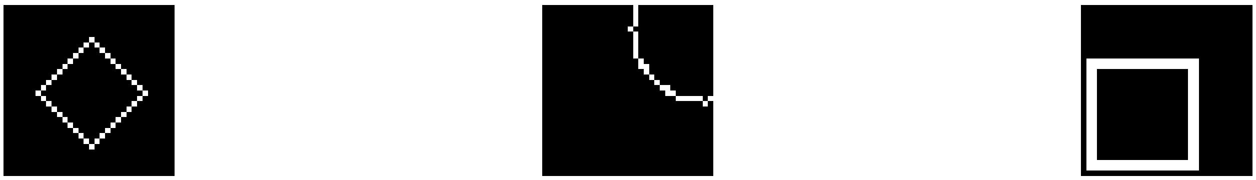
The file that contains the function definitions for these examples, `ball.py`, is listed in the appendix. The specific script used to produce the output for each problem are listed with problem itself. All images are scaled to 64×64`px` for viewing purposes.

1 Problem 1



(a) (b) (c)

Figure 1: Loci of Points

Figure 1 comprises the three 32×32 binary images created. Each image was made using the `make_ball()` function with different parameters passed to it. The parameters are tabulated in Table 1.

	Figure 1a	Figure 1b	Figure 1c
Center, (i_0, j_0)	(16,16)	(4,30)	(20,11)
Outer Radius, r	10	14	10
Range of contours, d	1	1	2
Distance Metric	D_4, "city block"	D_E, Euclidean	D_8 , "chessboard"

Table 1: Program Parameters

Figure 1a shows the isodistance contour produced when using "city block" distance as the metric (D_4) . Figure 1bshows a contour with produced using the Euclidean metric (D_E), and with a shifted center. Figure 1c demonstrates the isodistance contour created when using the "chessboard" distance metric (D_8) , over a range of distances and with a shifted center.

The isodistance contours take on very different shapes depending on the metric used and the radius chosen. With the Euclidian metric, it forms a discrete circular ring. Using the 4-connectivity metric, the contour depicts a diamond. With the 8-connectivity metric, it is a square. When the radius is small enough, however ($r < 3$), then D_4 and D_E provide give the same locus of points.

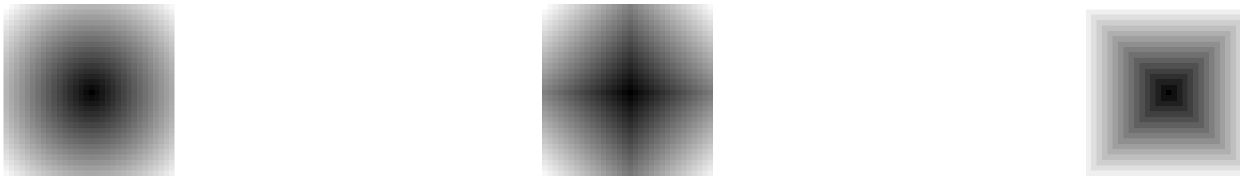
If the background contiguity is determined by 8-connectivity, then the D_8 metric is the only metric that can be used to create a single equidistant contour that separates the background into two non-contiguous regions. However, if such a contour is made over a range of distances, ($2 \leq d < r$) then all three metrics separate the background into two non-contiguous regions.

`problem1.py`

```
from ball import *

ball1 = make_ball(10, dmetric='4')
ball2 = make_ball(14, shell=1, center=(4,30))
ball3 = make_ball(8, r2=10, center=(20,11), dmetric='8')
imsave('1-1a.png',ball1)
imsave('1-1b.png',ball2)
imsave('1-1c.png',ball3)
```

2 Problem 2



(a) D_E (b) D_4 (c) D_8

Figure 2: Distance Maps

The images in Figure 2 are distance maps created using the different metrics, calculated as distance from the center. Darker shades represent smaller distances, and lighter shades represent larger distances. Figure 2a was created using the Euclidean metric. Figure 2b was created using the "city block" metric. Figure 2c was created using the "chessboard" metric.

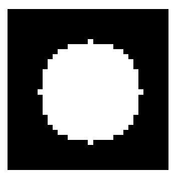


Figure 3: Circular Region

The region in Figure 3 was created using the distance map, Figure 2a, binarizing the image using the threshold radius, $r = 10$. The region corresponds to the run-length code in `out2.txt`.

out2.txt

```
(6,16,16) (7,12,20) (8,10,22) (9,9,23) (10,8,24) (11,8,24) (12,7,25) (13,7,25)
    (14,7,25) (15,7,25) (16,6,26) (17,7,25) (18,7,25) (19,7,25) (20,7,25) (21,8,24)
    (22,8,24) (23,9,23) (24,10,22) (25,12,20) (26,16,16)
```

problem2.py

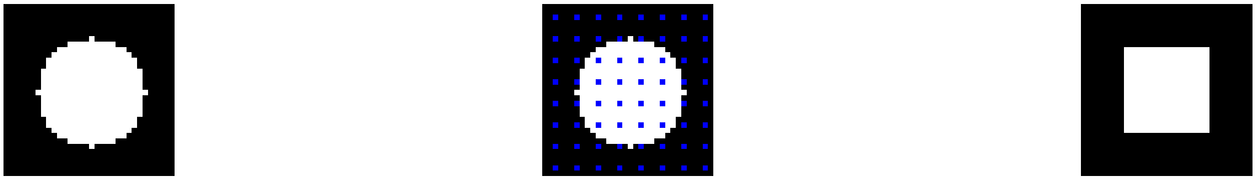
```python
from ball import *

ball1 = make_ball(10, dmetric='euclid', ret='img')
ball2 = make_ball(10, dmetric='4', ret='img')
ball3 = make_ball(10, dmetric='8', ret='img')
ball4 = make_ball(10, shell=11)

imsave('1-2a.png',ball1)
imsave('1-2b.png',ball2)
imsave('1-2c.png',ball3)
imsave('1-2d.png',ball4)

with open('out2.txt','w') as outfile:
    outfile.write(im2rl(ball4))
```

3 Problem 3



(a) Read Image (b) Image and Sampling Grid (c) Resampled Output

Figure 4: Reading and Sub-sampling

Figure 4a displays the decoded input from the run-length code in Problem 2, 4b the image with the sampling grid and 4c, the 8×8 re-sampled ball. The output quad tree code is displayed in `out3.txt`. The quad tree code was encoded from top left across the row, and then down through the columns.

`out3.txt`

```
g( g( b b b w ) g( b b w b ) g( b w b b ) g( w b b b ) )
```

`problem3.py`

```
from ball import *


im = rl2im(None,fn='out2.txt')
im2,samples = resample(im,newshape=(8,8))

qt = im2qt(im2)

imsave('1-3a.png',im)
imsave('1-3b.png',(im-im*samples,im-im*samples,im*(1-samples)+samples))
imsave('1-3c.png',im2)

with open('out3.txt','w') as outfile:
    outfile.write(qt)
```

4 Problem 4

To demonstrate its capability, I had `problem4.py` use `int2bin()` to convert the digits from 0 to 16 into binary. As an added feature, I had it pad them so they are all 4 digit binary strings. The output is listed below:

out4.txt

```
0	0000	1	0001
2	0010	3	0011
4	0100	5	0101
6	0110	7	0111
8	1000	9	1001
10	1010	11	1011
12	1100	13	1101
14	1110	15	1111
```

problem4.py

```python
import numpy as np

#convert integer to binary number string
def int2bin(n, pad=1):
    if n==0: return '0'*pad
    nbits = int(np.ceil(np.log2(n+1)))
    binrep = np.zeros(nbits)
    temp = n
    for i in np.arange(nbits)[::-1]:
        if temp < 2**i:
            pass
        else:
            binrep[i]=1
            temp -= 2**i
    binstr = ''
    for i in binrep[::-1]:
        binstr += str(int(i))

    #if the bin number has too few characters
    #then pad the front with zeros
    if pad>len(binstr):
        binstr = '0'*(pad-len(binstr))+binstr
    return binstr

with open('out4.txt','w') as outfile:
    for i in range(16):
        outfile.write('{}\t{}\t'.format(i,int2bin(i,4)))
        outfile.write('\n'*(i%2))
```

 A. Athanassiadis

5 Appendix: Common Code

Common functions used for these problems are contained in `ball.py`. Note that `**` denotes raising to a power, and "numpy" (np) is a library which adds mathematical and matrix functionality similar to what is default in Matlab and IDL.

ball.py

```
import numpy as np
from scipy.misc import imsave

#define various metrics for distances
def euclid_dist(pt1, pt2=[0]):
    #convert input into useful np.array datatype
    pt1 = np.array(pt1)
    pt2 = np.array(pt2)
    #make sure the points have the same dimensionality, or that one is a constant
    if pt2.ndim==0: pt2 = np.array([pt2])
    if (pt1.ndim == pt2.ndim) or (pt2.ndim == 1):
        return np.sqrt(sum((pt1-pt2)**2))
    else:
        print 'euclid_dist: distance not computable.  please give two points with the same dimensions'
        return 0

def four_dist(pt1, pt2=[0]):
    pt1 = np.array(pt1)
    pt2 = np.array(pt2)
    if pt2.ndim==0: pt2 = np.array([pt2])
    if (pt1.ndim == pt2.ndim) or (pt2.ndim == 1):
        return sum(abs(pt1-pt2))
    else:
        print 'four_dist: distance not computable. please give two points with the same dimensions'
        return 0

def eight_dist(pt1, pt2=[0]):
    pt1 = np.array(pt1)
    pt2 = np.array(pt2)
    if pt2.ndim==0: pt2 = np.array([pt2])
    if (pt1.ndim == pt2.ndim) or (pt2.ndim == 1):
        return max(abs(pt1-pt2))
    else:
        print 'eight_dist: distance not computable. please give two points with the same dimensions'
        return 0

#dictionary of distance algorithms for easy user access later
distalgs = {'euclid': euclid_dist, '4': four_dist, '8': eight_dist}

#draw a ball
#shell can be specified by outer radius and thickness (default thickness 1px)
#or by two bounding radii
#uses matrix coordinates (row,col)
def make_ball(r, shell=1, r2=None, bgsize=(32,32), center=None, dmetric='euclid', ret='mask'):
    #make sure the desired distance algorithm is valid
```

```
    if dmetric not in distalgs:
        dmetric = 'euclid'
        print 'make_ball: invalid metric. proceeding with euclidean metric'

    dist = distalgs[dmetric]

    #initialize input variables to array datatype (np.array)
    #set center to middle of image if not specified
    bgsize = np.array(bgsize)
    if center==None:
        center = bgsize/2

    #init image
    img = np.zeros(bgsize)

    #compute distances to center
    for x in range(bgsize[0]):
        for y in range(bgsize[1]):
            img[x,y] = dist((x,y), center)

    #create binary mask based on distances
    if r2 == None:
        mask = (img <= r) * (img > r-shell)
    else:
        mask = (img <= max([r, r2])) * (img > min([r,r2]))

    #if desired, only return distance map, else just return mask
    if ret=='img':
        return img
    else:
        return mask

#resample an image evenly into newshape
def resample(im,newshape=(8,8)):
    samples = np.zeros(im.shape)
    sim = np.zeros(newshape)

    #get the sample spacing in each direction
    #and the appropriate shift to center the sampling
    spacing = np.array(im.shape,dtype=np.float)/np.array(newshape,dtype=np.float)
    shift = spacing/2

    for n in range(newshape[0]):
        for m in range(newshape[1]):
            #np.around() to round indeces to nearest int
            samples[np.floor(n*spacing[0]+shift[0]),\
                    np.floor(m*spacing[1]+shift[1])] += 1

            sim[n,m] = im[np.floor(n*spacing[0]+shift[0]),\
                          np.floor(m*spacing[1]+shift[1])]

    return sim,samples

#turn a binary image into run-length data
def im2rl(im):
    arr=np.array(im)
```

```
    rl = ''
    opn = 0    #keep track of whether we're in the image or in the bkg
    for r in range(arr.shape[0]):
        for c in range(arr.shape[1]):
            if arr[r,c]==0 and opn==1: #if black and was white
                    rl += '{}) '.format(c-1)
                    opn = 0
            if arr[r,c]==1 and opn==0: #if white and was black
                    rl += '({},{},'.format(r,c)
                    opn = 1
        if opn==1:  #if last pix in row was white
            rl += '{}) '.format(c)
            opn = 0

    return rl

#decode run-length code into binary image
def rl2im(rl, fn=None, imdim=(32,32)):
    im = np.zeros(imdim)
    if not(fn==None):
        with open(fn,'r') as infile:
            rl=infile.readline()

    for code in rl.split():
        try:
            [r,c1,c2] = [int(i) for i in code[1:-1].split(',')]
            for c in range(c1,c2+1):
                im[r,c] = 1
        except:
            print 'improper input format. skipping input: {}\n'.format(code)

    return im

#convert image into quad-tree code
#encodes from top left of the quadrant at each level
def im2qt(im):
    arr = np.array(im)
    scores = []
    tmpscores = []
    qtcode = ''
    s1,s2 = arr.shape

    #uniformity metric is all black or all white.  Gray causes a daughter node to be created.
    if arr.sum()==0:
        qtcode = 'b '
    elif arr.sum()==np.prod(arr.shape):
        qtcode = 'w '
    else:
        qtcode += 'g( '
        for i1,i2 in [(0,s1/2),(s1/2,s1)]:
            for j1,j2 in [(0,s2/2),(s2/2,s2)]:
                qtcode += im2qt(arr[i1:i2,j1:j2])
        qtcode += ') '

    return qtcode
```