MIS 381N
Stochastic Control and Optimization: Project 4

GROUP 10 – Matthew Barrett, Tomas Fernandez, Jushira Thelakkat, Steven Zheng

NAÏVE THRESHOLDING:

We first conduct Naïve Thresholding to divide the image into two- foreground and background

The code for Naïve Thresholding is shown below; this can be done for any image as an input

```
library(igraph)
library(imager)
library(ggplot2)
library(dplyr)

image_seg_threshold <- function(inputImage) {
  pic = load.image(inputImage)
  pic.df = as.data.frame(pic)
  den = density(pic.df$value)

  #First and Second order edge detection
  firstDer = diff(den$y)/diff(den$x)
  secondDer = diff(firstDer)/diff(den$x[1:511])

  #Setting a minimum threshold of 0.5
  threshold = den$x[which(abs(firstDer) ==
                            min(abs(firstDer[intersect(which(secondDer > 0), which(den$y > 0.5))])))]

  pic.df$rgb.val = 0
  pic.df[pic.df$value > threshold,]$rgb.val = "#0000FF"
  pic.df[pic.df$value <= threshold,]$rgb.val = "#FF0000"
  p <- ggplot(pic.df,aes(x,y))+geom_raster(aes(fill=rgb.val))+scale_fill_identity()
  return (p+scale_y_reverse())
}
```

As given in the problem statement, there are few issues with thresholding;

Thresholding is too naïve to work on complicated images, especially because there is no incentive built to mark pixels closer to one another with the same background or foreground label. We thus jump to Min-Cut/Max Flow Optimization method.

MIN CUT/MAX FLOW OPTIMIZATION:

In this process, after inputting the image, we created boundaries on both x and y.

The below formulae have been given in the problem statement:

$$a_i = -log\frac{abs(p_i - \overline{p_f}\ )}{abs(p_i - \overline{p_f}\ ) + abs(p_i - \overline{p_b}\ )}$$

$$b_i = -log\frac{abs(p_i - \overline{p_b}\ )}{abs(p_i - \overline{p_f}\ ) + abs(p_i - \overline{p_b}\ )}$$

And the weight for an edge connecting pixels i and j is given by c_{ij},

$$c_{ij} = K\ \exp(\frac{-(p_i - p_j)^2}{\sigma^2})$$

So substituting K = 0.01 (also suggested) and σ=1, we calculated the weights of edges connecting pixels and iterated this using a loop for each row in the image. We used the resulting graph as input to the max flow solver.

The code for the Min-Cut/Max-Flow Optimization process is shown below:

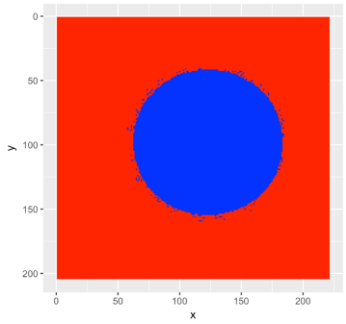
```
image_maxflow <- function(inputImage) {
  pic = load.image(inputImage)

  pic.row = nrow(pic)
  pic.col = ncol(pic)
  pic.df = as.data.frame(pic)
  pic.n = nrow(pic.df)

  #Creating boundaries
  bound.x = c(round(pic.col/4),round(pic.col/4)*3)
  bound.y = c(round(pic.row/4),round(pic.row/4)*3)

  pic.df = pic.df %>%
    mutate(foreground = as.integer((x>bound.x[1] & x<bound.x[2]) & (y>bound.y[1] & y<bound.y[2])))

  #Using mean pixels to use to calculate a and b
  mean_pixel.fore = mean(filter(pic.df, foreground == 1)$value)
  mean_pixel.back = mean(filter(pic.df, foreground == 0)$value)

  pic.df$a = -log10(abs(pic.df$value - mean_pixel.fore)/(abs(pic.df$value - mean_pixel.fore)+abs(pic.df$value - mean_pixel.back)))
  pic.df$b = -log10(abs(pic.df$value - mean_pixel.back)/(abs(pic.df$value - mean_pixel.fore)+abs(pic.df$value - mean_pixel.back)))

  K = 0.01
  sigma = 1
```

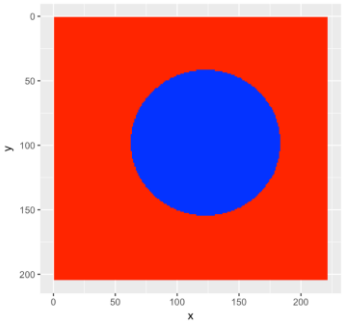
```
  from = c()
  to = c()
  weight = c()

  for (i in 1:pic.n){
    f = c()
    t = c()
    w = c()
    if (i - 1 > 0){
      j = i - 1
      f = c(f, i)
      t = c(t, j)
      w = c(w, K*exp(-(pic.df$value[i] - pic.df$value[j])^2/sigma^2))
    }
    if (i + 1 <= pic.n){
      j = i + 1
      f = c(f, i)
      t = c(t, j)
      w = c(w, K*exp(-(pic.df$value[i] - pic.df$value[j])^2/sigma^2))
    }
    if (i - pic.row > 0){
      j = i - pic.row
      f = c(f, i)
      t = c(t, j)
      w = c(w, K*exp(-(pic.df$value[i] - pic.df$value[j])^2/sigma^2))
    }
    if (i + pic.row <= pic.n){
      j = i + pic.row
      f = c(f, i)
      t = c(t, j)
      w = c(w, K*exp(-(pic.df$value[i] - pic.df$value[j])^2/sigma^2))
    }
    from = c(from, f)
    to = c(to, t)
    weight = c(weight, w)
  }

  from = c(from, rep("s",pic.n), c(1:pic.n))
  to = c(to, c(1:pic.n), rep("t",pic.n))
  weight = c(weight, pic.df$a, pic.df$b)
  link = as.data.frame(cbind(from, to, weight))
```

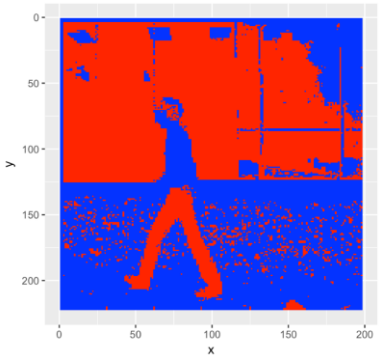
```
  g = graph_from_data_frame(d = link)

  solution = max_flow(g, source = 's', target = 't', capacity = weight)

  pic.fore = as.vector(solution$partition1[solution$partition1 < pic.n + 1])
  pic.back = as.vector(solution$partition2[solution$partition2 < pic.n + 1])

  pic.df$rgb.val = 0
  pic.df[pic.fg,]$rgb.val = '#0000FF'
  pic.df[pic.bg,]$rgb.val = '#FF0000'

  #Plotted solution and then reverse it to get the segmentation output of the original image
  p <- ggplot(pic.df,aes(x,y))+geom_raster(aes(fill=rgb.val))+scale_fill_identity()
  return (p+scale_y_reverse())
}
```

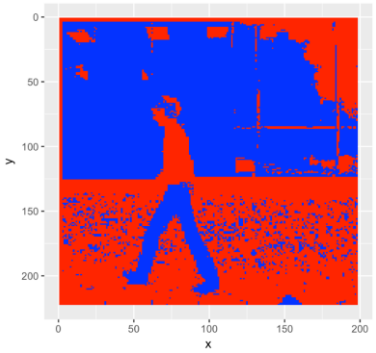
The outputs of these above functions for the different images are given below:

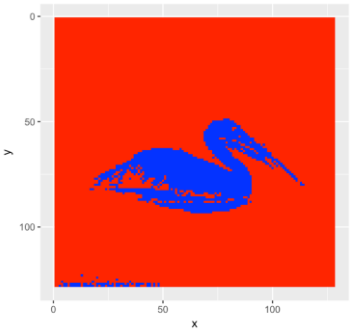
The first stage being the Naïve Thresholding o/p and the final being the Max Flow o/p. We observe that Max Flow does a significantly much better job than the simple Naïve Thresholding. As the image gets more complicated, we see that the difference in both these methods becomes increasingly visible.

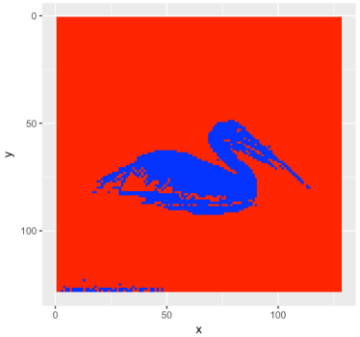

y
x
0
50
100
150
200

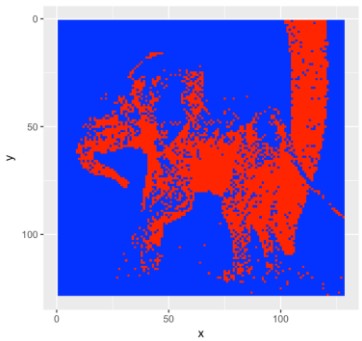

y
x
0
50
100
150
200

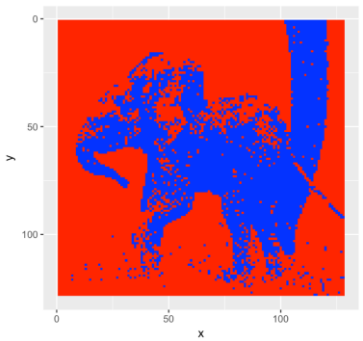

y
x
0
50
100
150
200


y
x
0
50
100
150
200


y
x
0
50
100


y
x
0
50
100


y
x
0
50
100


y
x
0
50
100