Appendix A: Fingerprint Preprocessing Function

```
import struct

import cStringIO

import Image


# Import C functions necessary for creating and destroying large arrays

cdef extern from "stdlib.h":

    void *malloc(int size)

    void free(void *ptr)


def transformOutline(imageTuple, variableList):

    # Define the necessary C variables

    cdef short int height

    cdef short int width

    cdef short int edgeLength

    cdef short int xLower

    cdef short int xUpper

    cdef short int yLower

    cdef short int yUpper

    cdef short int iLoop

    cdef short int iLoop2

    cdef short int iLoop3
```

```
cdef short int iLoop4

cdef int areaSum

cdef int index

cdef short int pixel

cdef short int *image

cdef short int *string

cdef float averagePixel

cdef float variance

cdef float varianceLimit

cdef short int whiteLimite

cdef short int blackLimit

cdef int leftSum

cdef int rightSum


# Store the python outline variables as C variables

width = variableList[0]

height = variableList[1]

edgeLength = variableList[2]

varianceLimit = variableList[3]

blackLimit = variableList[4]

whiteLimit = variableList[5]


# Use malloc to dynamically create potentially large c arrays

image = malloc(2*height*width)
```

```
string = malloc(2*height*width)


if(image == NULL or string == NULL):
    # Malloc was unsuccessful, we can not continue
    return -101


yLower = edgeLength/2
yUpper = height - yLower - 1
xLower = yLower
xUpper = width - xLower - 1


# Transform python image tuple into a C array
for iLoop from 0 <= iLoop < height:
    for iLoop2 from 0 <= iLoop2 < width:
        index = iLoop * width + iLoop2
        image[index] = imageTuple[index]


# Perform the outlining transformation
for iLoop from 0 <= iLoop < height:
    for iLoop2 from 0 <= iLoop2 < width:
        # Check to see if we are in bounds
        if(iLoop >= yLower and iLoop <= yUpper and iLoop2 >= xLower and iLoop2 <= xUpper):
            # We are in bounds, we can continue
            # Initialize the sum of our local area and the calculated variance
```

```
areaSum = 0
variance = 0
for iLoop3 from 0 <= iLoop3 < edgeLength:
    for iLoop4 from 0 <= iLoop4 < edgeLength:
        # Calculate the index
        index = (iLoop + iLoop3 - edgeLength/2) * width
        index = index + iLoop2 + iLoop4 - edgeLength/2
        areaSum = areaSum + image[index]
# We have the sum over the area, no we can determine the average
averagePixel = areaSum/(edgeLength * edgeLength)
# Now that we have the average, we can caluclate the variance
for iLoop3 from 0 <= iLoop3 < edgeLength:
    for iLoop4 from 0 <= iLoop4 < edgeLength:
        # Calculate the index
        index = (iLoop + iLoop3 - edgeLength/2) * width
        index = index + iLoop2 + iLoop4 - edgeLength/2
        variance = variance + (image[index] - averagePixel) * (image[index] - averagePixel)
# Perform final variance calculation
variance = variance / (edgeLength * edgeLength)
# Determine if pixel should be masked or unmasked bade on variance
index = iLoop * width + iLoop2
if(variance < varianceLimit):
    # There's not a lot of variance, we should mask this pixel
    string[index] = -1
```

```
            else:
                # There's variance here, probably not a smudge or open area
                # Now we must determine if the pixel is white or black
                if(averagePixel > whiteLimit):
                    # The pixel is easily classified as white
                    string[index] = 1
                elif(averagePixel < blackLimit):
                    # The pixel is easily classified as black
                    string[index] = 0
                else:
                    # We must do more work to determine if the pixel is black or white
                    if (image[index] > averagePixel):
                        # The pixel is lighter than it's sorroundings, call it white
                        string[index] = 1
                    else:
                        # The pixel must be black
                        string[index] = 0
        else:
            # We are not within the x and y boundaries, make the pixel green
            index = iLoop * width + iLoop2
            string[index] = -1

# Use a series of loops to transform the C string array into a python string
out = cStringIO.StringIO()
```

```
for iLoop from 0 <= iLoop < height:
    for iLoop2 from 0 <= iLoop2 < width:
        index = iLoop * width + iLoop2
        if(string[index] == -1):
            # Our point should be masked, make it green
            bin_str = struct.pack("BBB", 0, 255, 0)
            out.write(bin_str)
        elif(string[index] == 0):
            # Our point is black
            bin_str = struct.pack("BBB", 0, 0, 0)
            out.write(bin_str)
        else:
            # Our point is white
            bin_str = struct.pack("BBB", 255, 255, 255)
            out.write(bin_str)

# Don't forget to free the memory associated with the malloc'd arrays
free(image)
free(string)

# Return an image variable
tempTuple = (width, height)
return Image.fromstring("RGB", tempTuple, out.getvalue())
```

Appendix B: Scale-Spectra Generating Function

```
# Import the C functions we will need during our random walk
cdef extern from "math.h":
    double cos(double theta)
    double sin(double theta)
    double acos(double negOneToOne)
    double sqrt(double number)
    double pow(double base, double exp)
    float floor(float decimal)

# Import C functions necessary for creating and destroying large arrays
cdef extern from "stdlib.h":
    void *malloc(int size)
    void free(void *ptr)

def gridWalk(path, name, variableList):

    # Import the necessary python extensions
    import random

    # Define the necessary C variables
    cdef short int iHeight
    cdef short int iWdth
```

```
cdef float fScale
cdef short int iEdgeLength
cdef int iIterations
cdef int iRow
cdef int iColumn
cdef int iNetRow
cdef int iNetColumn
cdef short int iIsGreen
cdef short int iIsOB
cdef int iCounter
cdef int iCounter2
cdef float fHyp
cdef float fX
cdef float fY
cdef float fXStar
cdef float fYStar
cdef int iXOffset
cdef int iYOffset
cdef float fThetaRandom
cdef float fThetaRaw
cdef float fThetaAdjusted
cdef short in iIndex
cdef short int iPow
cdef short int iX
```

```
cdef short int iY
cdef char *image
cdef unsigned char *net

# Use malloc to dynamically create two potentially large c arrays
# The first array represents the image file
# The second array represents the walk data at a given scale
image = malloc(2 * iHeight * iWidth)
net = malloc(2 * iEdgeLength * iEdgeLength)

if(image == NULL):
   # Malloc was unsuccessful, we can not continue
   print "There was not enough memory to create the image array"
   return -101

if(net == NULL):
   # Malloc was unsuccessful, we can not continue
   print "There was not enough memory to create the net array"
   return -101

if(iIterations % 8 != 0):
   # We can not properly compress data unless iIterations % 8 is 0
   print "The number of iterations must be exactly divisible by 8"
   return -102
```

```
# Open a text file to store the results of the walk

fout = open("C:\\netWalkScale22.txt", "w")


# Determine and store information related to the image

self.im = Image.open(path + name)

iWidth = self.im.size[0]

iHeight = self.im.size[1]

# The following code places the images BnW values into a C array

orig_pixels = self.im.getdata()


pixelList = []

for i in range(0, len(orig_pixels)):

   pixelList.append(orig_pixels[i])

imageTuple = tuple(pixelList)


#      We now transfer the Python list representation of the image into

# a C array of characters where:

# A value of 0 represents black

# A value of 1 represents white

# A value of -1 represents the green mask

for iRow from 0 <= iRow < iHeight:

   for iColumn from 0 <= iColumn < iWidth:

      # Check to see if we are outside of the green mask
```

```
        if((imageTuple[iRow * iWidth + iColumn][0] + imageTuple[iRow * iWidth + iColumn][1])
!= 255):
            # We are outside of the green mask, transfer the value into a C array
            if(imageTuple[iRow * iWidth + iColumn][0] == 0):
                image[iRow * iWidth + iColumn] = 0
            else:
                image[iRow * iWidth + iColumn] = 1
        else:
            # The pixel is green
            image[iRow * iWidth + iColumn] = -1

    # Store the python random walk variables as C variables
    fScale = variableList[0]
    iEdgeLength = variableList[1]
    iIterations = variableList[2]

    # Seed the random number generator with the system time
    random.seed()

    # Initialize our net array
    for iCounter2 from 0 <= iCounter2 < pow(iEdgeLength, 2):
        net[iCounter2] = 0

    # Loop where iterations are performed
```

```
for iCounter from 1 <= iCounter <= iIterations:
    # Initialize our variables that ensure valid point selection
    iIsGreen = 0
    iIsOB = 0

    # Enter a loop where we pick points until we get valid ones
    while(iIsGreen == 0 or iIsOB == 0):
        # Assume all points in our net are valid -- until we find otherwise
        iIsGreen = 1
        iIsOB = 1
        # Choose the center and angle of our net
        fX = iWidth * random.random()
        fY = iHeight * random.random()
        fThetaRandom = 2 * 3.14159 * random.random()
        # Enter a loop check all n x n grid points for validity
        for iNetRow from 0 <= iNetRow < iEdgeLength:
            for iNetColumn from 0 <= iNetColumn < iEdgeLength:
                iXOffset = (iNetColumn - floor(iEdgeLength/2))
                iYOffset = (iNetRow - floor(iEdgeLength/2))
                fHyp = sqrt(iXOffset * iXOffset + iYOffset * iYOffset)
                if(fHyp != 0):
                    # Determine which of the four quadrants we are in
                    if(iYOffset >= 0):
                        # We are in quandrant I or quadrant II
```

```
            fThetaRaw = acos(iXOffset/fHyp)
        else:
            # We are in quadrant III or quadrant IV
            fThetaRaw = 2 * 3.14159265 - acos(iXOffset/fHyp)
    # Adjust our random angle with our net point angle
    fThetaAdjusted = fThetaRandom + fThetaRaw
    # Calculate real hypotenuse
    fHyp = fHyp * fScale / 2
    fHyp = fHyp / sqrt(2 * floor(iEdgeLength/2) * floor(iEdgeLength/2))
    fXStar = fX + fHyp * cos(fThetaAdjusted)
    fYStar = fY + fHyp * sin(fThetaAdjusted)
    # Now we have the x and y coordinates of a particular net point
    # Now we can check to see if these points are in bounds
    iX = fXStar
    iY = fYStar
    if(iX < 0 or iX >= iWidth of iY < 0 or iY >= iHeight):
        iIsOB = 0
        break
    # If the point was not OB, it may be green
    if(image[iY * iWidth + iX] == -1):
        iIsGreen = 0
        break
# End of iNetColumn for loop
# Check to see if we are OB or the point is green
```

```
    if(iIsOB == 0 or iIsGreen == 0):
      break
  # End of iNetRow for loop
# End iIsOB or iIsGreen while loop

# If we've made it this far we have a valid net of points
for iNetRow from 0 <= iNetRow < iEdgeLength:
  for iNetColumn from 0 <= iNetColumn < iEdgeLength:
    iXOffset = (iNetColumn - floor(iEdgeLength/2))
    iYOffset = (iNetRow - floor(iEdgeLength/2))
    fHyp = sqrt(iXOffset * iXOffset + iYOffset * iYOffset)
    if(fHyp != 0):
      # Determine which of the four quadrants we are in
      if(iYOffset >= 0):
        # We are in quandrant I or quadrant II
        fThetaRaw = acos(iXOffset/fHyp)
      else:
        # We are in quadrant III or quadrant IV
        fThetaRaw = 2 * 3.14159265 - acos(iXOffset/fHyp)
    # Adjust our random angle with our net point angle
    fThetaAdjusted = fThetaRandom + fThetaRaw
    # Calculate real hypotenuse
    fHyp = fHyp * fScale / 2
    fHyp = fHyp / sqrt(2 * floor(iEdgeLength/2) * floor(iEdgeLength/2))
```

```
fXStar = fX + fHyp * cos(fThetaAdjusted)

fYStar = fY + fHyp * sin(fThetaAdjusted)

# Now we have the x and y coordinates of a particular net point

# Now we can check to see if these points are in bounds

iX = fXStar

iY = fYStar

# Test the color of one particular net pixel

iIndex = iNetRow * iEdgeLength + iNetColumn

iPow = iCounter % 8

if(image[iY * iWidth + iX] == 0):

   # Our pixel is black

   # We always add zero if the pixel is black, so we take no action

elif(image[iY * iWidth + iX] == 1):

   # Our pixel is white

   net[iIndex] = net[iIndex] + pow(2, iPow)

else:

   # We should never see this!

   print "We have an error in the code that compresses the eight net values"

# See if we have 8 values in our array, if so compress them into a single line in a .txt file

if(iIterations % 8 == 0):

   # Compress the values into a single line of a .txt file

   for iCounter2 from 0 <= iCounter2 < pow(iEdgeLength, 2):

      # If we need any leading zeros, add them here

      if(net[iCounter2] > 99):
```

```
            str = str + net[index]
          elif(net[iCounter2] > 9 and net[index] < 100):
            str = str + '0' + net[index]
          elif(net[iCounter2] >= 0 and net[index] < 10):
            str = str + '00' + net[index]
          else:
            print "We have an error in the code section that adds leading zeros"
          # Add a space after the value
          str = str + ' '
          # Write the string to the file
          fout.write(str)
          # initialize our array for the next round
          net[iCounter2] = 0
        # End of iCounter2 for loop
      # End of .txt line write
    # End of iNetColumn for loop
  # End of iNetRow for loop
# End of iIterations for loop

# Free the memory associated with the malloc call
free(image)
free(net)

# Close the text file we've created
```

```
fout.close()

# Successfully return
return 0
```

Appendix C: Template Generating Function

```
# Import native Python modules

import comparison, os, shutil, sys, time

# Import our custom made Pyrex and Python extensions

import fpBnW01, fpWalk01

import ImageAnalyzer


# Add the path where the fingerprints are stored

imageDirectory = 'C:\\Python25\\Lib\\site-packages\\Pyrex\\Distutils\\2002 fvc 110 by 8\\Processed\\'

textFileDirectory = 'C:\\Python25\\Lib\\site-packages\\Pyrex\\Distutils\\2002 fvc 110 by 8\\textFiles\\'

os.sys.path.append(imageDirectory)

os.sys.path.append(textFileDirectory)


# Take care of some pre-loop needs

startTime = time.time()

iterations = 500000

startingScale = 0

endingScale = 30

increment = 0.5


# A text file will be created for each of the 880 outlined images

for iLoop in range(6,7):
```

```
    for iLoop2 in range(5,6):
        sName = str(iLoop) + '_' + str(iLoop2) + '.bmp'
        image = ImageAnalyzer.ImageAnalyzer(imageDirectory, sName)
        results = image.monochromeWalk(iterations, startingScale, endingScale, increment)
        if (type(results) == type(1)):
            print '\nprint was skipped\n'
        else:
            p1TextFile = open(textFileDirectory + str(iLoop) + '_' + str(iLoop2) + '.txt','w')
            p1TextFile.write(str(iLoop) + '_' + str(iLoop2) + 'b.txt' + '\n')
            p1TextFile.write('Created on ' + str(time.asctime()) + '\n')
            p1TextFile.write('File created by Joseph M. Stoffa\n\n')
            p1TextFile.write(' D(mm)\t Pbb\t Pww\t Pwb\n------\t------\t------\t------\n')
            for iLoop3 in range(0, int(float(endingScale)/float(increment)) + 1):
                scale = str('%.4f'%(results[iLoop3][0] * 0.08467))
                p1TextFile.write(scale + '\t' + str(results[iLoop3][1]) + '\t')
                p1TextFile.write(str(results[iLoop3][2]) + '\t' + str(results[iLoop3][3]) + '\n')
            p1TextFile.close()
            print 'Text file for ', sName, ' has been created'

totalTime = time.time() - startTime
print 'The total time taken was ', totalTime
print 'FIN'
```

Appendix D: Matching Score Calculation Function

```
# Import the necessary Python libraries
import random


def comparison(printOne, printTwo, metric, quantity, startingScale = 0, endingScale = 9999):
    # This is where the module documentation is stored
    """
    The comparison module compares two fingerprints
    The listData input is a python list containing data on two fingerprints
        printOne[0][0...n] is the scale data for fingerprint one
        printOne[1][0...n] is the alpha1 data for fingerprint one
        printOne[2][0...n] is the alpha2 data for fingerprint one
        printOne[3][0...n] is the beta data for fingerprint one
        printTwo[0][0...n] is the scale data for fingerprint two
        printTwo[1][0...n] is the alpha1 data for fingerprint two
        printTwo[2][0...n] is the alpha2 data for fingerprint two
        printTwo[3][0...n] is the beta data for fingerprint two
    The "metric" variable refers to the method of comparison
    The following text arguments are valid for the metric variable
        linear -- Computes the linear distance between two spectra
        square -- Computes the square of the linear distance between two spectra
        FFT -- computes the linear difference between the Fourier transform of two spectra
    The "quantity" variable determines what the "metric" compares
```

```
The following text arguments are valid for the quantity variable
    alphaOne -- The quantity measured by the metric will be the probability of white-white
    alphaTwo -- Quantity compared will be probability of black-black
    beta -- Quantity compared will be probability of black-white + white-black
    determinant -- Quantity compared will be the determinant of the 2x2 matrix
    eigenValues -- Quantity compared will be the Eigenvalues of the 2x2 matrix
    trace -- Quantity compared will be the trace of the 2x2 matrix
The startingScale and endingScale determine which section of spectra undergoes comparison
If these areguments are omitted, all scales will be compared
The startingScale and endingScale are inclusive, these scales will be compared
Any two numerical arguments for the startingScale and endingScale are valid given that
    The startingScale is less than the endingScale
"""

# Check to see if arguments are valid
# Check if listData is a variable of type list
if(type(printOne) != type([]) or type(printTwo) != type([])):
    # Our input is not a list, we need to return an error value
    return -230
# Check if data representing alphas and beta is in integer form
for iLoop in range(1,4):
    if(type(printOne[iLoop][0]) != type(1) or type(printTwo[iLoop][0]) != type(1)):
        return -231
# Check that the user has selected an acceptable metric
```

```
if(metric != 'linear' and metric != 'percentage' and metric != 'square'):
    return -232
# Check that the user has selected a valid quantity to comapre
if(quantity != 'alphaOne' and quantity != 'alphaTwo' and quantity != 'beta'):
    if(quantity != 'determinant' and quantity != 'eigenValues' and quantity != 'trace'):
        if(quantity != 'ndeterminant' and quantity != 'random'):
            return -233
# Check to ensure the starting scale is smaller than the ending scale
if(startingScale > endingScale):
    return -234
# We need to determine the number of entries in printOne
entries = len(printOne[0])
# We need to determine the index of our startingScale
startingIndex = 0
while(startingScale > printOne[0][startingIndex]):
    startingIndex = startingIndex + 1
    if(startingIndex > entries):
        # Our startingScale is greater than the last scale in the data list
        return -235
# Determine the number of iterations that occur at each scale of fingerprint 1
# This assumes the number of iterations that occured at the first scale occur at all scales
# Also, fingerprint 1 and 2 must have the same number of iterations at each scale
#   Otherwise, the comparison result will be fingerprint order specific
iterations = printOne[1][0] + printOne[2][0] + 2*printOne[3][0]
```

```
alphaOneW0 = printOne[1][0]

alphaOneB0 = printOne[2][0]

alphaTwoW0 = printTwo[1][0]

alphaTwoB0 = printTwo[2][0]

# Initialize some variables before entering our loop

differenceSum = 0

iCounter = startingIndex

iScalesCompared = 0

# See if we reach the endingScale before we reach the last index in the print

while(iCounter < entries and printOne[0][iCounter] <= endingScale):

    # Make certain our two scales are equal

    if(printOne[0][iCounter] != printTwo[0][iCounter]):

        return -236

    # Determine the difference based on our metric

    if(quantity == 'alphaOne'):

        # Calculate the difference between the alpha ones

        difference = abs(printOne[1][iCounter] - printTwo[1][iCounter])

    elif(quantity == 'alphaTwo'):

        # Calculate the difference between the alpha twos

        difference = abs(printOne[2][iCounter] - printTwo[2][iCounter])

    elif(quantity == 'beta'):

        # Calculate the difference between the betas

        difference = abs(printOne[3][iCounter] - printTwo[3][iCounter])

    elif(quantity == 'determinant'):
```

```
        # Calculate the difference between the two determinants
        difference = (printOne[1][iCounter]*printOne[2][iCounter] - pow(printOne[3][iCounter], 2))
        difference = abs(difference - (printTwo[1][iCounter]*printTwo[2][iCounter] - pow(printTwo[3][iCounter], 2)))
    elif(quantity == 'random'):
        # Calculate the difference between the two determinants
        difference = random.random()
    elif(quantity == 'ndeterminant'):
        # Calculate the difference between the two determinants
        difference = (printOne[1][iCounter]*printOne[2][iCounter] - pow(printOne[3][iCounter], 2))/alphaOneW0/alphaOneB0
        difference = abs(difference - ((printTwo[1][iCounter]*printTwo[2][iCounter] - pow(printTwo[3][iCounter], 2))/alphaTwoW0/alphaTwoB0))
        """
        elif(quantity ==  'eigenvalues'):
        # Do something else
        return -237
        """
    elif(quantity == 'trace'):
        # Calculate the difference between the two traces
        difference = printOne[1][iCounter] + printOne[2][iCounter]
        difference = abs(difference - printTwo[1][iCounter] - printTwo[2][iCounter])
    else:
        # We should never see this line of code
        return -237
```

```
    error = (pow(printOne[1][iCounter], 0.5) + pow(printTwo[1][iCounter], 0.5))/2*2*.7
    if(quantity == 'determinant'):
      error = pow(error, 2)

    if(metric == 'square'):
      difference = pow(difference, 2)
      error = pow(error, 2)

    if(metric == 'percentage'):
      if(difference < error):
        difference = 0
      else:
        difference = 1

    # We've completed on difference calculation
    iCounter = iCounter + 1
    if(metric == 'linear' or metric == 'square'):
      differenceSum = differenceSum + float(difference)/float(error)
    if(metric == 'percentage'):
      differenceSum = differenceSum + difference
    iScalesCompared = iScalesCompared + 1

  # The main loop is over
```

```
# We can now return the average difference between two scale spectra

if(metric == 'linear' or metric == 'square'):
    return float(differenceSum/float(iScalesCompared))
if(metric == 'percentage'):
    return float(1.0 - float(differenceSum)/float(iScalesCompared))
```