# PAMI homework 4: Clustering

*Egor Makhov 10493074*

```
library("jpeg")
library("png")
library("graphics")
library("ggplot2")
library("grid")
library("gridExtra")

# Kmeans based segmenter
segment_image = function(img, n, alg="Hartigan-Wong", itm=10){
  # create a flat, segmented image data set using kmeans
  # Segment an RGB image into n groups based on color values using Kmeans
  df = data.frame(
    red = matrix(img[,,1], ncol=1),
    green = matrix(img[,,2], ncol=1),
    blue = matrix(img[,,3], ncol=1)
  )

  K = kmeans(df,centers=n,iter.max=itm,algorithm=alg)
  # the function returns a structure whose attribute "cluster" contains the labels
  df$label = K$cluster

  # compute rgb values and color codes based on Kmeans centers
  colors = data.frame(
    label = 1:nrow(K$centers),
    R = K$centers[,"red"],
    G = K$centers[,"green"],
    B = K$centers[,"blue"],
    color=rgb(K$centers)
  )

  # merge color codes on to df but maintain the original order of df
  df$order = 1:nrow(df)
  df = merge(df, colors)
  df = df[order(df$order),]
  df$order = NULL

  return(df)
}

#
# reconstitue the segmented images to RGB matrix
#
build_segmented_image = function(df, img){
  # reconstitue the segmented images to RGB array

  # get mean color channel values for each row of the df.
  R = matrix(df$R, nrow=dim(img)[1])
  G = matrix(df$G, nrow=dim(img)[1])
  B = matrix(df$B, nrow=dim(img)[1])
```

```
  # reconsitute the segmented image in the same shape as the input image
  img_segmented = array(dim=dim(img))
  img_segmented[,,1] = R
  img_segmented[,,2] = G
  img_segmented[,,3] = B

  return(img_segmented)
}

#
# 2D projection for visualizing the kmeans clustering
#
project2D_from_RGB = function(df){
  # Compute the projection of the RGB channels into 2D
  PCA = prcomp(df[,c("red","green","blue")], center=TRUE, scale=TRUE)
  pc2 = PCA$x[,1:2]
  df$x = pc2[,1]
  df$y = pc2[,2]
  return(df[,c("x","y","label","R","G","B", "color")])
}

#
# Create the projection plot of the clustered segments
#
plot_projection <- function(df, sample.size){
  # plot the projection of the segmented image data in 2D, using the
  # mean segment colors as the colors for the points in the projection
  index = sample(1:nrow(df), sample.size)
  return(ggplot(df[index,], aes(x=x, y=y, col=color)) + geom_point(size=2) + scale_color_identity())
}

#
# Inspect
#
inspect_segmentation <- function(image.raw, image.segmented, image.proj){
  # helper function to review the results of segmentation visually
  img1 = rasterGrob(image.raw)
  img2 = rasterGrob(image.segmented)
  plt = plot_projection(image.proj, 50000)
  grid.arrange(arrangeGrob(img1,img2, nrow=1),plt,heights=1:2, widths=3:4)
}
```

Welcome to the fourth PAMI demo/homework (year 2016): clustering

This year we are focusing on a single algorithm: K-Means. In particular, we are going to use a particular application (color quantization) to study some of its properties.

To do this, we are relying on a recent demo which is published and described in detail at the following URL:

http://www.r-bloggers.com/color-quantization-in-r/

Check it out and get ready to answer the following questions!

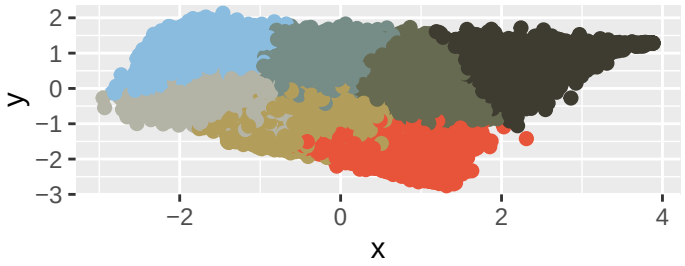
The first task you are assigned is to understand how k-means is used to perform color quantization. To do this, you first need to complete the code in the segment_image function (adding a call to the k-means function you find in the "stats" library), then answer the following question:

**Q1) What does the segment_image function do? What are the operations to be performed if you want to do color quantization using k-means?**

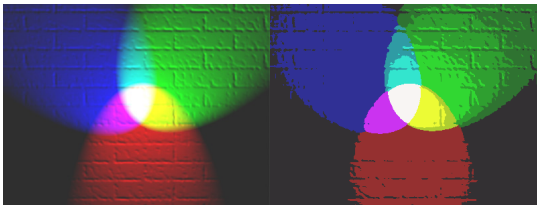
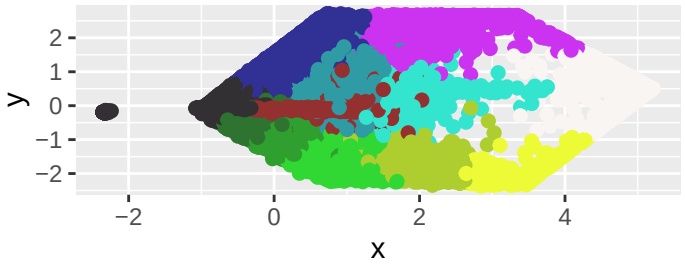
Fuction divides our image into pixels (for mandrill it will be 512*512=262144pxls) and segments them into n clusters(colors). So to do that you need to: 1. Divide image into pixels and create dataframe containing their color (as red, green, blue colums). 2. Cluster them using k-means into k clusters. 3. Create a new dataframe with new colors of our pixels (in our case we creating 3 colums(R,G,B)(issue of R image represintation) and final color(we will not use it in future)) 4. Build a segmented image using second df (in our case we merged 2 df into a big one and building segmented in another function).

---

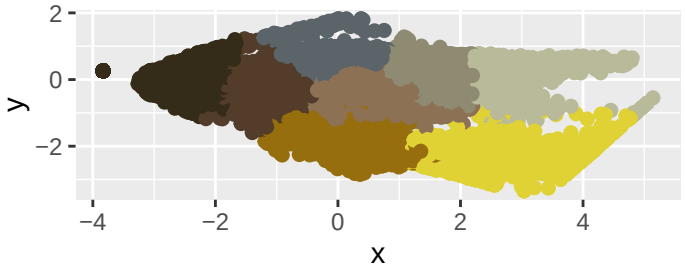
```
# some interesting sample images -- download them if they aren't in the current working directory
if(!file.exists("mandrill.png")){
  download.file(url = "http://graphics.cs.williams.edu/data/images/mandrill.png", destfile="mandrill.png
  download.file(url = "https://upload.wikimedia.org/wikipedia/commons/2/28/RGB_illumination.jpg", destfi
  download.file(url = "http://r0k.us/graphics/kodak/kodak/kodim03.png", destfile="kodim03.png")
  download.file(url = "http://r0k.us/graphics/kodak/kodak/kodim22.png", destfile="kodim22.png")
}

# we can work with both JPEGs and PNGS.  For simplicty, we'll always write out to PNG though.
mandrill <- readPNG("mandrill.png")
rgb <- readJPEG("RGB_illumination.jpg")
hats <- readPNG("kodim03.png")
barn <- readPNG("kodim22.png")

# segment -- tune the number of segments for each image
mandrill.df = segment_image(mandrill, 7)
rgb.df = segment_image(rgb, 12)
hats.df = segment_image(hats, 8)
barn.df = segment_image(barn, 10)

# project RGB channels
mandrill.proj = project2D_from_RGB(mandrill.df)
rgb.proj = project2D_from_RGB(rgb.df)
hats.proj = project2D_from_RGB(hats.df)
barn.proj = project2D_from_RGB(barn.df)

# create segmented image data structure and write to disk
mandrill.segmented = build_segmented_image(mandrill.df, mandrill)
rgb.segmented = build_segmented_image(rgb.df, rgb)
hats.segmented = build_segmented_image(hats.df, hats)
barn.segmented = build_segmented_image(barn.df, barn)

# write the segmented images to disk
writePNG(mandrill.segmented, "mandrill_segmented.png" )
writePNG(rgb.segmented, "rgb_illumination_segmented.png")
writePNG(hats.segmented, "kodim03_segmented.png")
writePNG(barn.segmented, "kodim22_segmented.png")

# inspect the results
inspect_segmentation(mandrill, mandrill.segmented, mandrill.proj)
```

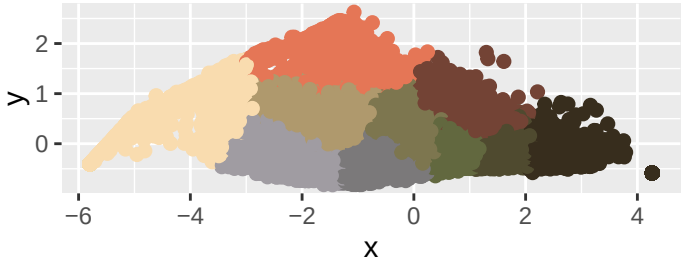

```
inspect_segmentation(rgb, rgb.segmented, rgb.proj)
```


```
inspect_segmentation(hats, hats.segmented, hats.proj)
```


```
inspect_segmentation(barn, barn.segmented, barn.proj)
```


Now that you ran the full demo, you should have the quantized images saved on your disk, and four windows showing you the results of the quantization (both on the colormap and on the final image). Try to answer the following questions:

**Q2: What are the observations in this particular application and what is the dimensionality of your problem?**

To perform Color Quantization we are projecting all pixels of image into RGB space and clustering them into k clusters. So the observations are color of our pixels and dimesionality = 3 (Red,Green,Blue colors).

**Q3) Try to run the following code many times and comment the results: are they always the same or not? Does this behavior confirm your expectations? Now try to do the same after modifying the call to k-means as kmeans(df,n, algorithm = "MacQueen"). Do results change now or not? What can you deduce from this (the fact that in some cases the two algorithms behave differently)?**

HINT: differences might not be visible at a glance. Try to use a more robust way to check them out, e.g. if you have two rgb.segmented_* images (1 and 2) you can run norm(as.matrix(rgb.segmented_1 - rgb.segmented_2)). Also, you can show the differences with inspect_segmentation(rgb, abs(rgb.segmented_1-rgb.segmented_2), rgb.proj)

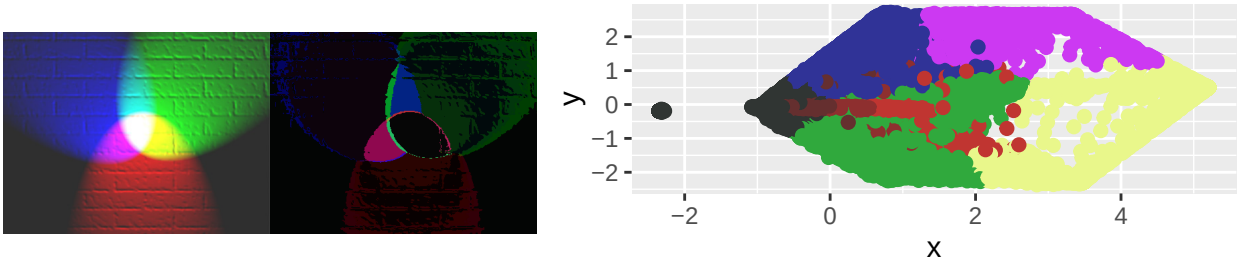
```
rgb.df = segment_image(rgb, 8)
rgb.proj = project2D_from_RGB(rgb.df)
rgb.segmented_1 = build_segmented_image(rgb.df, rgb)

rgb.df = segment_image(rgb, 8)
rgb.proj = project2D_from_RGB(rgb.df)
rgb.segmented_2 = build_segmented_image(rgb.df, rgb)

norm(as.matrix(rgb.segmented_1 - rgb.segmented_2))
```

```
## [1] 18035.76
```

```
norm(as.matrix(rgb - rgb.segmented_2))
```

```
## [1] 17827.05
```

```
inspect_segmentation(rgb, abs(rgb.segmented_1-rgb.segmented_2), rgb.proj)
```




As we can see result isn't always the same. It's happening because k-means stops balancing in local minimum, so with different starting positions (most of the algorithms using some randomness in calculating start centroid positions) of centroids we are obtaining different results.

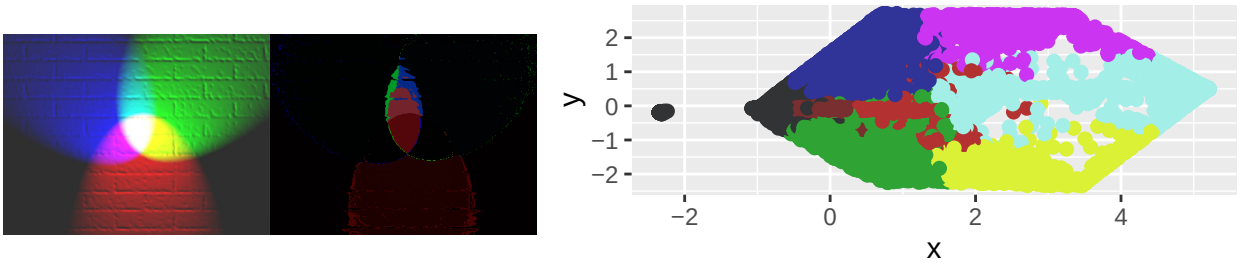
```
rgb.df = segment_image(rgb, 8, "MacQueen", 30)
rgb.proj = project2D_from_RGB(rgb.df)
rgb.segmented_1 = build_segmented_image(rgb.df, rgb)

rgb.df = segment_image(rgb, 8, "MacQueen", 30)
rgb.proj = project2D_from_RGB(rgb.df)
rgb.segmented_2 = build_segmented_image(rgb.df, rgb)

norm(as.matrix(rgb.segmented_1 - rgb.segmented_2))
```

```
## [1] 7658.372
```

```
norm(as.matrix(rgb - rgb.segmented_2))
```

```
## [1] 16208.51
```

```
inspect_segmentation(rgb, abs(rgb.segmented_1-rgb.segmented_2), rgb.proj)
```




As you can see, since MacQueen algorithm is different from Hartigan-Wong's, there are difference between reclusteried image comparision too.

The strange thing is that MacQueen less error between clusterized and original image versus Hartigan-Wong algorithm. But I guess it's because of the color characteristics of the image.

---

**Q4) We know that k-means needs to have K (the number of clusters) provided in advance. What is the meaning of k here? Try to run the previous code many times with different values of k and comment the results. Are they better? Are they more stable?**

In our case k is a number of colors in which we are segmenting our image.

```
mandrill.df = segment_image(mandrill, 2)
mandrill.proj = project2D_from_RGB(mandrill.df)
mandrill.segmented = build_segmented_image(mandrill.df, mandrill)
writePNG(mandrill.segmented, "mandrill_s_2.png" )

mandrill.df = segment_image(mandrill, 20)
```

```
## Warning: Quick-TRANSfer stage steps exceeded maximum (= 13107200)
```

```
mandrill.proj = project2D_from_RGB(mandrill.df)
mandrill.segmented = build_segmented_image(mandrill.df, mandrill)
writePNG(mandrill.segmented, "mandrill_s_20.png" )

mandrill.df = segment_image(mandrill, 100)
```

```
## Warning: Quick-TRANSfer stage steps exceeded maximum (= 13107200)
```

```
mandrill.proj = project2D_from_RGB(mandrill.df)
mandrill.segmented = build_segmented_image(mandrill.df, mandrill)
writePNG(mandrill.segmented, "mandrill_s_100.png" )

rgb.df = segment_image(rgb, 2)
rgb.proj = project2D_from_RGB(rgb.df)
rgb.segmented = build_segmented_image(rgb.df, rgb)
writePNG(rgb.segmented, "rgb_illum_s_2.png")
```

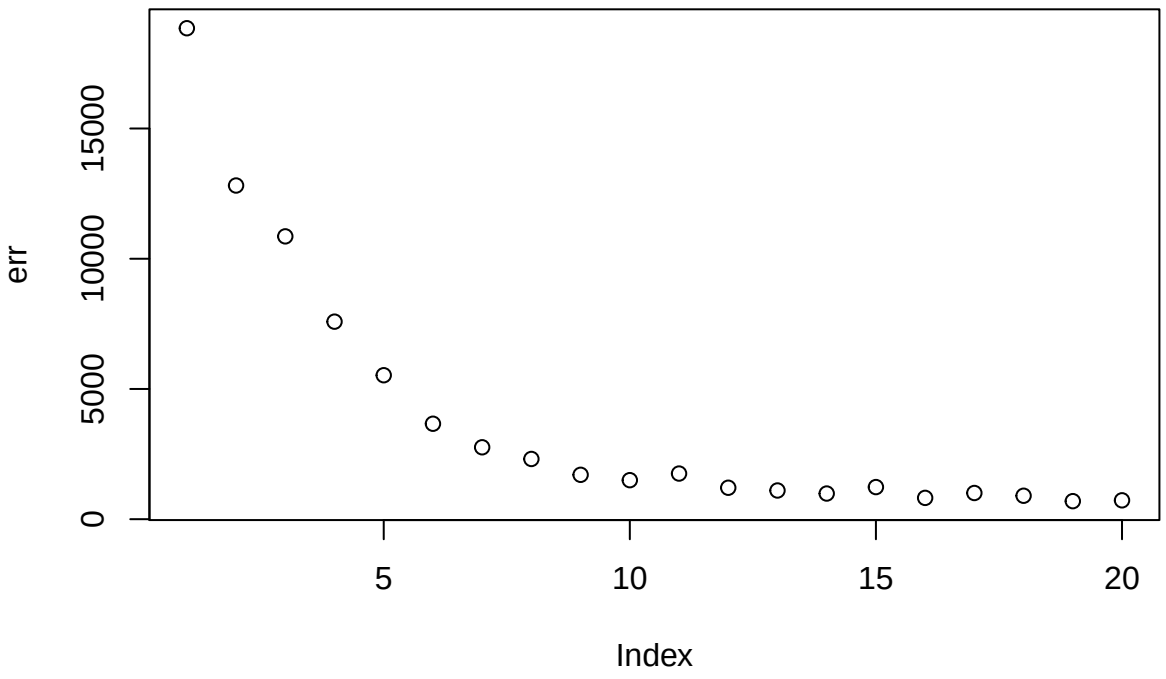
You can find result images in attached files. As you can see with increase of k we are increasing the number of colors. But can we say that mandrill_s_2.png is better than mandrill_s_100.png or vice versa? The main goal of Color Quantization is to decrease size of image file (for k=2 it's 39kb vs 626kb) with keeping an ability to understand what's pictured on our image. For mandrill k=2 is ok, but for rgb_illum_s_2.png it's not. That means that for each image we should pick k individually in case to keep underdating the image. So with increasing of k, size of file and stability(quality of color transitions) increasing too.

---

As you have seen, increasing K gives you a quantized image with a better quality, i.e. one which is closer to the original image.

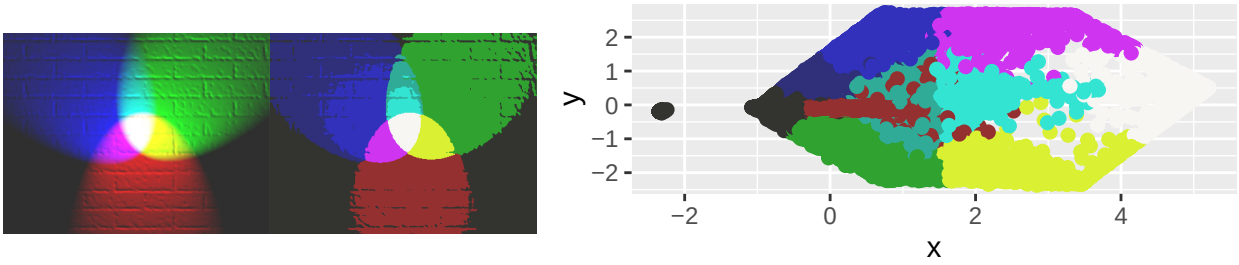
**Q5) Is there a way to find the "best K" for this application? As an example, try running the following code to calculate the SSE of the quantized image w.r.t. the original one for different values of K, plot them and comment the results (could you spot a K which is clearly better than others using the "elbow method"? Is the result qualitatively satisfying or do you need a much bigger K to have a good image?). Also think about alternatives to the elbow method if that is not sufficient (e.g. what if you choose the K that lowers the average error down to a given threshold?)**

```
image = rgb #just for the fast switch

err = 0
for(k in 1:20){
  image.segmented = build_segmented_image(segment_image(image, k, "Lloyd", 1000), image)
  err[k] =  sum((image-image.segmented)^2)
}
plot(err)
```



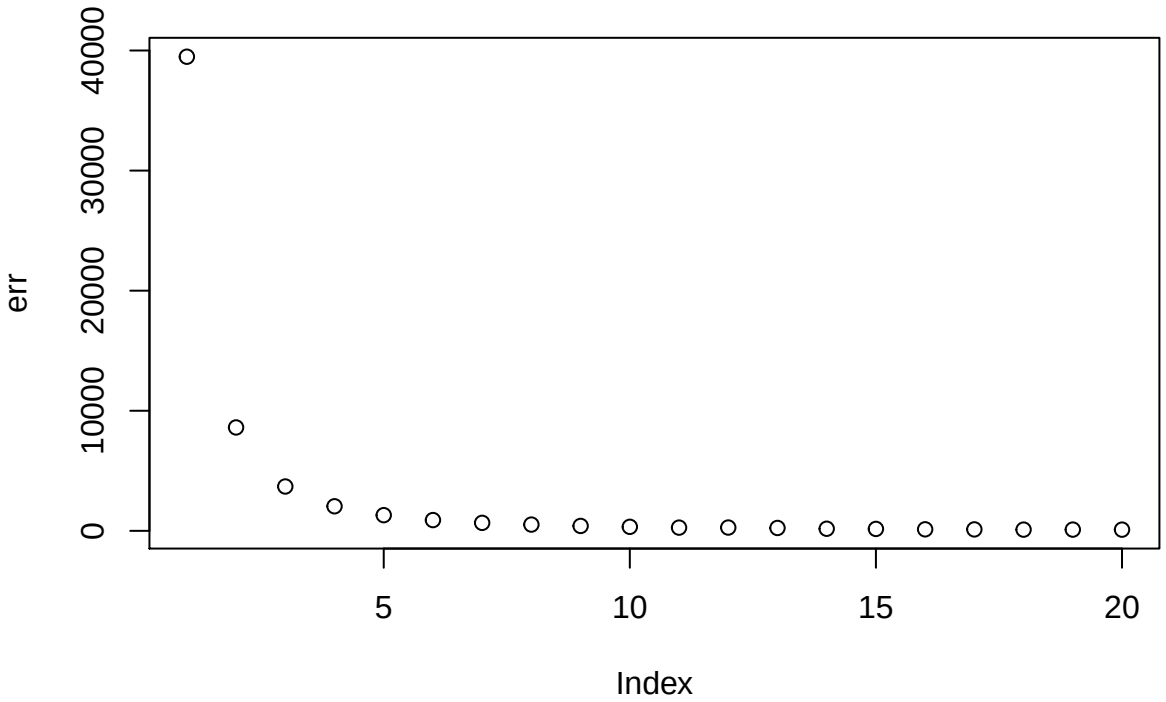

We can spot an elbow at k=10. Let's try it

```
best_k = 10
image.df = segment_image(image, best_k)
image.segmented = build_segmented_image(image.df, image)
image.proj = project2D_from_RGB(image.df)
inspect_segmentation(image, image.segmented, image.proj)
```



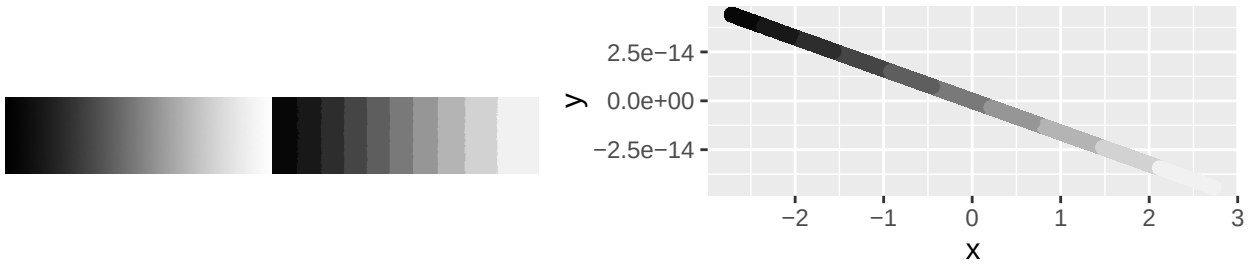

As you can see, we can distingush most of the color transitions. But is it ok for us? With the increasing of the k, the segmented image will be more stable, but we will still can spot a difference with the original one.

```
gradient <- readJPEG("gradient.jpg")
image = gradient #just for the fast switch

err = 0
for(k in 1:20){
  image.segmented = build_segmented_image(segment_image(image, k, "Lloyd", 1000), image)
  err[k] =  sum((image-image.segmented)^2)
}
plot(err)
```




According to the specific features of gradient image, k will be just n of lines, so our plot is very smooth. Can we choose the best k with 'elbow method'? I guess no. And more than that, other methods to define k will be unapplicable for our application.

```
best_k = 10
image.df = segment_image(image, best_k)
image.segmented = build_segmented_image(image.df, image)
image.proj = project2D_from_RGB(image.df)
inspect_segmentation(image, image.segmented, image.proj)
```




The only thing we can do is to select a threshold for SSE.

```
k <- 2
err = 2001
threshold = 2000
while(err > threshold){
  image.segmented <- build_segmented_image(segment_image(image, k, "Lloyd", 1000), image)
  err =  sum((image-image.segmented)^2)
  k <- k+1
}
print(k-1)
```

```
## [1] 5
```

```
inspect_segmentation(image, image.segmented, image.proj)
```