Image Contour Extraction

COMP3032 - 2010

Joshua England - je7g08@soton.ac.uk

Abstract

This short report explores how a Dynamic Programming method can be used for finding the optimum contour across an image. This is part of the COMP3032 coursework assignment.

1. Introduction

The task given [1] was to use a Dynamic Programming[2] method to find the optimum contour across an given search space of a grey-scale image defined by two other contours. The optimum contour was defined by the formulas below.

$$S_i(v_{i+1}, v_i) = \min_{v_i - 1}[S_{i-1}(v_i, v_{v-1}) + E_i(v_{i+1}, v_i, v_{i-1})] \tag{1}$$

$$E_i(v_{i+1}, v_i, v_{i-1}) = \lambda \frac{|v_{i+1} - 2\mathrm{v}_i + v_{i-1}|^2}{|v_{i+1} - v_{i-1}|^2} + (1-\lambda) I(v_i) \tag{2}$$

Where v_i represents a pixel on the image, $\lambda \in [0,1]$ and $I(v_i)$ is the intensity at that point.

2. Method

Initially MATLAB/Octave was used to develop a heuristic solution, however, due to problems displaying images with Octave a change was made to use Python[1] and the PIL library[2] [3]. This also allowed for easier image manipulation and conversion between different image formats (mainly RGB to grey-scale).

The search space was extracted using Bresenham's line drawing algorithm [4] to retrieve a list of pixels as if a line was being drawn between corresponding points on each contour. This list of pixels was then divided into M segments to get the real search space with some points being repeated if the line was not long enough. Tables (python deep list) were then formed containing pixel intensities and locations respectively.

An extra deep list was generated to represent the results of applying formula's (1) and (2). The format of the deep list was as follows:

- A list for each pair of points on the contours $(k \in [1, N-2])$
    - A list for each pixel in the line joining the contours in the search space $(j \in [0, M-1])$
        - A list for each pixel in the next column $(i \in [0, M-1])$

1 Developed and tested on Python 2.6.6 on Ubuntu (linux) 10.10
2 Developed and tested using Python Image Library (PIL) version 1.1.7 compiled from source for Ubuntu (linux) 10.10 64bit edition and Python 2.6.6

- Minimal energy

The *minimal energy* corresponds to formula (1) taking the minimum of moving from any pixel in the previous pair of points $((k-1),x)$ to this pixel (k,j) and onto the given pixel in the next column $((k+1),i)$.

The contour was then found by examining the last column of the table to find the minimal cost of moving from any pixel in column $N-2$ to column $N-1$. Then the minimal of reaching the minimal pixel from column $N-3$ is found. This is repeated for all but the last two columns. The python code for the process is shown in Text 3.

```
def gen_contour():
   min = (0, 0)
   eng = ENGMAX
   for i in range (0, M):
     for j in range(0, M):
       if Scores[-1][i][j] < eng:
         eng = Scores[-1][i][j]
         min = (i, j)
   Contour.insert(0, PixelLocations[-1][min[1]])
   Contour.insert(0, PixelLocations[-2][min[0]])
   oldmin = min[0]
   for k in range(2, N-2):
     eng = ENGMAX
     min = 0
     for i in range(0, M):
       if Scores[-k][i][oldmin] < eng:
         eng = Scores[-k][i][oldmin]
         min = i
     Contour.insert(0, PixelLocations[-(k+1)][min])
     oldmin = min
```

Text 3: Python code showing the generation of the contour

Using the PIL library [3] an output image is formed showing all the pixels in the search space and a line connecting those which form the minimal contour. This is then shown in a new window.

3. Results

Illustration 1, 2 and 3 show the output of the script using different values for *M* using an image of a tongue given in the assignment.

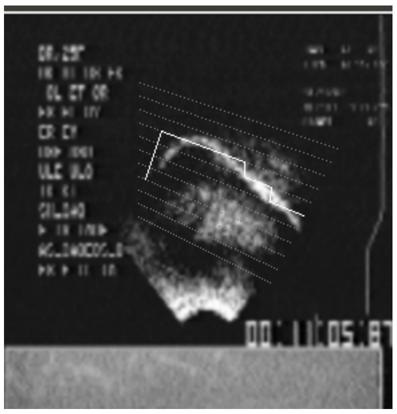


Illustration 2: Contour with M=10, λ = 0.5

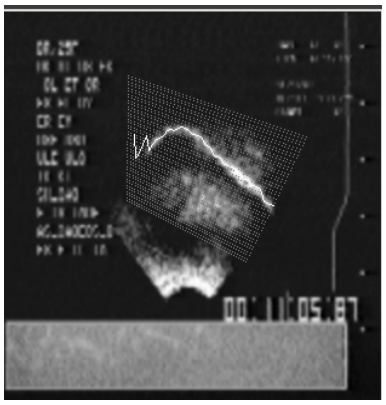


Illustration 3: Contour with M=40, λ = 0.5

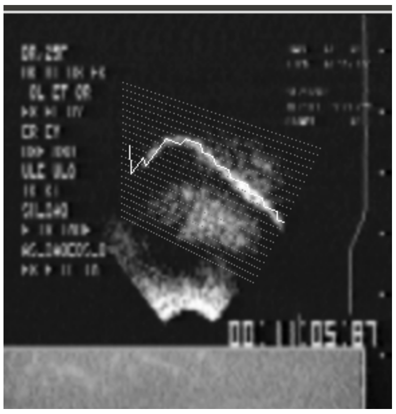


Illustration 1: Contour with M=20 λ = 0.5

As can be seen, increasing the value of *M* makes gives better resolution to the contour but with a dramatically increased time complexity. More screen shots of the output can be found in Appendix A or in the zip file accompanying this document.

4. Scaling

Unfortunately this algorithm does not scale very well. This is due to the number of comparisons needed for form the deep list of scores (4) and also for the backtrack (5).

$$\theta((N-2)\times M^3) \tag{4}$$

$$\theta(M^2+(N-4)\times M) \tag{5}$$

A solution to this problem could be to use Parallel Computing by performing calculations on a GPU rather than on a CPU using an interface like Nvidia CUDA [5]. This would help to flatten some of the nested for-loops in the program but care would have to be taken for synchronisation.

Nevertheless this is a better time complexity than brute-force search which examines every possible contour (6) while a greedy or heuristic method would require extra development to make sure that it was correct.

$$\theta(M^N) \tag{6}$$

5. Regularisation Parameter

Variances in the regularisation parameter λ force the algorithm to take more or less account for the pixel intensities. As can be seen from (2) when λ approaches 1 the pixel intensity becomes less important so a straighter line across the image is found to be optimal. But when λ is 0 the curve connecting the 3 pixels being examined is still taken into account. Screenshots from the program with different values for λ can be seen in Illustration 4.

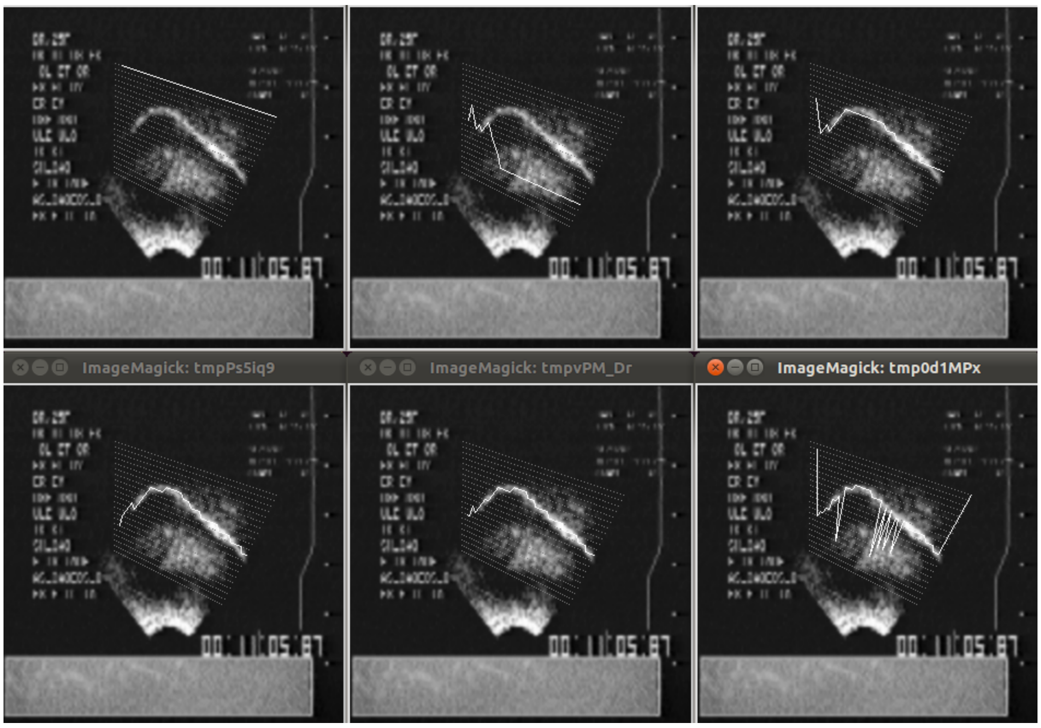


Illustration 4: Screenshots showing the output for different lambda values. From top left to bottom right lambda is 1, 0.8, 0.6, 0.4, 0.2 and 0.

6. Different Images

Tests were also ran which used a different input image which can be seen in Illustration 5 and Illustration 6. These use a black line to show the contour and red dots for the points examined.

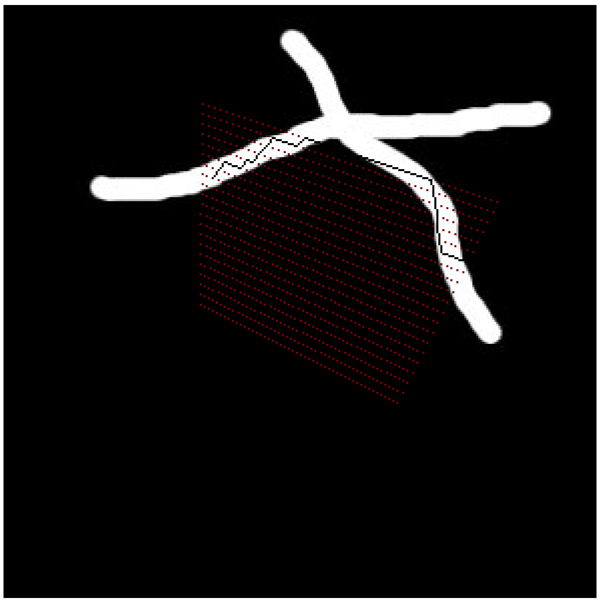


Illustration 5: Output on a simple image

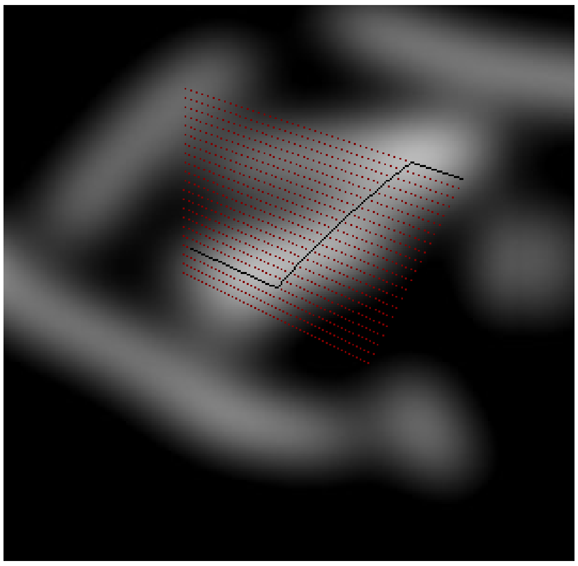


Illustration 6: Output on a blurred image

There were also tests using different contour files, a screen shot of one can be seen in Illustration 7.

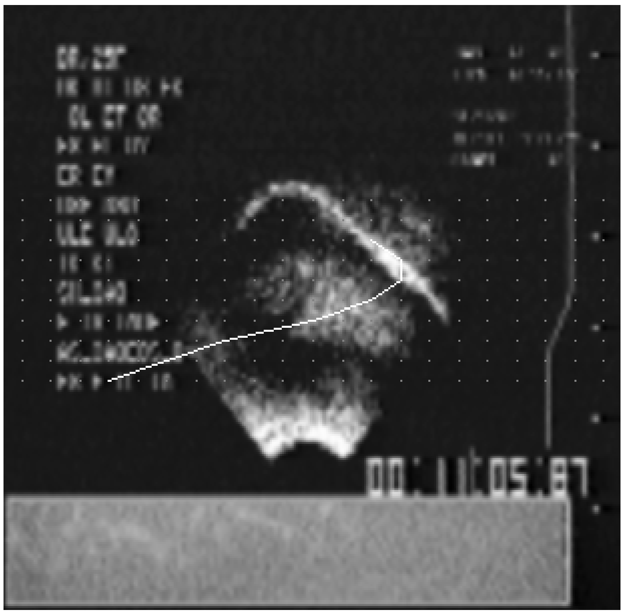


Illustration 7: Algorithm on the image of a tongue with different contour files bordering the search space

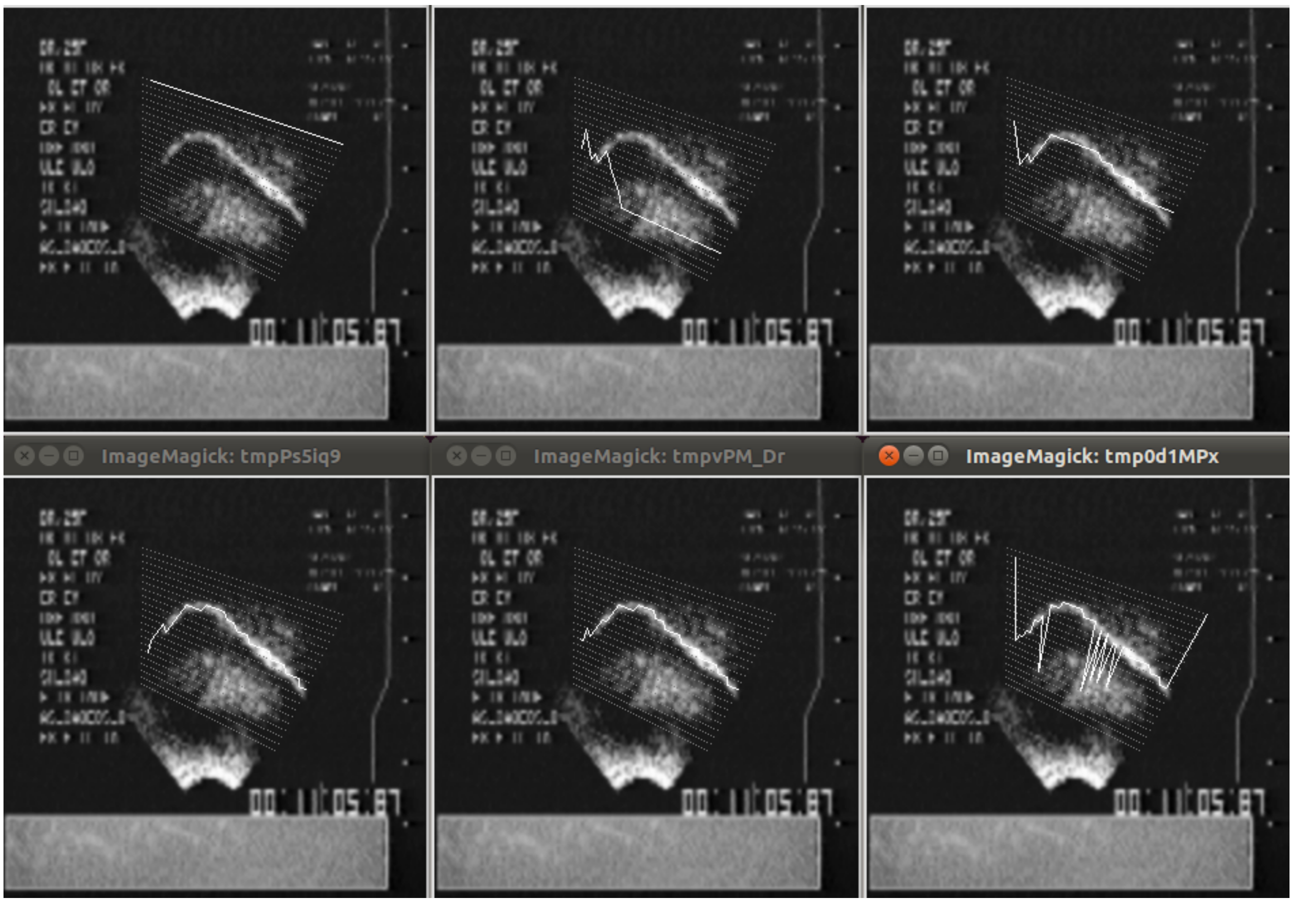
7. Conclusion

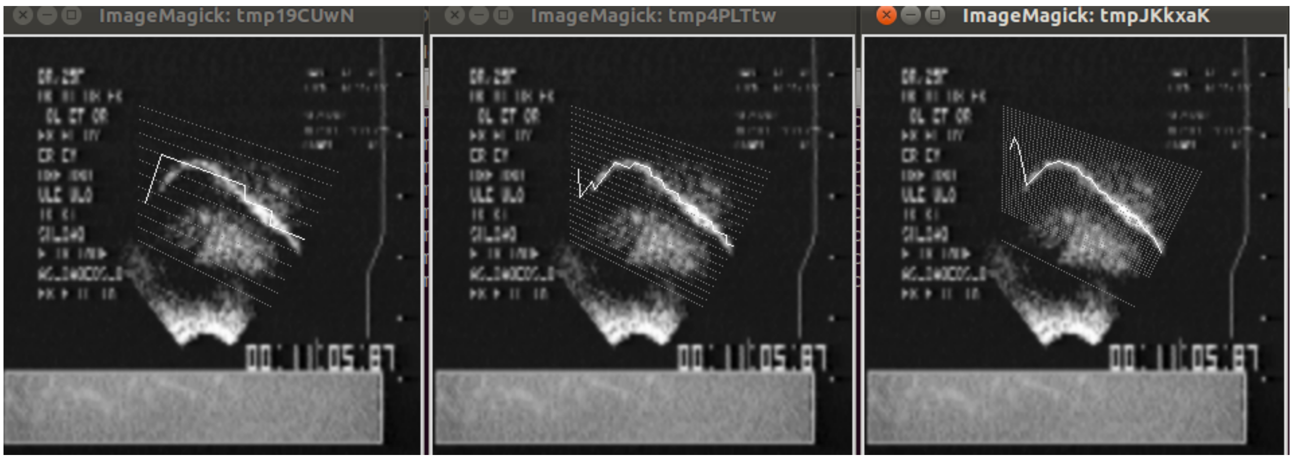
The algorithm and code developed does find the contours on images effectively and, with some more development, could easily become more robust. Despite being $\theta(M^3)$ algorithm it is far better than brute-force and the parallel paradigm could offer significant execution improvements.

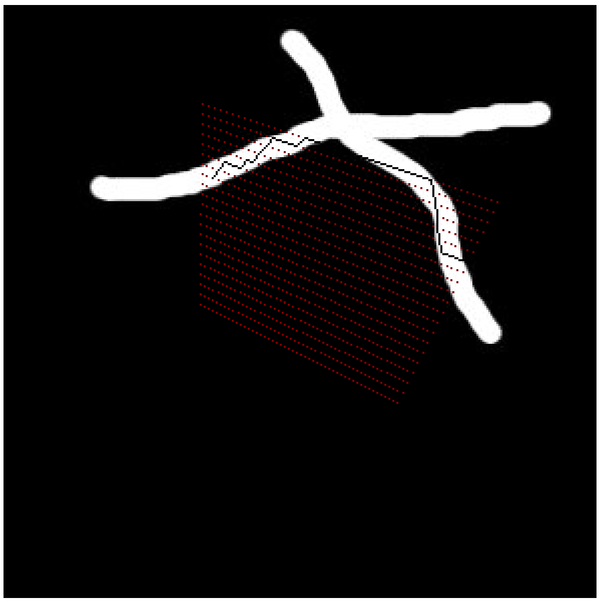
References

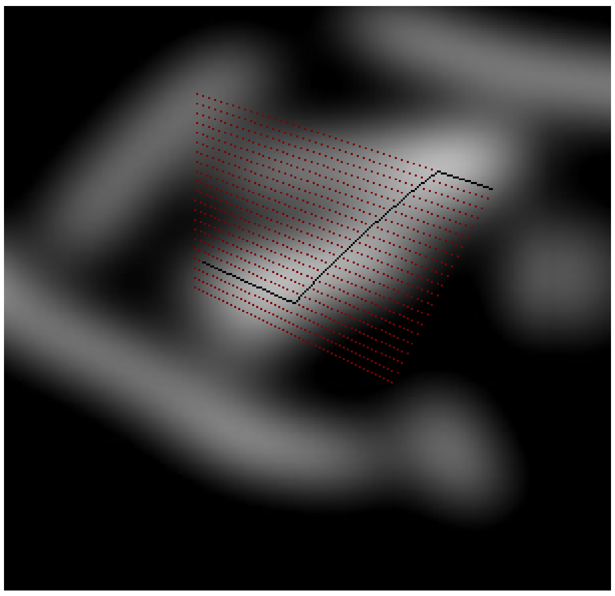
[1] Steven Gunn, COMP3032 Assignment: Image Contour Extraction, 2010, https://secure.ecs.soton.ac.uk/notes/comp3032/Assignment/ICE.pdf
[2] Michael A. Trick, A Tutorial on Dynamic Programming, 1997, http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html
[3] Anon., Python Imaging Library (PIL), 2010, http://www.pythonware.com/products/pil/
[4] Tom Ootjers, GameDev.net - Line Drawing Algorithm Explained, 2010, http://www.gamedev.net/reference/articles/article1275.asp
[5] Anon, What is CUDA?, 2010, http://www.nvidia.com/object/what_is_cuda_new.html

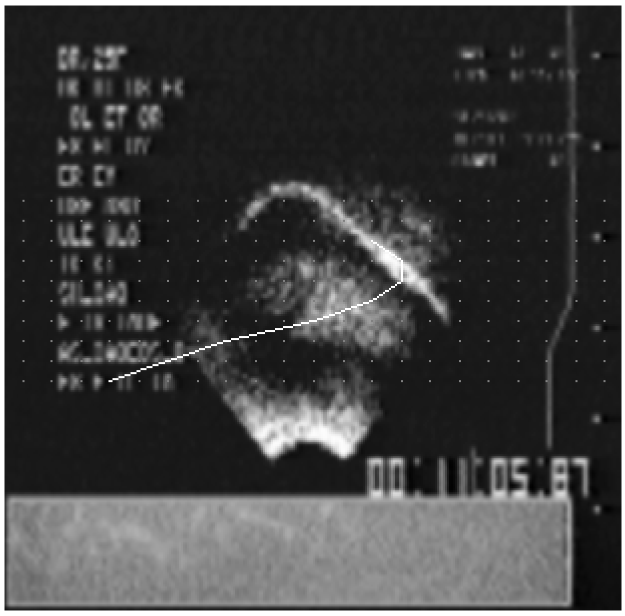
Appendix A : Screenshots











Appendix B : Source Code

```
# Curved Optimal Contour Finder
# Comp3032 Coursework
# By Joshua England (je7g08@soton.ac.uk)

# Developed and tested with Python 2.6.6 and Ubuntu Linux 10.10
# Required Python Image Library (PIL) to work
# Developed and tested with PIL 1.1.7 built from source
# See http://www.pythonware.com/products/pil/ for details

import Image
import ImageDraw
from optparse import OptionParser

# constants
IMAGE_PATH = "tongue.png"
CONTOUR1_PATH = "init1.ctr"
CONTOUR2_PATH = "init2.ctr"
M = 20
N = 0
LAMBDA = 0.5
ENGMAX = 5000

# globals
ImageObj = None
OutputImageObj = None
Contour1 = []
Contour2 = []
PixelLocations = []
IntensityValues = []
Scores = []
Contour = []

def load_image():
  """
  Loads the image specified
  """
  global ImageObj
  global OutputImageObj
  ImageObj = Image.open(IMAGE_PATH)
  OutputImageObj = ImageObj.copy()
  ImageObj = ImageObj.convert("L")

def load_contours():
  """
  Reads the contour files specified
  constructing two lists of vertex's
  """
  global N
  cfile1 = open(CONTOUR1_PATH, "r")
  for line in cfile1:
    l = line.strip().split(" ")
    x = int(float(l[0]))
    y = int(float(l[2]))
    Contour1.append((x, y))
  cfile2 = open(CONTOUR2_PATH, "r")
  for line in cfile2:
    l = line.strip().split(" ")
    x = int(float(l[0]))
    y = int(float(l[2]))
    Contour2.append((x, y))
  if len(Contour2) != len(Contour1):
    print """Sorry but the contour files you provided do not have the
same
    number of points in each"""
    exit()
  N = len(Contour2)

def pixels_between_points(v1, v2):
  """ Returns a list of interpolated points as if
  it was drawing a list on the screen between the two points

  Based on code from John Carter, lecturer for comp3004 Graphics
  """
  list = []
  dx = v2[0] - v1[0]
  dy = v2[1] - v1[1]
  incrN = 0
  incrEN = 0
  incrE = 0
  incrNE = 0
  y_end = 0
  x_end = 0
  Xinc = 0
  Yinc = 0
  d = 0
  if abs(dx) > abs(dy):
    if dx < 0:
      tmp = v1
      v2 = v1
      v1 = tmp
    if v2[1]>v1[1]:
      YInc = 1
    else:
      YInc = -1
    dx = abs(dx)
    dy = abs(dy)
    d = 2*dy -dx
    incrE = 2*dy
    incrEN = 2*(dy-dx)
    x_end = v2[0]
    x = v1[0]
    y = v1[1]
    list.append((x, y))
    while x < x_end:
      if d <= 0:
        d = d + incrE
        x = x + 1
      else:
        d = d + incrNE
        x = x + 1
        y = y + Yinc
      list.append((x, y))
  else:
    if dy < 0:
      tmp = v2
      v2 = v1
      v1 = tmp
    if v2[0] > v2[0]:
      Xinc = 1
    else:
      Xinc = -1
    dx = abs(dx)
    dy = abs(dy)
    d = 2*dx-dy
    incrN = 2*dx
    incrEN = 2*(dx-dy)
    y_end = v2[1]
    x = v1[0]
    y = v1[1]
    list.append((x,y))
    while y < y_end:
      if d <= 0:
        d = d + incrN
        y = y + 1
      else:
        d = d + incrEN
        y = y + 1
        x = x + Xinc
      list.append((x,y))
  return list
```

```

def load_pixellocations():
  """
  For each pair of contour points, constructs a line
  between them and divides into M segments
  """
  for i in range(len(Contour1)):
    list = []
    pixels_between = pixels_between_points(Contour1[i],
Contour2[i])
    division = float(len(pixels_between))/float(M)
    list.append(pixels_between[0])
    for i in range(1, M):
      list.append(pixels_between[int(i*division)])
    list.append(pixels_between[-1])
    PixelLocations.append(list)

def load_intensityvalues():
  """ Generates a value between 0 and 1 for each
  pixel on the image """
  for row in PixelLocations:
    list = []
    for pixel in row:
      list.append(1- float(ImageObj.getpixel(pixel))/256)
    IntensityValues.append(list)

def magnitude(loc1, loc2):
  loc1 = PixelLocations[loc1[0]][loc1[1]]
  loc2 = PixelLocations[loc2[0]][loc2[1]]
  return (loc2[0] - loc1[0])**2 + (loc2[1] - loc1[1])**2

def straightness(loc1, loc2, loc3):
  loc1 = PixelLocations[loc1[0]][loc1[1]]
  loc2 = PixelLocations[loc2[0]][loc2[1]]
  loc3 = PixelLocations[loc3[0]][loc3[1]]
  return (loc3[0] - 2*loc2[0] + loc1[0])**2 + (loc3[1] - 2*loc2[1] +
loc1[1])**2

def get_energy(loc1, loc2, loc3):
  """ Energy cost of moving from loc1 through loc2 to loc3"""
  return LAMBDA*straightness(loc1, loc2, loc3)/magnitude(loc1,
loc3) + (1-LAMBDA)*IntensityValues[loc2[0]][loc2[1]]

def get_score(loc1, loc2):
  """
  Get the score at loc2 having moved from loc1
  """
  if loc1[0] == 0 and loc2[0] == 1:
   return 0
  a = Scores[loc1[0] - 1]
  b = a[loc1[1]]
  c = b[loc2[1]]
  return c

def min_movement(pixel, nextpixel):
  """
  Calculates the minimal cost of moving from one pixel to an
adjacent pixel
  """
  min = 0
  eng = ENGMAX
  k = pixel[0]
  h = pixel[1]
  j = nextpixel[1]
  for i in range(0, M):
    e = get_score((k-1, i), (k, h)) + get_energy((k-1, i), (k, h), (k+1, j))
    if e < eng:
      min = i
      eng = e
  return (eng)

def gen_nextcollist(pixel):
  """
  Returns a list of minimal energies for reaching every pixel in the
next column
  from a given pixel in the search space
  """
  list = []
  k = pixel[0]
  for i in range(0, M):
    list.append(min_movement(pixel, (k+1, i)))
  return list

def gen_column(col):
  """
  Returns a (deep) list of scores for a column of the search space
  """
  list = []
  for i in range(0, M):
    list.append(gen_nextcollist((col, i)))
  return list

def gen_scores():
  """
  Applies the algorithm to generate a table (deep list) of scores
  """
  for k in range(1, N-1):
    Scores.append(gen_column(k))

def gen_contour():
  """
  Generates the path for the contour by backtracking
  through the scores table
  """
  min = (0, 0)
  eng = ENGMAX
  for i in range (0, M):
    for j in range(0, M):
      if Scores[-1][i][j] < eng:
        eng = Scores[-1][i][j]
        min = (i, j)
  Contour.insert(0, PixelLocations[-1][min[1]])
  Contour.insert(0, PixelLocations[-2][min[0]])
  oldmin = min[0]
  for k in range(2, N-2):
    eng = ENGMAX
    min = 0
    for i in range(0, M):
      if Scores[-k][i][oldmin] < eng:
        eng = Scores[-k][i][oldmin]
        min = i
    Contour.insert(0, PixelLocations[-(k+1)][min])
    oldmin = min

def show_contour():
  """
  Shows the pixels evaluated and the contour in a new window
  """
  draw = ImageDraw.Draw(OutputImageObj)
  for list in PixelLocations:
    for pixel in list:
      i = pixel[0]
      j = pixel[1]
      draw.point((i,j), fill=128)
  print "Optimal Contour:"
  for i in range(1, len(Contour)):
    draw.line(Contour[i-1] + Contour[i], fill=256)
    print """  %d  %d  """ % (Contour[i-1][0], Contour[i-1][1])
  OutputImageObj.show()

def commandline():
  """
  Process options from the command line and set global variables if
needed
```

```
  """
  parser = OptionParser()
  parser.add_option("--lambda", dest="lambdaval", default=0.5,
    help="regularisation parameter", metavar="FLOAT")
  parser.add_option("--image", dest="image_path",
default="tongue.png",
    help="path to the image file being use")
  parser.add_option("--contour1", dest="cont1", default="init1.ctr",
    help="path to first contour")
  parser.add_option("--contour2", dest="cont2", default="init2.ctr",
    help="path to second contour")
  parser.add_option("--divisions", dest="m", metavar="INT",
default=10,
    help="number of subdivisions")
  (options, args) = parser.parse_args()
  global LAMBDA, M, IMAGE_PATH, CONTOUR1_PATH,
CONTOUR2_PATH
  LAMBDA = float(options.lambdaval)
  M = int(options.m)
  IMAGE_PATH = options.image_path
  CONTOUR1_PATH = options.cont1
  CONTOUR2_PATH = options.cont2

def init():
  """
  Parses input flags and loads data needed
  """
  commandline()
  load_image()
  load_contours()
  load_pixellocations()
  load_intensityvalues()

def main():
  """
  Main method for the program like c/java
  """
  init()
  gen_scores()
  gen_contour()
  show_contour()

# Trick to make python scripts act like C
if __name__ == "__main__":
  main()
```