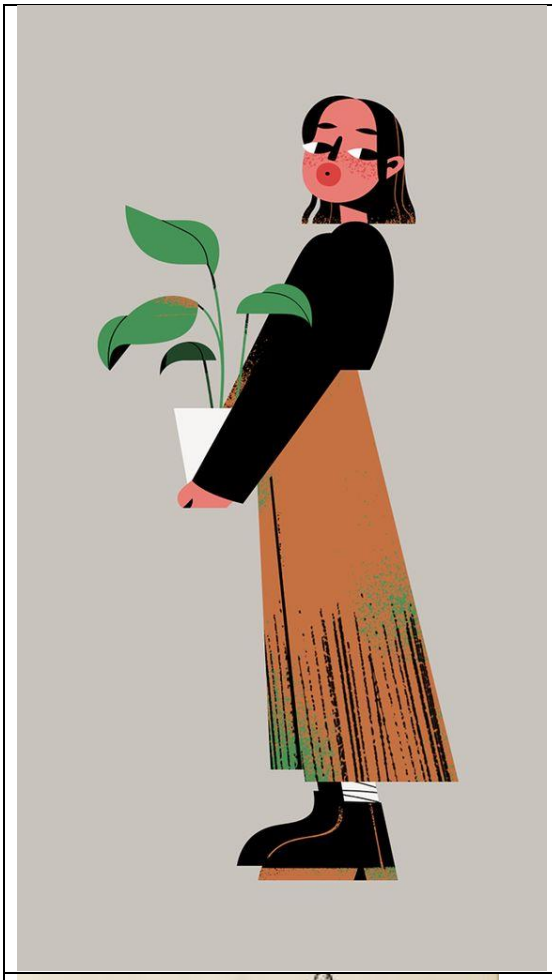
The best results were achieved on flat images, without gradient or significant shadows or much details. Or where segments have significant differences in colour (such as “statue”) Greyscale is enough for such images and demonstrates great results.

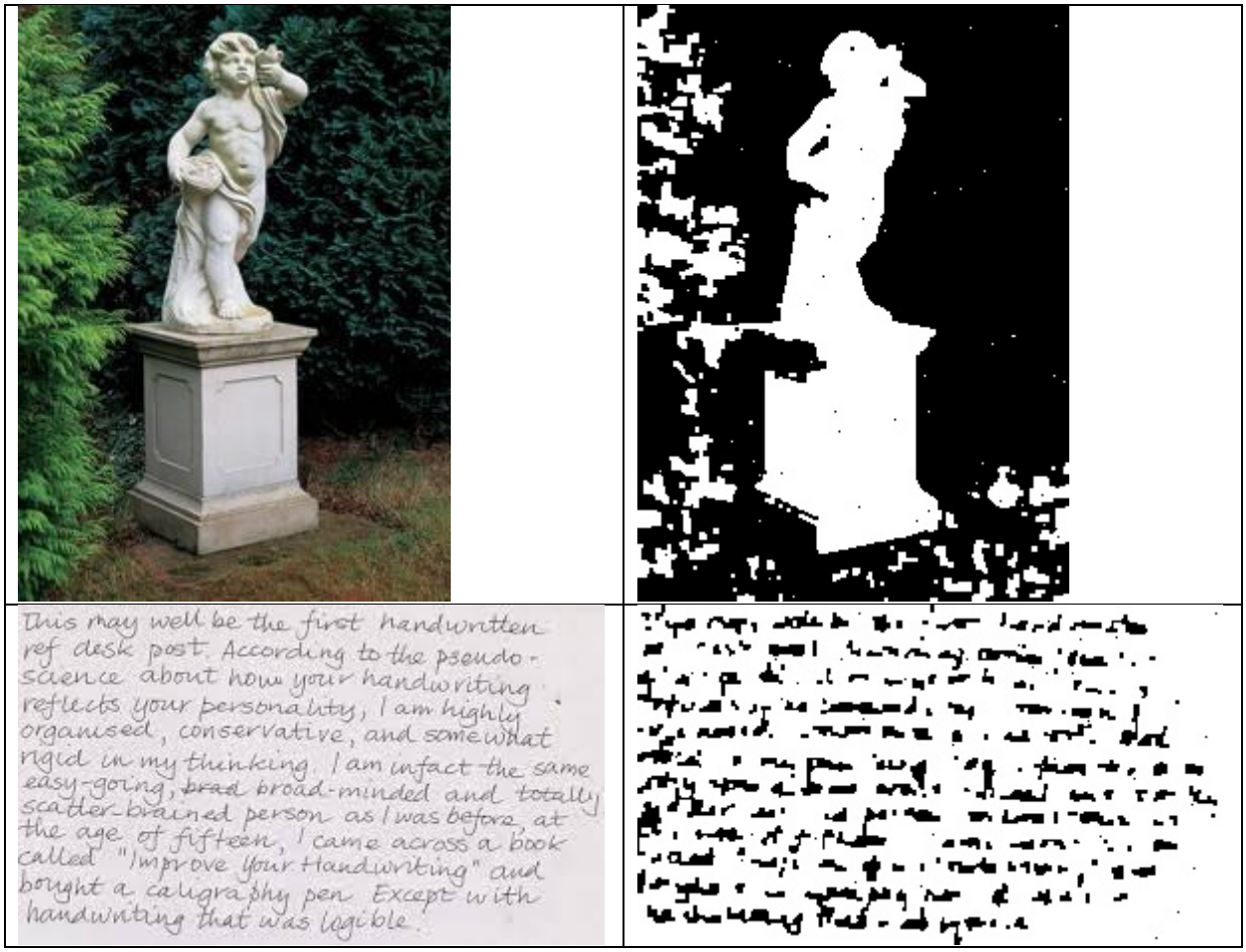




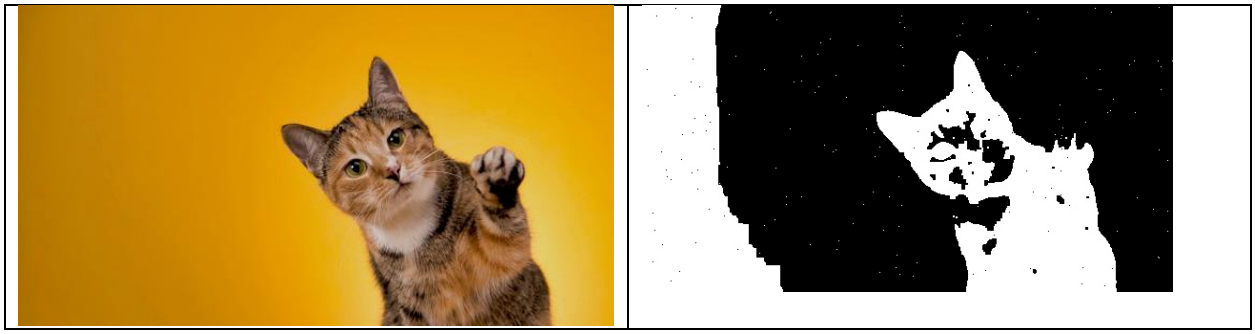


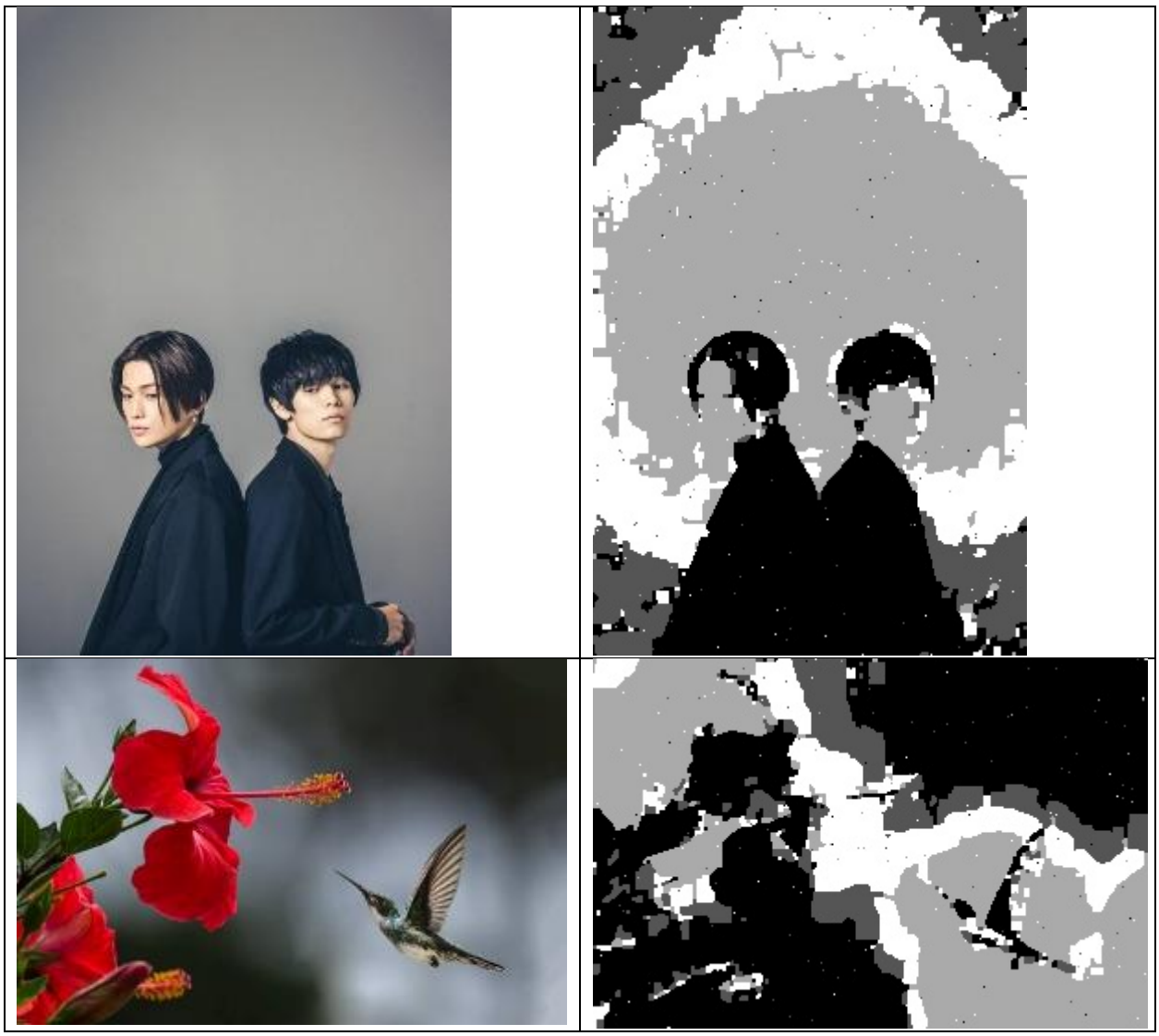






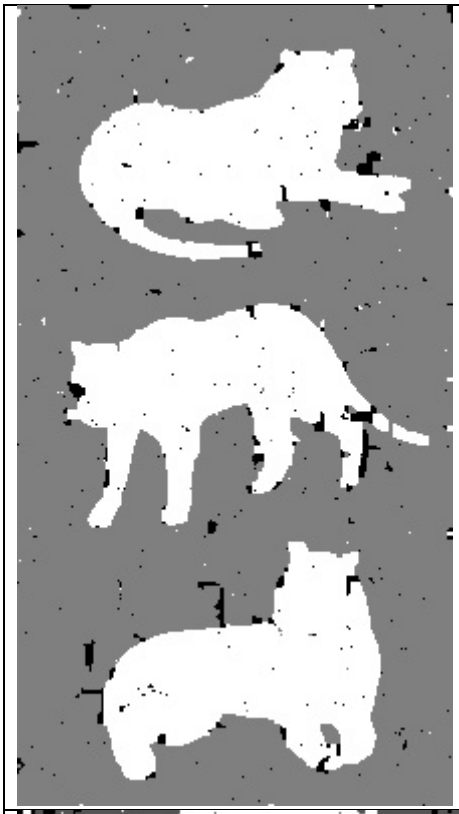
Simple gradient in pictures, as with “cat” or “two people portrait” or even “calibri” is not recognized as one segment, as variance of segment is greater. Without using variance or reducing it significance for energy function resulted in images that look as noise.

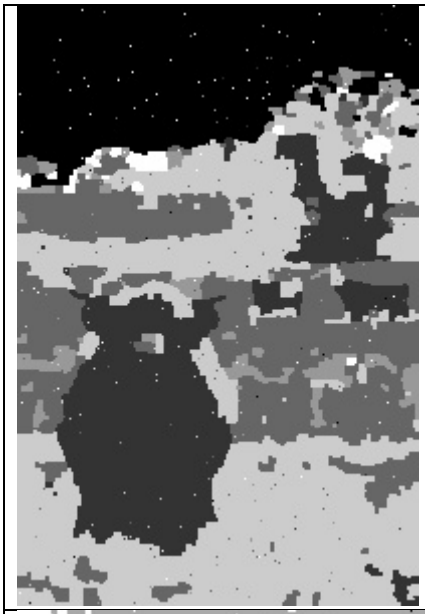




One of the greatest problems of the MRF model for image segmentation is significant shadows, as on “teapot” or “tiger”, as while people recognize the colour in shadow and in light as one colour, none of the colourspaces reflect that. For example white in shadow would give purplish colour, and in processing part in shadow and in light are recognized as different colours. None of the energy functions or colourspaces used were efficient in combating the problem.

Another problem, is inability to effectively control number of labels, or more precisely, segmentation of the image on the grater amount of labels. Which means that segmentation of image in three segments might look as segmentation in two segments with added noise. As with “tigers”. Or will added segments will divide semantical segment further.

	Colourspace - greyscale Segments – 3 Energy function – basic (calculateEnergy_bw) Iterations - 2000000 T-10.0
	Colourspace - greyscale Segments – 4 Energy function – basic (calculateEnergy_bw) Iterations - 4000000 T-10.0

	Colourspace – L*a*b* Segments – 6 Energy function – calculateEnergy_colour Iterations - 100000000 T-4.0
	Colourspace - greyscale Segments – 4 Energy function – basic (calculateEnergy_bw) Iterations - 3000000 T-10.0
	Colourspace - greyscale Segments – 4 Energy function – basic (calculateEnergy_bw) Iterations - 100000000 T-4

The last identified problem of universal usage of MRFs for image segmentation, is how image is being segmented itself. As there are no element of control or learning, and energy function are dependent only on colour feature (even if in different formats), segments identified are homogemeus but segmentation is not

semantic at all, which can easily be seen with “icons” or “desk”. Partially the issue can be conducted by additionally separating parts of one segment which are not connected, but the number of segments wouldn’t be controllable and any remenant noise would remain an issue.

However the issue of what to consider proper segments is not unique for this model and is universal problem of image segmentation.

CONCLUSIONS

Within this paper, in first chapter GRF was defined, and it’s relation to image processing and segmentation is stated. Second chapter, outlined GRF model and associated progrmme used for image segmentation. In third chapter, results of application of different variations of the model were analyzed, and conclusions on general application were drawn.

While GRFs introduce benefits in a way of presenting images and have definitely use in image processing and segmentation, they are in no way universal tool and cannot be applied to any image without preprocessing, some form of supervision or learning.

Such model is much more useful for simple images with clear foreground and background, and only two main segments.

Main problems of the studied model included inability to recognize gradient as one object and inability to exclude shadows, as well as control over number of segments.

Further study of the model in terms of extraction and application of useful additional information from coloured images, compared to greyscale ones need to be done. Important problem the control over segments, which remains to be seen, can it be achieved for big variety of images without any training.

LIST OF SOURCES

[1] - Oliver C. Ibe, in Markov Processes for Stochastic Modeling (Second Edition), 2013

[2] Hammersley, J. M.; Clifford, P., Markov fields on finite graphs and lattices. 1971

[3] Koralov, L., Sinai, Y.G.. Gibbs Random Fields. In: Theory of Probability and Random Processes. Universitext. Springer, Berlin, Heidelberg. 2012. https://doi.org/10.1007/978-3-540-68829-7_22

[4] Li, S. Z. . Markov Random Field Modeling in Image Analysis. In Computer Science Workbench. Springer Japan. 2001. https://doi.org/10.1007/978-4-431-67044-5

[5] Markov Random Fields W.G.H.S. Weligampola (E/14/379) June 2020

[6] Ian H. Witten, ... Christopher J. Pal, in Data Mining (Fourth Edition), 2017

[7] Shun-Zheng Yu, in Hidden Semi-Markov Models, 2016

[8] - Huawu Deng∗, David A, Unsupervised image segmentation using a simple MRF model with a new implementation scheme. Clausi Department of Systems Design Engineering, University of Waterloo, 200 University Avenue West, Waterloo, Ont., Canada N2L 3G1 Received 15 October 2003; accepted 27 April 2004

[9] - S.Z. Li, Markov Random Field Modeling in Computer Vision, Springer, New York, 2001

[10] G. Aubert, P. Kornprobst, in Encyclopedia of Mathematical Physics, 2006

[11] Mohapatra, Parthajit et al. “Color Image Segmentation Using MRF Model and Simulated Annealing.” 2006.

[12] - Jiebo Luo, Chang Wen Chen, Kevin J. Parker, "Applications of Gibbs random field in image processing: from segmentation to enhancement," Proc. SPIE 2308, Visual Communications and Image Processing '94, 1994; doi: 10.1117/12.185954

[13] Yuri Boykov, Olga Veksler and Ramin Zabih, Fast Approximate energy minimization via Graph Cuts, PAMI 2001

[14] Introduction to Markov Random Fields. In Markov Random Fields for Vision and Image Processing. The MIT Press. 2011https://doi.org/10.7551/mitpress/8579.003.0001

[15] Julien Stoehr, Richard Everitt, Matthew T. Moores. A review on statistical inference methods for discrete Markov random fields. 2017. ffhal-01462078v2f

[16] Kato, Z., & Pong, T.-C. A Markov random field image segmentation model for color textured images. In Image and Vision Computing (Vol. 24, Issue 10, pp. 1103–1114). Elsevier BV. 2006. https://doi.org/10.1016/j.imavis.2006.03.005

ANNEXES

Annex 1. Program for GRF model

```
import numpy as np
import matplotlib.pyplot as plt
from random import randint
import math
import cv2

BETA = 1

def isSafe(a,b,x,y):
    return x>=0 and x<a and y>=0 and y<b

def delta(i,l):
    if(i==l):
        return -BETA
    return BETA

def reconstruct(labs):
    labels = labs
    for i in range(len(labels)):
        for j in range(len(labels[0])):
            labels[i][j] = (labels[i][j]*255)/(SEGM-1)
    return labels

def input_bw (imagepath):
    original = cv2.imread(imagepath)
    return cv2.cvtColor(original,cv2.COLOR_BGR2GRAY)
```

```
def input_RGB (imagepath):
    return cv2.imread(imagepath)

def input_LAB (imagepath):
    original = cv2.imread(imagepath)
    return cv2.cvtColor(original.astype(np.float32)/255, cv2.COLOR_RGB2Lab)

def input_OHTA (imagepath):
    original=cv2.imread(imagepath)
    for i in original:
        for j in i:
            R=j[0]
            G=j[1]
            B=j[2]
            i1=float(R*(1/3)+G*(1/3)+B*(1/3))
            i2=float(R*(1/2)+B*(-1/2))
            i3=float(R*(-1/4)+G*(1/2)+B*(-1/4))
            j=[i1,i2,i3]
    return original


def initialize_bw(img):
    labels = np.zeros(shape=img.shape ,dtype=np.uint8)
    number = [0.0]*SEGM
    sums = [0.0]*SEGM
    squares = [0.0]*SEGM
    for i in range(len(img)):
```

```
        for j in range(len(img[0])):

            l = randint(0,SEGM-1)

            sums[l] += img[i][j]
            squares[l] += img[i][j]**2
            number[l] += 1.0
            labels[i][j] = l
    return (sums,squares,number,labels)

def initialize_colour(img):
    labels = np.zeros(shape=(img.shape[0],img.shape[1]) ,dtype=np.uint8)
    number = [0.0]*SEGM
    sums0 = [0.0]*SEGM
    squares0 = [0.0]*SEGM
    sums1 = [0.0]*SEGM
    squares1 = [0.0]*SEGM
    sums2 = [0.0]*SEGM
    squares2 = [0.0]*SEGM
    for i in range(len(img)):
        for j in range(len(img[0])):

            l = randint(0,SEGM-1)

            sums0[l] += img[i][j][0]
            sums1[l] += img[i][j][1]
            sums2[l] += img[i][j][2]
            squares0[l] += img[i][j][0]**2
```

```
            squares1[l] += img[i][j][1]**2
            squares2[l] += img[i][j][2]**2
            number[l] += 1.0
            labels[i][j] = l
    return (sums0,squares0,sums1,squares1,sums2,squares2,number,labels)

def initialize_colour_nobr(img):
    labels = np.zeros(shape=(img.shape[0],img.shape[1]) ,dtype=np.uint8)
    number = [0.0]*SEGM
    sums0 = [0.0]*SEGM
    squares0 = [0.0]*SEGM
    sums1 = [0.0]*SEGM
    squares1 = [0.0]*SEGM

    for i in range(len(img)):
        for j in range(len(img[0])):

            l = randint(0,SEGM-1)

            sums0[l] += img[i][j][0]
            sums1[l] += img[i][j][1]
            squares0[l] += img[i][j][0]**2
            squares1[l] += img[i][j][1]**2
            number[l] += 1.0
            labels[i][j] = l
    return (sums0,squares0,sums1,squares1,number,labels)

images=['doulfin.jpg','duck.jpg','mirror.jpg','yak.jpg','writing.jpg','statue.jpg','object
s1.jpg','objects.jpg',
```

```
    'mountains.jpg','manekineko.jpg','icons.png','desk.jpg',
    'curls.jpg','crayola.jpg','clothes.jpg','cat.jpg','bee.jpg',
    'rabbit.jpg','teapot.jpg','ilustration.jpg','ilustration1.jpg',
    'iitr.jpg','actors.jpeg', 'birds.jpeg','calibri.jpeg', 'catlight.jpeg',
    'deer.jpeg','eye.jpeg','facessketch.jpeg',
    'girlssketch.jpeg','oldbookshelf.jpeg','sketchbook.jpeg',
    'sketchcats.jpeg','tiger.jpeg','tigers.jpeg']



SEGM = 4
ITERATIONS = 100000000
imagepath =
img = input_bw(imagepath)
img = input_RGB (imagepath)
img = input_LAB (imagepath)



def run_black():

    variances = [variance(sums[i],squares[i],number[i]) for i in range(SEGM)]
    energy = calculateEnergy_bw(img,variances,labels)
    temp = TEMPERATURE
    it = ITERATIONS
```

```
    while it>0:
        (a,b) = img.shape
        change = False
        x = randint(0,a-1)
        y = randint(0,b-1)
        val = float(img[x][y])
        l = labels[x][y]
        newl = l
        while newl == l:
            newl = randint(0,SEGM-1)

        val = float(val)
        prevsums = sums[l] - val
        updatesums = sums[newl] + val

        prevsquares = squares[l] - val*val
        updatesquares = squares[newl] + val*val

        prevvar = variance(prevsums,prevsquares,number[l]-1)
        updatevar = variance(updatesums,updatesquares,number[newl]+1)

        newenergy = energy
        newenergy -=
math.log(math.sqrt(2*math.pi*variance(sums[l],squares[l],number[l])))+(((img[x][
y]-(sums[l]/number[l]))**2)/(2*variance(sums[l],squares[l],number[l])))



        newenergy -=
math.log(math.sqrt(2*math.pi*variance(sums[l],squares[l],number[l])))+(((img[x][
y]-(sums[l]/number[l]))**2)/(2*variance(sums[l],squares[l],number[l])))
```

```
        newenergy += math.log(math.sqrt(2*math.pi*prevvar))+(((val-
(prevsums/(number[l]-1)))**2)/(2*prevvar))

        newenergy -=
math.log(math.sqrt(2*math.pi*variance(sums[newl],squares[newl],number[newl]))
)+(((img[x][y]-
(sums[newl]/number[newl]))**2)/(2*variance(sums[newl],squares[newl],number[
newl])))

        newenergy += math.log(math.sqrt(2*math.pi*updatevar))+(((val-
(updatesums/(number[newl]+1)))**2)/(2*updatevar))


    for (p,q) in NEIGHBORHOOD:
        if isSafe(a,b,x+p,y+q):
            newenergy -= delta(l,labels[x+p][y+q])
            newenergy += delta(newl,labels[x+p][y+q])


    if newenergy < energy:
        change = True
    else:
        prob = 1.1
        if temp != 0:
            prob = np.exp((energy-newenergy)/temp)
        if prob >= (randint(0,1000)+0.0)/1000:
            change = True


    if change:
        labels[x][y] = newl
        energy = newenergy


        number[l] -= 1
        sums[l] = prevsums
```

```
            squares[l] = prevsquares

            number[newl] += 1
            sums[newl] = updatesums
            squares[newl] = updatesquares

        temp *= COOLRATE
        it -= 1




def run_colour():
    temp = TEMPERATURE
    it = ITERATIONS
    while it>0:
        (a,b) = (img.shape[0],img.shape[1])
        change = False
        x = randint(0,a-1)
        y = randint(0,b-1)
        val0 = float(img[x][y][0])
        val1 = float(img[x][y][1])
        val2 = float(img[x][y][2])
        l = labels[x][y]
        newl = l
        while newl == l:
```

```
            newl = randint(0,SEGM-1)

        val0 = float(val0)
        val1 = float(val1)
        val2 = float(val2)

        prevsums0 = sums0[l] - val0
        updatesums0 = sums0[newl] + val0

        prevsquares0 = squares0[l] - val0*val0
        updatesquares0 = squares0[newl] + val0*val0

        prevsums1 = sums1[l] - val1
        updatesums1 = sums1[newl] + val1

        prevsquares1 = squares1[l] - val1*val1
        updatesquares1 = squares1[newl] + val1*val1

        prevsums2 = sums2[l] - val2
        updatesums2 = sums2[newl] + val2

        prevsquares2 = squares2[l] - val2*val2
        updatesquares2 = squares2[newl] + val2*val2

        prevvar =
variance(prevsums0+prevsums1+prevsums2,prevsquares0+prevsquares1+prevsqua
res2,number[l]-1)
        updatevar =
variance(updatesums0+updatesums1+updatesums2,updatesquares0+updatesquares
1+updatesquares2,number[newl]+1)
```

```
        newenergy = energy
        newenergy -= math.log(math.sqrt(2*math.pi*variance(sums0[l]+sums1[l]+sums2[l],squares0[l]+squares1[l]+squares2[l],number[l])))+((((img[x][y][0]-(sums0[l]/number[l]))**2)+((img[x][y][1]-(sums1[l]/number[l]))**2)+(img[x][y][2]-(sums2[l]/number[l]))**2)/(2*variance(sums0[l]+sums1[l]+sums2[l],squares0[l]+squares1[l]+squares2[l],number[l])))
        newenergy += math.log(math.sqrt(2*math.pi*prevvar))+((((img[x][y][0]-(prevsums0/(number[l]-1)))**2)+((img[x][y][1]-(prevsums1/(number[l]-1)))**2)+(img[x][y][2]-(prevsums2/(number[l]-1))**2))/(2*prevvar))
        newenergy -= math.log(math.sqrt(2*math.pi*variance(sums0[newl]+sums1[newl]+sums2[newl],squares0[newl]+squares1[newl]+squares2[newl],number[newl])))+((((img[x][y][0]-(sums0[newl]/number[newl]))**2)+((img[x][y][1]-(sums1[newl]/number[newl]))**2)+(img[x][y][2]-(sums2[newl]/number[newl]))**2)/(2*variance(sums0[newl]+sums1[newl]+sums2[newl],squares0[newl]+squares1[newl]+squares2[newl],number[newl])))
        newenergy += math.log(math.sqrt(2*math.pi*updatevar))+((((img[x][y][0]-(updatesums0/(number[l]+1)))**2)+((img[x][y][1]-(updatesums1/(number[l]+1)))**2)+(img[x][y][2]-(updatesums2/(number[l]+1))**2))/(2*updatevar))

        for (p,q) in NEIGHBORHOOD:
            if isSafe(a,b,x+p,y+q):
                newenergy -= delta(l,labels[x+p][y+q])
                newenergy += delta(newl,labels[x+p][y+q])

        if newenergy < energy:
            change = True
        else:
            prob = 1.1
            if temp != 0:
```

```
            prob = np.exp((energy-newenergy)/temp)
        if prob >= (randint(0,1000)+0.0)/1000:
            change = True

    if change:
        labels[x][y] = newl
        energy = newenergy

        number[l] -= 1
        sums0[l] = prevsums0
        squares0[l] = prevsquares0
        sums1[l] = prevsums1
        squares1[l] = prevsquares1
        sums2[l] = prevsums2
        squares2[l] = prevsquares2

        number[newl] += 1
        sums0[newl] = updatesums0
        squares0[newl] = updatesquares0
        sums1[newl] = updatesums1
        squares1[newl] = updatesquares1
        sums2[newl] = updatesums2
        squares2[newl] = updatesquares2

    temp *= COOLRATE
    it -= 1
```

```
def run_colour_nobr():
    temp = TEMPERATURE
    it = ITERATIONS
    while it>0:
        (a,b) = (img.shape[0],img.shape[1])
        change = False
        x = randint(0,a-1)
        y = randint(0,b-1)
        val0 = float(img[x][y][0])
        val1 = float(img[x][y][1])

        l = labels[x][y]
        newl = l
        while newl == l:
            newl = randint(0,SEGM-1)

        val0 = float(val0)
        val1 = float(val1)



        prevsums0 = sums0[l] - val0
        updatesums0 = sums0[newl] + val0

        prevsquares0 = squares0[l] - val0*val0
        updatesquares0 = squares0[newl] + val0*val0
```

```
        prevsums1 = sums1[l] - val1
        updatesums1 = sums1[newl] + val1


        prevsquares1 = squares1[l] - val1*val1
        updatesquares1 = squares1[newl] + val1*val1



        prevvar =
variance(prevsums0+prevsums1,prevsquares0+prevsquares1,number[l]-1)
        updatevar =
variance(updatesums0+updatesums1,updatesquares0+updatesquares1,number[new
l]+1)


        newenergy = energy
        newenergy -=
math.log(math.sqrt(2*math.pi*variance(sums0[l]+sums1[l],squares0[l]+squares1[l
],number[l])))+((((img[x][y][0]-(sums0[l]/number[l]))**2)+((img[x][y][1]-
(sums1[l]/number[l]))**2))/(2*variance(sums0[l]+sums1[l],squares0[l]+squares1[l
],number[l])))
        newenergy += math.log(math.sqrt(2*math.pi*prevvar))+((((img[x][y][0]-
(prevsums0/(number[l]-1)))**2)+((img[x][y][1]-(prevsums1/(number[l]-
1)))**2))/(2*prevvar))
        newenergy -=
math.log(math.sqrt(2*math.pi*variance(sums0[newl]+sums1[newl],squares0[newl
]+squares1[newl],number[newl])))+((((img[x][y][0]-
(sums0[newl]/number[newl]))**2)+((img[x][y][1]-
(sums1[newl]/number[newl]))**2))/(2*variance(sums0[newl]+sums1[newl],square
s0[newl]+squares1[newl],number[newl])))
        newenergy += math.log(math.sqrt(2*math.pi*updatevar))+((((img[x][y][0]-
(updatesums0/(number[l]+1)))**2)+((img[x][y][1]-
(updatesums1/(number[l]+1)))**2))/(2*updatevar))
        for (p,q) in NEIGHBORHOOD:
            if isSafe(a,b,x+p,y+q):
```

```
            newenergy -= delta(l,labels[x+p][y+q])
            newenergy += delta(newl,labels[x+p][y+q])

    if newenergy < energy:
        change = True
    else:
        prob = 1.1
        if temp != 0:
            prob = np.exp((energy-newenergy)/temp)
        if prob >= (randint(0,1000)+0.0)/1000:
            change = True

    if change:
        labels[x][y] = newl
        energy = newenergy

        number[l] -= 1
        sums0[l] = prevsums0
        squares0[l] = prevsquares0
        sums1[l] = prevsums1
        squares1[l] = prevsquares1



        number[newl] += 1
        sums0[newl] = updatesums0
        squares0[newl] = updatesquares0
        sums1[newl] = updatesums1
        squares1[newl] = updatesquares1
```

```
        temp *= COOLRATE
        it -= 1




def run_colour_clr():
    variances =
[variance(sums0[i]+sums1[i]+sums2[i],squares0[i]+squares1[i]+squares2[i],numbe
r[i]) for i in range(SEGM)]
    energy = calculateEnergy_clr(img,variances,labels)
    temp = TEMPERATURE
    it = ITERATIONS
    while it>0:
        (a,b) = (img.shape[0],img.shape[1])
        change = False
        x = randint(0,a-1)
        y = randint(0,b-1)
        val0 = float(img[x][y][0])
        val1 = float(img[x][y][1])
        val2 = float(img[x][y][2])
        l = labels[x][y]
        newl = l
        while newl == l:
            newl = randint(0,SEGM-1)
```

```
        val0 = float(val0)
        val1 = float(val1)
        val2 = float(val2)

        prevsums0 = sums0[l] - val0
        updatesums0 = sums0[newl] + val0

        prevsquares0 = squares0[l] - val0*val0
        updatesquares0 = squares0[newl] + val0*val0

        prevsums1 = sums1[l] - val1
        updatesums1 = sums1[newl] + val1

        prevsquares1 = squares1[l] - val1*val1
        updatesquares1 = squares1[newl] + val1*val1

        prevsums2 = sums2[l] - val2
        updatesums2 = sums2[newl] + val2

        prevsquares2 = squares2[l] - val2*val2
        updatesquares2 = squares2[newl] + val2*val2

        prevvar =
variance(prevsums0+prevsums1+prevsums2,prevsquares0+prevsquares1+prevsqua
res2,number[l]-1)
        updatevar =
variance(updatesums0+updatesums1+updatesums2,updatesquares0+updatesquares
1+updatesquares2,number[newl]+1)

        newenergy = energy
```

```
        newenergy -=
beta_variances*math.log(math.sqrt(2*math.pi*variance(sums0[l]+sums1[l]+sums2
[l],squares0[l]+squares1[l]+squares2[l],number[l])))+((((img[x][y][0]-
(sums0[l]/number[l]))**2)+((img[x][y][1]-
(sums1[l]/number[l]))**2)+(img[x][y][2]-
(sums2[l]/number[l]))**2)/(2*variance(sums0[l]+sums1[l]+sums2[l],squares0[l]+s
quares1[l]+squares2[l],number[l])))

        newenergy +=
beta_variances*math.log(math.sqrt(2*math.pi*prevvar))+((((img[x][y][0]-
(prevsums0/(number[l]-1)))**2)+((img[x][y][1]-(prevsums1/(number[l]-
1)))**2)+(img[x][y][2]-(prevsums2/(number[l]-1))**2))/(2*prevvar))

        newenergy -=
beta_variances*math.log(math.sqrt(2*math.pi*variance(sums0[newl]+sums1[newl
]+sums2[newl],squares0[newl]+squares1[newl]+squares2[newl],number[newl])))+
((((img[x][y][0]-(sums0[newl]/number[newl]))**2)+((img[x][y][1]-
(sums1[newl]/number[newl]))**2)+(img[x][y][2]-
(sums2[newl]/number[newl]))**2)/(2*variance(sums0[newl]+sums1[newl]+sums2
[newl],squares0[newl]+squares1[newl]+squares2[newl],number[newl])))

        newenergy +=
beta_variances*math.log(math.sqrt(2*math.pi*updatevar))+((((img[x][y][0]-
(updatesums[l]/(number[l]+1)))**2)+((img[x][y][1]-
(updatesums1/(number[l]+1)))**2)+(img[x][y][2]-
(updatesums2/(number[l]+1))**2))/(2*updatevar))


        for (p,q) in NEIGHBORHOOD:
            if isSafe(a,b,x+p,y+q):
                newenergy -= beta_neighbours*delta(l,labels[x+p][y+q])
                newenergy += beta_neighbours*delta(newl,labels[x+p][y+q])


        if newenergy < energy:
            change = True
        else:
            prob = 1.1
            if temp != 0:
                prob = np.exp((energy-newenergy)/temp)
```

```
    if prob >= (randint(0,1000)+0.0)/1000:
        change = True

if change:
    labels[x][y] = newl
    energy = newenergy

    number[l] -= 1
    sums0[l] = prevsums0
    squares0[l] = prevsquares0
    sums1[l] = prevsums1
    squares1[l] = prevsquares1
    sums2[l] = prevsums2
    squares2[l] = prevsquares2

    number[newl] += 1
    sums0[newl] = updatesums0
    squares0[newl] = updatesquares0
    sums1[newl] = updatesums1
    squares1[newl] = updatesquares1
    sums2[newl] = updatesums2
    squares2[newl] = updatesquares2

temp *= COOLRATE
it -= 1
```

```
def run_colour_clr2():
    variances =
[variance(sums0[i]+sums1[i]+sums2[i],squares0[i]+squares1[i]+squares2[i],numbe
r[i]) for i in range(SEGM)]
    energy = calculateEnergy_clr2(img,variances,labels)
    temp = TEMPERATURE
    it = ITERATIONS
    while it>0:

        (a,b) = (img.shape[0],img.shape[1])
        change = False
        x = randint(0,a-1)
        y = randint(0,b-1)
        val0 = float(img[x][y][0])
        val1 = float(img[x][y][1])
        val2 = float(img[x][y][2])
        l = labels[x][y]
        newl = l
        while newl == l:
            newl = randint(0,SEGM-1)

        val0 = float(val0)
        val1 = float(val1)
        val2 = float(val2)

        prevsums0 = sums0[l] - val0
        updatesums0 = sums0[newl] + val0
```

```
        prevsquares0 = squares0[l] - val0*val0
        updatesquares0 = squares0[newl] + val0*val0

        prevsums1 = sums1[l] - val1
        updatesums1 = sums1[newl] + val1

        prevsquares1 = squares1[l] - val1*val1
        updatesquares1 = squares1[newl] + val1*val1

        prevsums2 = sums2[l] - val2
        updatesums2 = sums2[newl] + val2

        prevsquares2 = squares2[l] - val2*val2
        updatesquares2 = squares2[newl] + val2*val2

        prevvar =
variance(prevsums0+prevsums1+prevsums2,prevsquares0+prevsquares1+prevsqua
res2,number[l]-1)
        updatevar =
variance(updatesums0+updatesums1+updatesums2,updatesquares0+updatesquares
1+updatesquares2,number[newl]+1)

        newenergy = energy

        newenergy -=
beta_variances*math.log(math.sqrt(2*math.pi*variance(sums0[l]+sums1[l]+sums2
[l],squares0[l]+squares1[l]+squares2[l],number[l])))+((((img[x][y][0]-
(sums0[l]/number[l]))**2)+((img[x][y][1]-
(sums1[l]/number[l]))**2)+(img[x][y][2]-
(sums2[l]/number[l]))**2)/(2*variance(sums0[l]+sums1[l]+sums2[l],squares0[l]+s
quares1[l]+squares2[l],number[l])))
```

```
        newenergy +=
beta_variances*math.log(math.sqrt(2*math.pi*prevvar))+((((img[x][y][0]-
(prevsums0/(number[l]-1)))**2)+((img[x][y][1]-(prevsums1/(number[l]-
1)))**2)+(img[x][y][2]-(prevsums2/(number[l]-1))**2))/(2*prevvar))

        newenergy -=
beta_variances*math.log(math.sqrt(2*math.pi*variance(sums0[newl]+sums1[newl
]+sums2[newl],squares0[newl]+squares1[newl]+squares2[newl],number[newl])))+
((((img[x][y][0]-(sums0[newl]/number[newl]))**2)+((img[x][y][1]-
(sums1[newl]/number[newl]))**2)+(img[x][y][2]-
(sums2[newl]/number[newl]))**2)/(2*variance(sums0[newl]+sums1[newl]+sums2
[newl],squares0[newl]+squares1[newl]+squares2[newl],number[newl])))

        newenergy +=
beta_variances*math.log(math.sqrt(2*math.pi*updatevar))+((((img[x][y][0]-
(updatesums0/(number[l]+1)))**2)+((img[x][y][1]-
(updatesums1/(number[l]+1)))**2)+(img[x][y][2]-
(updatesums2/(number[l]+1))**2))/(2*updatevar))


        for (p,q) in NEIGHBORHOOD:
            if isSafe(a,b,x+p,y+q):
                newenergy -= beta_neighbours*delta(l,labels[x+p][y+q])
                newenergy += beta_neighbours*delta(newl,labels[x+p][y+q])


        for (p,q) in NEIGHBORHOOD:
                if isSafe(a,b,x+p,y+q):
                    t=0
                    temp=img[x][y][0]-img[x+p][y+q][0]
                    if temp<coefcol0 and temp>-coefcol0:
                        t=t+1
                    temp=img[x][y][1]-img[x+p][y+q][1]
                    if temp<coefcol1 and temp>-coefcol1:
                        t=t+1
                    temp=img[x][y][2]-img[x+p][y+q][2]
                    if temp<coefcol2 and temp>-coefcol2:
```

```
            t=t+1

        if t>=diffort and l==labels[x+p][y+q]:
            newenergy -= beta_neighbours*((-BETA)/2.0)
        else:
            newenergy += beta_neighbours*((-BETA)/2.0)
        if t>=diffort and newl==labels[x+p][y+q]:
            newenergy -= beta_neighbours*((BETA)/2.0)
        else:
            newenergy += beta_neighbours*((BETA)/2.0)

if newenergy < energy:
    change = True
else:
    prob = 1.1
    if temp != 0:
        prob = np.exp((energy-newenergy)/temp)
    if prob >= (randint(0,1000)+0.0)/1000:
        change = True

if change:
    labels[x][y] = newl
    energy = newenergy

    number[l] -= 1
    sums0[l] = prevsums0
    squares0[l] = prevsquares0
    sums1[l] = prevsums1
```

```
            squares1[l] = prevsquares1
            sums2[l] = prevsums2
            squares2[l] = prevsquares2

            number[newl] += 1
            sums0[newl] = updatesums0
            squares0[newl] = updatesquares0
            sums1[newl] = updatesums1
            squares1[newl] = updatesquares1
            sums2[newl] = updatesums2
            squares2[newl] = updatesquares2

        temp *= COOLRATE
        it -= 1




def run_colour_clr22():
    variances =
[variance(sums0[i]+sums1[i]+sums2[i],squares0[i]+squares1[i]+squares2[i],numbe
r[i]) for i in range(SEGM)]
    energy = calculateEnergy_clr22(img,variances,labels)
    temp = TEMPERATURE
    it = ITERATIONS
    while it>0:
        (a,b) = (img.shape[0],img.shape[1])
        change = False
```

```
x = randint(0,a-1)
y = randint(0,b-1)
val0 = float(img[x][y][0])
val1 = float(img[x][y][1])
val2 = float(img[x][y][2])
l = labels[x][y]
newl = l
while newl == l:
    newl = randint(0,SEGM-1)

val0 = float(val0)
val1 = float(val1)
val2 = float(val2)

prevsums0 = sums0[l] - val0
updatesums0 = sums0[newl] + val0

prevsquares0 = squares0[l] - val0*val0
updatesquares0 = squares0[newl] + val0*val0

prevsums1 = sums1[l] - val1
updatesums1 = sums1[newl] + val1

prevsquares1 = squares1[l] - val1*val1
updatesquares1 = squares1[newl] + val1*val1

prevsums2 = sums2[l] - val2
updatesums2 = sums2[newl] + val2
```

```
        prevsquares2 = squares2[l] - val2*val2
        updatesquares2 = squares2[newl] + val2*val2


        prevvar =
variance(prevsums0+prevsums1+prevsums2,prevsquares0+prevsquares1+prevsqua
res2,number[l]-1)
        updatevar =
variance(updatesums0+updatesums1+updatesums2,updatesquares0+updatesquares
1+updatesquares2,number[newl]+1)


        newenergy = energy
        newenergy -=
math.log(math.sqrt(2*math.pi*variance(sums0[l]+sums1[l]+sums2[l],squares0[l]+
squares1[l]+squares2[l],number[l])))+((((img[x][y][0]-
(sums0[l]/number[l]))**2)+((img[x][y][1]-
(sums1[l]/number[l]))**2)+(img[x][y][2]-
(sums2[l]/number[l]))**2)/(2*variance(sums0[l]+sums1[l]+sums2[l],squares0[l]+s
quares1[l]+squares2[l],number[l])))
        newenergy += math.log(math.sqrt(2*math.pi*prevvar))+((((img[x][y][0]-
(prevsums0/(number[l]-1)))**2)+((img[x][y][1]-(prevsums1/(number[l]-
1)))**2)+(img[x][y][2]-(prevsums2/(number[l]-1))**2))/(2*prevvar))
        newenergy -=
math.log(math.sqrt(2*math.pi*variance(sums0[newl]+sums1[newl]+sums2[newl],
squares0[newl]+squares1[newl]+squares2[newl],number[newl])))+((((img[x][y][0]
-(sums0[newl]/number[newl]))**2)+((img[x][y][1]-
(sums1[newl]/number[newl]))**2)+(img[x][y][2]-
(sums2[newl]/number[newl]))**2)/(2*variance(sums0[newl]+sums1[newl]+sums2
[newl],squares0[newl]+squares1[newl]+squares2[newl],number[newl])))
        newenergy += math.log(math.sqrt(2*math.pi*updatevar))+((((img[x][y][0]-
(updatesums0/(number[l]+1)))**2)+((img[x][y][1]-
(updatesums1/(number[l]+1)))**2)+(img[x][y][2]-
(updatesums2/(number[l]+1))**2))/(2*updatevar))
        for (p,q) in NEIGHBORHOOD:
            if isSafe(a,b,x+p,y+q):
```

```
            newenergy -= beta_neighbours*delta(l,labels[x+p][y+q])
            newenergy += beta_neighbours*delta(newl,labels[x+p][y+q])

        for (p,q) in NEIGHBORHOOD:
            if isSafe(a,b,x+p,y+q):
                t=0
                tb=0
                temp=float(img[x][y][0])-float(img[x+p][y+q][0])
                if (temp<coefcol0 and temp>-coefcol0) :
                    t=t+1
                if (temp>difbord0 and temp<-difbord0) :
                    tb=tb+1
                temp=float(img[x][y][1])-float(img[x+p][y+q][1])
                if (temp<coefcol1 and temp>-coefcol1) :
                    tb=tb+1
                if (temp>difbord1 and temp<-difbord1) :
                    tb=tb+1
                temp=float(img[x][y][2])-float(img[x+p][y+q][2])
                if (temp<coefcol2 and temp>-coefcol2) :
                    t=t+1
                if (temp>difbord2 and temp<-difbord2) :
                    tb=tb+1



                if (t>=diffort and l==labels[x+p][y+q]) or (tb>=diffort and
l!=labels[x+p][y+q]):
                    newenergy -= beta_neighbours*((-BETA)/2.0)
                else:
```

```
                    newenergy += beta_neighbours*((-BETA)/2.0)
                if (t>=diffort and newl==labels[x+p][y+q])or (tb>=diffort and
newl!=labels[x+p][y+q]):
                    newenergy -= beta_neighbours*((BETA)/2.0)
                else:
                    newenergy += beta_neighbours*((BETA)/2.0)

        if newenergy < energy:
            change = True
        else:
            prob = 1.1
            if temp != 0:
                prob = np.exp((energy-newenergy)/temp)
            if prob >= (randint(0,1000)+0.0)/1000:
                change = True

        if change:
            labels[x][y] = newl
            energy = newenergy

            number[l] -= 1
            sums0[l] = prevsums0
            squares0[l] = prevsquares0
            sums1[l] = prevsums1
            squares1[l] = prevsquares1
            sums2[l] = prevsums2
            squares2[l] = prevsquares2

            number[newl] += 1
```

```
            sums0[newl] = updatesums0
            squares0[newl] = updatesquares0
            sums1[newl] = updatesums1
            squares1[newl] = updatesquares1
            sums2[newl] = updatesums2
            squares2[newl] = updatesquares2

        temp *= COOLRATE
        it -= 1




def save (colour, energy_function,neighbours):
    string=""
    if colour==0: #blackwhite
        string=imagepath+'_segm-'+str(SEGM)+'_neighbours-
'+str(neighbours)+'_bw_basic_'+'_iter-'+str(ITERATIONS)+'_T-
'+str(TEMPERATURE)+'_'
    else:
        col="
        if colour==1: #rgb
            col='_RGB_'
        if colour==2: #lab
            col="_LAB_"
        else:
            col="_OHTA_"
        if energy_function=="no_br" or energy_function=="basic":
```

```
            string=imagepath+'_segm-'+str(SEGM)+'_neighbours-
'+str(neighbours)+'_'+col+'_iter-'+str(ITERATIONS)+'_T-
'+str(TEMPERATURE)+'_e-'+energy_function
        else:
            string=imagepath+'_segm-'+str(SEGM)+'_neighbours-
'+str(neighbours)+'_'+col+'coefcol-'+str(coefcol)+'_coefb-
'+str(coefbrightnes)+'_dif-'+str(diffort)+'_bv-'+str(beta_variances)+'_bn2-
'+str(beta_neighbours)+'_iter-'+str(ITERATIONS)+'_T-
'+str(TEMPERATURE)+'_e-'+energy_function
    plt.imshow(reconstruct(labels) , interpolation='nearest',cmap='gray')
    cv2.imwrite(string+'.jpg',labels)
    print(string)



def variance(sums1,squares1,number1):
    t=(squares1-((sums1**2)/number1)/number1)
    return t



sums0,squares0,sums1,squares1,sums2,squares2,number,labels = initialize(img)
variances =
[variance(sums0[i]+sums1[i]+sums2[i],squares0[i]+squares1[i]+squares2[i],numbe
r[i]) for i in range(SEGM)]


def calculateEnergy_bw(img,variances,labels):
    energy = 0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
```

```
            energy += math.log(math.sqrt(2*math.pi*variances[l]))+(((img[i][j]-
(sums[l]/number[l]))**2)/(2*variances[l]))
            for (p,q) in NEIGHBORHOOD:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    energy += (delta(l,labels[i+p][j+q]))
    return energy


def calculateEnergy_colour(img,variances,labels):
    energy = 0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
            energy += math.log(math.sqrt(2*math.pi*variances[l]))
            energy += ((((img[i][j][0]-(sums0[l]/number[l]))**2)+
                    ((img[i][j][1]-(sums1[l]/number[l]))**2)+((img[i][j][2]-
(sums2[l]/number[l]))**2))/(2*variances[l]))
            for (p,q) in NEIGHBORHOOD:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    energy += (delta(l,labels[i+p][j+q])/2.0)
    return energy


def calculateEnergy_clr(img,variances,labels):
    energy = 0.0
    energy2=0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
```

```
            energy += math.log(math.sqrt(2*math.pi*variances[l]))
            energy += ((((img[i][j][0]-(sums0[l]/number[l]))**2)+
                    ((img[i][j][1]-(sums1[l]/number[l]))**2)+((img[i][j][2]-
(sums2[l]/number[l]))**2))/(2*variances[l]))
            for (p,q) in NEIGHBORHOOD:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    energy2 += (delta(l,labels[i+p][j+q])/2.0)
    return beta_variances*energy+beta_neighbours*energy2



def calculateEnergy_clr2(img,variances,labels):
    energy = 0.0
    energy2=0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
            energy += math.log(math.sqrt(2*math.pi*variances[l]))
            energy += ((((img[i][j][0]-(sums0[l]/number[l]))**2)+
                    ((img[i][j][1]-(sums1[l]/number[l]))**2)+((img[i][j][2]-
(sums2[l]/number[l]))**2))/(2*variances[l]))
            for (p,q) in NEIGHBORHOOD:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    t=0
                    if img[i][j][0]-img[i+p][j+q][0]<coef and img[i][j][0]-
img[i+p][j+q][0]>coef:
                        t=t+1
                    if img[i][j][1]-img[i+p][j+q][1]<coef and img[i][j][1]-
img[i+p][j+q][1]>coef:
                        t=t+1
```

```
                if img[i][j][2]-img[i+p][j+q][2]<coefbrightnes and img[i][j][2]-
img[i+p][j+q][2]>coefbrightnes:
                    t=t+1
                delt=delta(l,labels[i+p][j+q])
                if t>=diffort and l==labels[i+p][j+q]:
                    energy2 += ((-BETA))
                else:
                    energy2 += ((BETA))



    return beta_variances*energy+beta_neighbours*energy2



def calculateEnergy_clr22(img,variances,labels):
    energy = 0.0
    energy2=0.0
    for i in range(len(img)):
        for j in range(len(img[0])):
            l = labels[i][j]
            energy += math.log(math.sqrt(2*math.pi*variances[l]))
            energy += ((((img[i][j][0]-(sums0[l]/number[l]))**2)+
                    ((img[i][j][1]-(sums1[l]/number[l]))**2)+((img[i][j][2]-
(sums2[l]/number[l]))**2))/(2*variances[l]))
            for (p,q) in NEIGHBORHOOD:
                if isSafe(img.shape[0],img.shape[1],i+p,j+q):
                    t=0
                    tb=0
                    temp=float(img[i][j][0])-float(img[i+p][j+q][0])
                    if (temp<coefcol0 and temp>-coefcol0) :
```

```
                t=t+1
            if (temp>difbord0 and temp<-difbord0) :
                tb=tb+1
            temp=float(img[i][j][1])-float(img[i+p][j+q][1])
            if (temp<coefcol1 and temp>-coefcol1) :
                t=t+1
            if (temp>difbord1 and temp<-difbord1) :
                tb=tb+1
            temp=float(img[i][j][2])-float(img[i+p][j+q][2])
            if (temp<coefcol2 and temp>-coefcol2) :
                t=t+1
            if (temp>difbord2 and temp<-difbord2) :
                tb=tb+1

            if (t>=diffort and l==labels[i+p][j+q]) or (tb>=diffort and
l!=labels[i+p][j+q]):
                energy2 += ((-BETA))
            else:
                energy2 += ((BETA))


    return beta_variances*energy+beta_neighbours*energy2


TEMPERATURE = 4.0
COOLRATE = 0.95
#NEIGHBORHOOD = [(-1,0) , (1,0) , (0,-1) , (0,1),(-1,1) , (1,1) , (1,-1) , (-1,-1)]
NEIGHBORHOOD = [(-1,0) , (1,0) , (0,-1) , (0,1)]
SEGMs=[2,3,4,5,10]
```

```
iterationss=[1000000,2000000,3000000,5000000]#100000,]
for it in iterationss:
    for s in SEGMs:
        for im in images:
            print('it-'+str(it)+'_s-'+str(s)+'_im-'+str(im))
            SEGM = s
            ITERATIONS = it
            imagepath = im
            img = input_bw(imagepath)
            sums,squares,number,labels = initialize_bw(img)
            #img = input_RGB (imagepath)
            #img = input_LAB (imagepath)
            run_black()
            save (0, "bw",len(NEIGHBORHOOD))



TEMPERATURE = 8.0
COOLRATE = 0.95

b_var=[0.25,0.5,0.75,0.9,1,0,0.1,]
coef_col=[5,1,10,15,10,50]
coef_border=[200, 100,50]
coef_br=[0.01,0.25,0.5,0.75,1,5,10, 15,0.01]
dif_t=[2,3,1,0]

SEGMs=[5,4,3,2,6,10]
iterationss=[1000000,2000000,3000000,5000000,7000000]#
```

```
for it in iterationss:
    for var in b_var:
        for cl0 in coef_col:
            for cl1 in coef_col:
                for cl2 in coef_col:
                    for dt in dif_t:
                        for s in SEGMs:
                            for im in images:
                                print('it-'+str(it)+'_s-'+str(s)+'_im-'+str(im))
                                beta_variances=var
                                beta_neighbours=1-beta_variances
                                coefcol0=cl0
                                coefcol1=cl1
                                coefcol2=cl2
                                diffort=dt

                                SEGM = s
                                ITERATIONS = it
                                imagepath = im
                                img = input_RGB (imagepath)
                                sums0,squares0,sums1,squares1,sums2,squares2,number,labels
= initialize_colour(img)
                                run_colour_clr2()
                                save (1, "clr2")
                                SEGM = s
                                ITERATIONS = it
                                imagepath = im
                                img = input_LAB (imagepath)
```

```
                        sums0,squares0,sums1,squares1,sums2,squares2,number,labels
= initialize_colour(img)
                        run_colour_clr2()
                        save (2, "clr2")



TEMPERATURE = 5.0
COOLRATE = 0.95

b_var=[0.75,0.9,0.5,0.1,0.25,0,1]
coef_col=[0.01,0.025,0.1,0.2,0.5,0.75,1,1.5,0.05,0.075, 2,5]
coef_border=[50]#1,5,10,15,20,30,40,50]
dif_t=[3,2,1,0]

SEGMs=[3,4,2,5,6,10]
iterationss=[1000000,2000000,3000000,5000000,7000000,10000000]#
for it in iterationss:
    for var in b_var:
        for cl0 in coef_col:
            for cl1 in coef_col:
                for cl2 in coef_col:
                    for cb0 in coef_border:
                        for cb1 in coef_border:
                            for cb2 in coef_border:
                                for dt in dif_t:
                                    for s in SEGMs:
                                        for im in images:
                                            beta_variances=var
                                            beta_neighbours=1-beta_variances
```

```
                    coefcol0=cl0
                    coefcol1=cl1
                    coefcol2=cl2
                    difbord0=cb0
                    difbord1=cb1
                    difbord2=cb2
                    if cl0>=cb0 or cl1>=cb1 or cl2>=cb2:
                       break
                    print('it-'+str(it)+'_s-'+str(s)+'_im-'+str(im))
                    diffort=dt


                    SEGM = s
                    ITERATIONS = it
                    imagepath = im
                    img = input_RGB (imagepath)

sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
                    run_colour_clr22()
                    save (1, "clr22")
                    SEGM = s
                    ITERATIONS = it
                    imagepath = im
                    img = input_LAB (imagepath)

sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
                    run_colour_clr22()
                    save (2, "clr22")
```

```
TEMPERATURE = 5.0
COOLRATE = 0.95
NEIGHBORHOOD = [(-1,0) , (1,0) , (0,-1) , (0,1)]
#NEIGHBORHOOD = [(-1,0) , (1,0) , (0,-1) , (0,1),(-1,1) , (1,1) , (1,-1) , (-1,-1)]
b_var=[0.25,0.5,0.75,0.9,1,0,0.1,]
coef_col=[10,5,2,1,0.1,0.25,0.5,0.75,0.001]
coef_br=[20,15,10,5,2,1,0.1,0.25,0.5,0.75,0.001]
dif_t=[2,3,1,0]

SEGMs=[2,3,4,5,6,10]
iterationss=[100000,1000000,2000000,3000000,5000000]#
for it in iterationss:
    for var in b_var:
        for cl in coef_col:
            for cb in coef_br:
                for dt in dif_t:
                    for s in SEGMs:
                        for im in images:
                            print('it-'+str(it)+'_s-'+str(s)+'_im-'+str(im))
                            beta_variances=var
                            beta_neighbours=1-beta_variances
                            coefcol=cl
                            coefbrightnes=cb
                            diffort=dt
                            SEGM = s
                            ITERATIONS = it
                            imagepath = im
```

```
                img = input_RGB (imagepath)
                sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
                run_colour_clr()
                save (1, "clr",len(NEIGHBORHOOD))
                SEGM = s
                ITERATIONS = it
                imagepath = im
                img = input_LAB (imagepath)
                sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
                run_colour_clr()
                save (2, "clr",len(NEIGHBORHOOD))
                SEGM = s
                ITERATIONS = it
                imagepath = im
                img = input_OHTA (imagepath)
                sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
                run_colour_clr()
                save (3, "clr",len(NEIGHBORHOOD))
                SEGM = s
                ITERATIONS = it
                imagepath = im
                img = input_RGB (imagepath)
                sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
                run_colour_clr2()
                save (1, "clr2",len(NEIGHBORHOOD))
                SEGM = s
```

```
                    ITERATIONS = it
                    imagepath = im
                    img = input_LAB (imagepath)
                    sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
                    run_colour_clr2()
                    save (2, "clr2",len(NEIGHBORHOOD))
                    SEGM = s
                    ITERATIONS = it
                    imagepath = im
                    img = input_OHTA (imagepath)
                    sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
                    run_colour_clr2()
                    save (3, "clr2",len(NEIGHBORHOOD))



TEMPERATURE = 4.0
COOLRATE = 0.95
SEGMs=[2,3,4,5,10]
iterationss=[1000000]#100000,1000000,
for it in iterationss:
    for s in SEGMs:
        for im in images:
            print('it-'+str(it)+'_s-'+str(s)+'_im-'+str(im))
            SEGM = s
            ITERATIONS = it
            imagepath = im
            img = input_RGB (imagepath)
```

```
        sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
        run_colour_basic()
        save (1, "basic")
        SEGM = s
        ITERATIONS = it
        imagepath = im
        img = input_LAB (imagepath)
        sums0,squares0,sums1,squares1,sums2,squares2,number,labels =
initialize_colour(img)
        run_colour_basic()
        save (2, "basic")
        SEGM = s
        ITERATIONS = it
        imagepath = im
        img = input_LAB (imagepath)
        sums0,squares0,sums1,squares1,number,labels =
initialize_colour_nobr(img)
        run_colour_nobr()
        save (2, "no_br")
```

Annex 2. Segmented images (file attached)