Simulation

Natural Computing Homework

Stephanie Athow Chris Smith

May 2, 2015

Contents

List of Figures

List of Tables

List of Algorithms

Document Preparation and Updates

Current Version [X.X.X]

Prepared By:
Stephanie Athow #1
Chris Smith#2

Revision History

Date	***Author***	***Version***	***Comments***
2/2/15	*Team Member #1*	*1.0.0*	*Initial version*
3/4/15	*Team Member #2*	*1.1.0*	*Edited version*

1

Fractals - Text Chapter 7

1.1 Problem 10

Implement a bracketed OL-system and reproduce all plant-like structures of Figure 7.24 [of the book]. Change some derivation rules and see what happens. Make your own portfolio with, at least, ten plants.

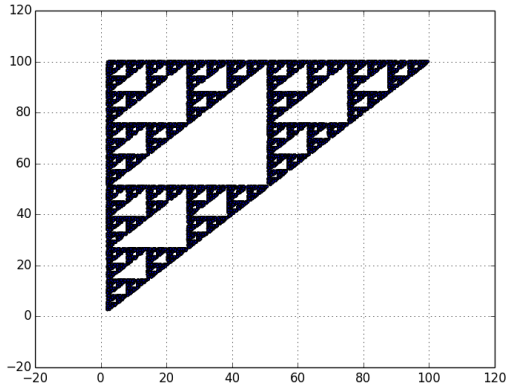
1.2 Problem 15

Implement a RIFS to generate all the fractals whose codes are presented in Table: 1.2

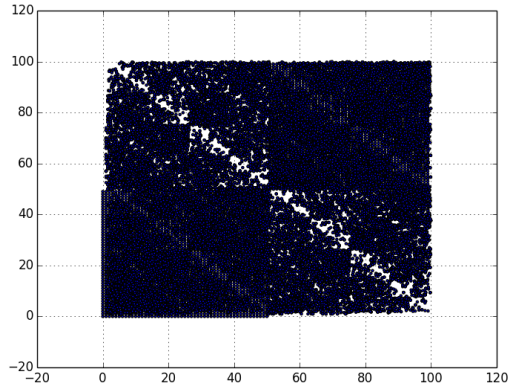
RIFS is an acronym for Random Iterated Function System (RIFS). An iterated function system (IFS) recursively applies a set of affine transformations to each point in a point list. Generally, the affine transformations are a contractive mapping. Unless you are zooming in, there is a depth at which the points are very dense and not all of them need to be mapped. This is where the RIFS comes into play. Each point in the point list has one of the functions of the set applied to it. The resolution is still good and the program runs much faster because there are few computations done.

The codes (functions) given in 1.2 are defined such that w is the function number, a, b, c, d are scaling factors, e, f are offset values and p is the probability the function will be selected. These values are used in the equation for affine transformation 1.2. The Sierpinski Gasket and Square fractals must have an initial point list to perform the transformations. The Barnsley Fern and Tree need only a single point to perform the transformations. Figure 1.1 shows the Sierpinski Gasket, Square, Bransley Fern and Tree generated using the RIFS codes.

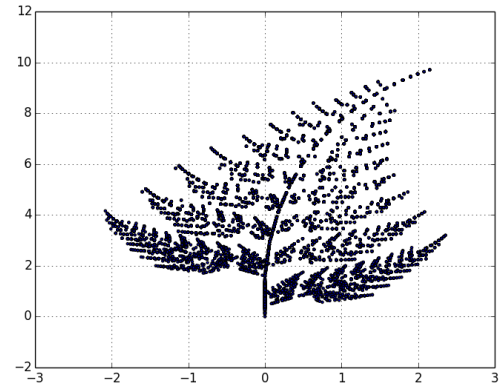
$$w(x_1, x_2) = (ax_1 + bx_2 + e, cx_1 + dx_2 + f) \tag{1.1}$$



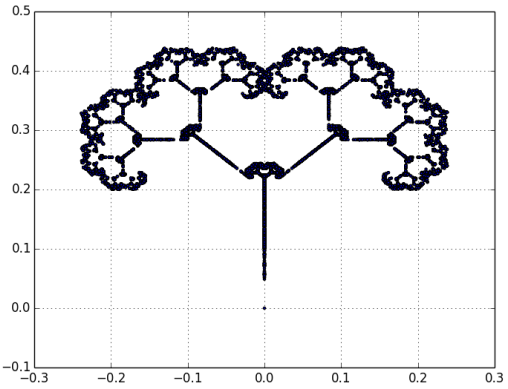
(a) Sierpinski Gasket



(b) Square



(c) Barnsley Fern



(d) Tree

Figure 1.1: Plots of Various Fractals

Table 1.1: RIFS codes to generate fractals

(a) Sierpinski Gasket

w	a	b	c	d	e	f	p
1	0.5	0	0	0.5	1	1	0.33
2	0.5	0	0	0.5	1	50	0.33
3	0.5	0	0	0.5	50	50	0.34

(b) Square

w	a	b	c	d	e	f	p
1	0.5	0	0	0.5	1	1	0.25
2	0.5	0	0	0.5	50	1	0.25
3	0.5	0	0	0.5	1	50	0.25
4	0.5	0	0	0.5	50	50	0.25

(c) Barnsley Fern

w	a	b	c	d	e	f	p
1	0	0	0	0.16	0	0	0.01
2	0.85	0.04	-0.04	0.85	0	1.6	0.85
3	0.2	-0.26	0.23	0.22	0	1.6	0.07
4	-0.15	0.28	0.26	0.24	0	0.44	0.07

(d) Tree

w	a	b	c	d	e	f	p
1	0	0	0	0.5	0	0	0.05
2	0.42	-0.42	0.42	0.42	0	0.2	0.40
3	0.42	0.42	-0.42	0.42	0	0.2	0.40
4	0.1	0	0	0.1	0	0.2	0.15

1.3 Problem 21

Implement the random midpoint displacement algorithm in 3D and generate some fractal landscapes. Study the influence of H on the landscapes generated.

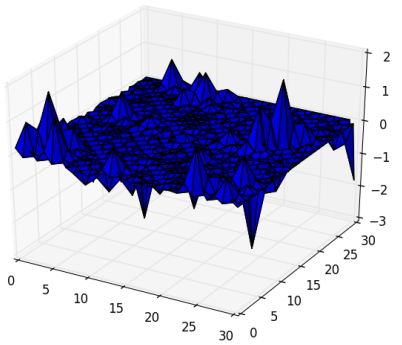
In 2D, the random midpoint displacement starts with a line, determines the midpoint and randomly perturbs it using equations 1.3 and 1.3. In equation 1.3, $\Delta(i)$ stores the value for depth i, σ is the standard deviation of the Gaussian distribution, H is the 'roughness factor'. H is always $0 < H < 1$. The closer to 0, the more rough the surface will be. The closer to 1, the more smooth the surface will be. In equation 1.3 x_1 is the new value for the midpoint, x_0 is the left end point, x_2 is the right end point, $\Delta(t)$ is from equation 1.3, t is the recursion depth and $rand$ is a random number. Now, there are two line segments and the midpoint of each line segment is perturbed. This method is recursively applied while decreasing the perturbation amount for each depth.

$$x_1 = 0.5(x_0 + x_2) + \Delta(t)rand \tag{1.2}$$

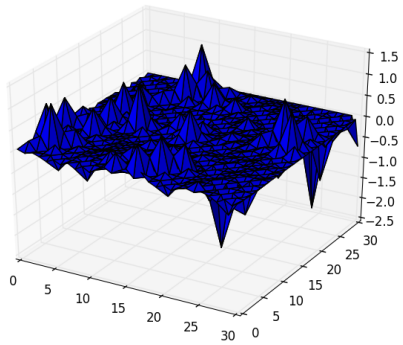
$$\Delta(i) = \sigma(H)^{(i+1)/2)} \tag{1.3}$$

The random midpoint displacement for 3D starts with a 2D box. The box is subdivided using the midpoints of the edges, so one box becomes four boxes, four boxes becomes sixteen, ect. The intersections of the midpoint lines becomes the perturbation points. They are perturbed in the same manner as the 2D method except the average is the average of the four surrounding points. Figure 1.2 shows some surfaces with varying H values.

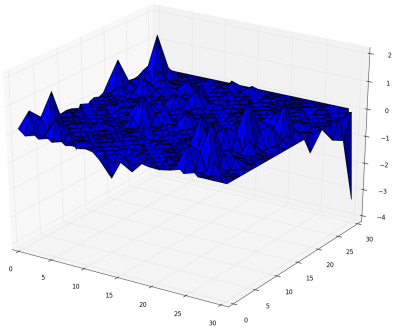
t



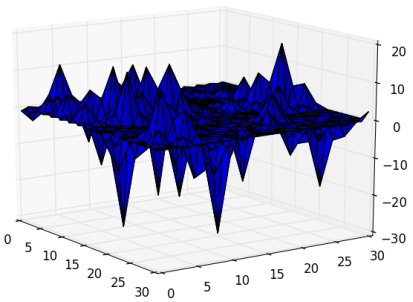
(a) $\sigma = 1.5$, $H = 0.2$



(b) $\sigma = 1.5$, $H = 0.5$



(c) $\sigma = 1.5$, $H = 0.75$



(d) $\sigma = 5$, $H = 0.75$

Figure 1.2: Brownian Surfaces with Varying σ's and H's

2

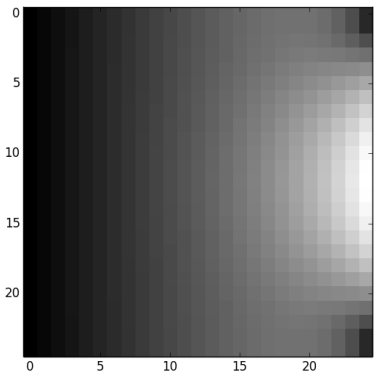
Cellular Automata - Chapter 7

2.1 Problem: Slides 1

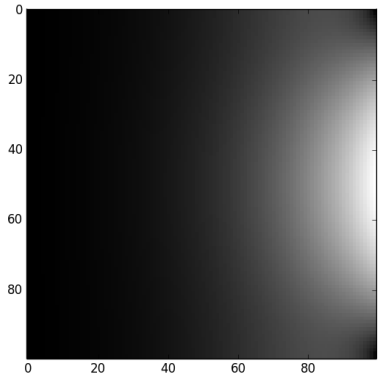
Modify the heat flow example to deal with insulated conditions on the top and bottom boundary. Insulation means zero flux or u[N][j] = u[N-1][j]. This implies that instead of a fixed valued ghost points on the top and bottom, you modify the CA rule using the previous relation.

Cellular automata can be used to study some types of dynamics. In this case, a box with insulated top and bottom walls, and a parabolic heat source on the right wall is presented. The insulated conditions means that the top and bottom boundaries are equal to the grid square below and above them. Using the four neighbors rule, the simplified equation 2.1 is applied to $N-2$ by $N-2$ squares.

Figure 2.1 shows a grid of 25 x 25 on the left and a grid of 100 x 100 on the right. With the finer grid, a much smoother gradient appears



(a) 25 x 25 grid, Temperature = 10



(b) 100 x 100 grid, Temperature = 10

Figure 2.1: Example Heat Flow Grids

$$c_{i,j}(t+1) = (c_{i-1,j}(t) + c_{i+1,j}(t) + c_{i,j-1}(t) + c_{i,j+1}(t))/4 \tag{2.1}$$

2.2 Problem: Slides 2

Reproduce patterns theta, lambda, mu, alpha in the Gray-Scott Model (CA). You don't need to follow their color scheme.

The Gray-Scott model is used to model pattern formation. The general idea is there's a steady-state grid of some chemical, let's call it u. Then, it is perturbed with noise and in the middle with a second chemical, call this one v, which reacts with the first chemical. With time, the chemicals reach a steady-state again and a pattern emerges. In equations 2.2 and 2.2, u and v are the chemicals reacting, F and k are tuneable parameters which will produce different patterns. Typical values are $0.00 < F < 0.08$ and $0.03 < k < 0.07$ and Figure 2.2 shows values that produce various named patterns.

Using Dr. McGough's code provided on his class website (http://www.mcs.sdsmt.edu/jmcgough/csc492/) and changing the F and k parameters, the images in Figure 2.3 were created. Patterns α, θ, and λ are very similar to the examples given in the lecture slides. However, pattern μ does not compare well to the example. Potential sources of error are gize size, grid granuality or an incorrectly estimated value.

$$\frac{\partial u}{\partial t} = d_1 \Delta u - uv^2 + F(1-u) \tag{2.2}$$

$$\frac{\partial v}{\partial t} = d_2 \Delta v + uv^2 - (F+k)v \tag{2.3}$$

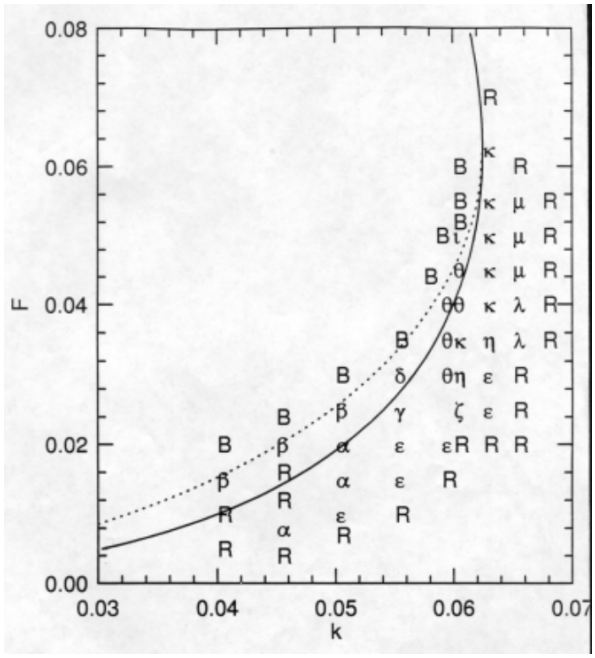
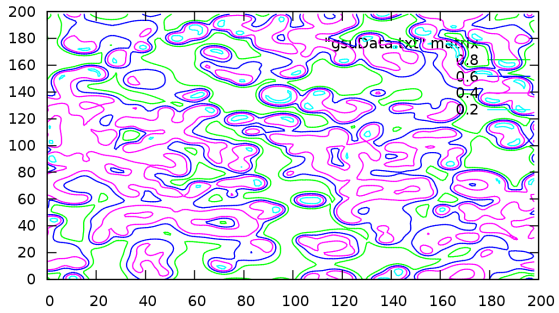
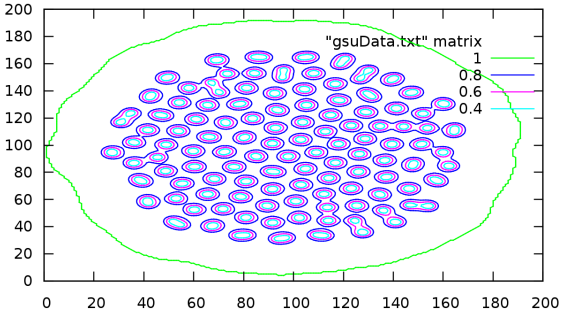


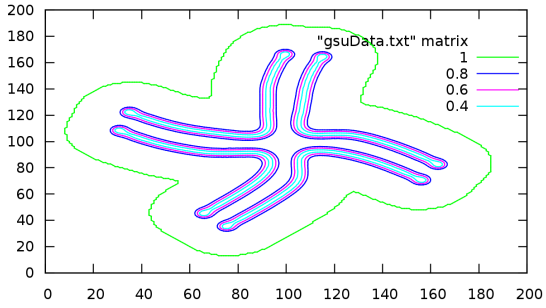
Figure 2.2: Gray-Scott Chart



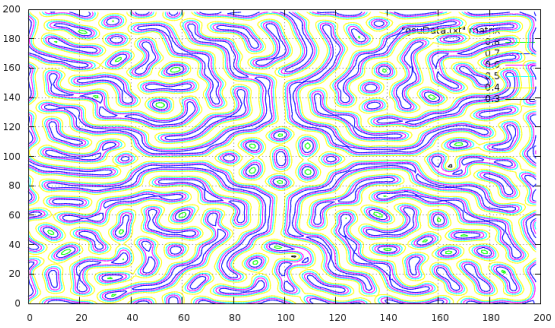
(a) pattern α



(b) pattern λ



(c) pattern μ



(d) pattern θ

Figure 2.3: Various Gray-Scott Patterns

3

ALife - Text Chapter 8

3.1 Problem 3

**Choose on e of the sample project of StarLogo and solve its exploration tasks (http://education.mit.edu/starlo
Write a brief report with the results obtained including any theoretical background knowledge
that may eventually be necessary to perform the explorations.**

3.2 Problem 4

Implement a bi-dimensional CA following the rules of "The Game of Life."

4

DNA Computing - Text Chapter 9

4.1 Problem 1

4.2 Problem 2

4.3 Problem 5

A

Supporting Materials

Supporting ...

B

Code

Listing B.1: ch7_15.py

```
##############################################################################
# Problem: 7.15, "Fundamentals of Natural Computing"
# Author: Stephanie Athow
# Date: 9 April 2015
# Problem Statement:
#  Implement a RIFS to generate all the fractals whose codes are presented in
#  Table 7.3. - Stored in Text file: IFS_Codes
##############################################################################

import numpy as np
import random
import matplotlib.pyplot as plt


MAX_POINTS = 10000
DEPTH = 5
XMAX = 100
YMAX = 100

##############################################################################
#                    Generate Valid Points
# Creates starting point list for Sierpinski Gasket and Square
# opt: 0 - Sierpinski Gasket
# opt: 1 - Square
##############################################################################
def genValidPoints( pointList, opt ):
   if ( opt == 0 ):
      y = 0
      x = 0

      for i in range ( 0, XMAX ):
         point = ( x, y )
         pointList.append( point )
         x += 1

      while ( x > 0 ):
         x += -1
```

```
            y += 1
            point = ( x, y )
            pointList.append( point )

        for i in range( 0, YMAX ):
            point = (0, y)
            pointList.append( point )
            y += -1

    if ( opt == 1 ):
        y = 0
        x = 0

        for i in range( 0, XMAX/2 ):
            for j in range( 0, YMAX/2 ):
                point = ( x, y )
                pointList.append( point )
                y += 1
            x += 1
            y = 0

        for i in range( XMAX/2, XMAX ):
            for j in range( YMAX/2, YMAX ):
                point = ( x, y )
                pointList.append( point )
                y += 1
            x += 1
            y = YMAX/2


##############################################################################
#                        Select Transformation
# Using the probability per transformation, selects which transformation is
#  done. Returns index number
##############################################################################
def selectTransformation( codes, code_num ):
    sum = 0
    function = 0
    max = 3
    prob = random.random()
    #print prob

    # NOTE: must change to (0,3) for gasket! otherwise (0,4)
    if ( code_num == 0 ):
        max = 4

    else:
        max = 5

    for function in range( 0, max ):
        sum = sum + float(codes[code_num][function][6])
        #print 'Sum: ', sum
        if( sum > prob ):
            break
```

```
   #print 'function: ', function
   return function

##############################################################################
#                     Select Valid Point
##############################################################################
def selectPoint( pointList ):
   length = len( pointList )
   i = random.randint( 0, length )
   return i

##############################################################################
#                  Run transformation on Point to DEPTH
# From Appendix B.4.6:
#  w(x1, x2) = ( a*x1 + b*x2 + e, c*x1 + d*x2 + f )
#     OR
#  w(X) = [ a b, c d] * [ x1, x2] + [ e, f ]
##############################################################################
def runTransformation( pointList, index, function ):
   #print "pointList: ", pointList
   x, y = pointList[ index-1 ]
   xnew, ynew = 0, 0
   a, b, c, d, e, f, p = function

   xnew = float(a)*x + float(b)*y + float(e)
   ynew = float(c)*x + float(d)*y + float(f)
   point = ( xnew, ynew )
   return point

##############################################################################
#                     Read in IFS Codes from file
# Reads in IFS codes from text file
##############################################################################
def readCodes(codes, titles, fractal_count):
   count = -1
   # open file
   f = open( 'IFS_Codes.txt', 'r' )

   for line in f:
      # remove added ''
      line = line.strip( '\'')

      # ignore blank lines
      if not line.strip( ):
         pass

      # ignore '#'
      elif( line[0] == "#"):
         pass

      # save into titles list
      elif( line[0] == '(' ):
         line = line.rstrip()
```

```
            titles.append(line)
            count += 1
            #print count
            #if( count > 1 ):
            codes.append([])

         # store into 2d list of codes
         else:
            line = line.rstrip()
            line = line.split()
            #print line
            codes[count].append(line)

      # close file
      f.close()

      #return fractal count
      return count

##############################################################################
#                        Plot Image of RFIS
# Plots image of the generated fractal
##############################################################################
def plotImage( pointList, title ):
      length = len( pointList )
      for i in range( 0, length ):
         x, y = pointList[i]
         plt.scatter( x, y, marker="." )

      plt.grid( True )
      #plt.show()
      plt.savefig( title + '.png' )
      plt.clf()

##############################################################################
#                            Main
##############################################################################
codes = []
titles = []
plist = []
plistnew = []
#pointlists = []
fractal_count = 0
i = 0

# read in codes from text file
fractal_count = readCodes( codes, titles, fractal_count )

# calculate and generate fractal images
for i in range( 0, fractal_count+1 ):
      # gasket and square - need initial list seeded
      if ( i < 2 ):
         genValidPoints( plist, i )
         for j in range( MAX_POINTS ):
```

```
            index = selectPoint( plist )
            #print "index: ", index
            for iterations in range( DEPTH ):
               trans_num = selectTransformation( codes, i )
               new_point = runTransformation( plist, index, codes[i][trans_num] )
               plistnew.append( new_point )
            plist = plistnew

         plotImage( plist, titles[i] )
         #pointlists.append( plist )
         #pointlists.append( [] )
         #del plist[:]
         #del plistnew[:]

      # fern and tree - need inital point seeded
      else:
         plist = [ (0,0) ]
         for j in range( MAX_POINTS ):
            index = selectPoint( plist )
            for iterations in range( DEPTH ):
               trans_num = selectTransformation( codes, i )
               new_point = runTransformation( plist, index, codes[i][trans_num] )
               plist.append(new_point)
         plotImage( plist, titles[i] )
         #pointlists.append( plist )
         #pointlists.append( [] )
      del plist[:]
      del plistnew[:]
```

Listing B.2: ch7_21_3D.py

```
##############################################################################
# Problem: 7.21, "Fundamentals of Natural Computing"
# Author: Stephanie Athow
# Date: 18 April 2015
# Problem Statement:
#  Implement the random midpoint displacement algorithm in 3D and generate
#  some fractal landscapes. Study the influence of H on the landscapes
#  generated.
##############################################################################

import numpy as np
import random
import scipy
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import griddata

NRC = 5         # number of recursion calls
sigma = 12      # standard deviation of the Gaussian distribution
mu = 5          # mean of Gaussian distribution
H = 0.75    # should be 0 < H < 1

##############################################################################
#                          Recursion
#  Recursively applies subdivision of the grid
##############################################################################
def Recursion( grid, delta, x0, x2, y0, y2, t, nrc ):
   x1 = ( x0 + x2 ) / 2
   y1 = ( y0 + y2 ) / 2

   neighbors = grid[x0][y0] + grid[x0][y2] + grid[x2][y0] + grid[x2][y2]
   val = 0.25 * neighbors + delta[t] * random.gauss( mu, sigma )
   grid[x1][y1] = val

   if (t < nrc):
      Recursion( grid, delta, x0, x1, y0, y1, t+1, nrc )
      Recursion( grid, delta, x0, x1, y1, y2, t+1, nrc )
      Recursion( grid, delta, x1, x2, y1, y2, t+1, nrc )
      Recursion( grid, delta, x1, x2, y0, y1, t+1, nrc )

##############################################################################
#                          Plot
##############################################################################
def plotImage( grid, delta, NRC, N ):
   X = []
   Y = []
   Z = []
   fig = plt.figure()
   ax = fig.add_subplot( 111, projection = '3d' )
   numrows = len( grid )
   numcols = len( grid[0] )
```

```
   for i in range( 0, numrows ):
      for j in range( 0, numcols ):
         x, y = i, j
         height = grid[i][j]
         X.append(x)
         Y.append(y)
         Z.append(height)

   ax.plot_trisurf( X, Y, Z)

   #plt.grid( True )
   plt.show()
   plt.savefig( 'brownian_surface.png' )

################################################################################
#                              Main
################################################################################
random.seed()
N = pow( 2, NRC )                          # number of points
grid = np.zeros( (N-1, N-1), dtype=float )      # holds grid
grid[ N-2, N-2 ] = sigma*random.gauss( mu, sigma ) # end point
delta = np.zeros( ( N-1, 1 ), dtype=float )        # holds variances

for i in range(0, NRC-1):
   delta[i] = sigma * pow( 0.5, (i+1)/2 )

Recursion( grid, delta, 0, N-2, 0, N-2, 0, NRC-1 )

plotImage( grid, delta, NRC, N )
```

Listing B.3: grayscott_mcgough.c

```
// Author: Dr. Jeff McGough
// Accessed: 29 April 2015
// Modifications:
//    f, k values to create different Gray-Scott patterns

#include <stdio.h>
#include <stdlib.h>
#include <time.h>


#define N 200
#define N1 201
#define Nh 100

int main()
{
    int i,j, count =0;
    double  f, k, e, dx, dxx, dt, d1, d2;
    double  diff1, diff2, nl;
    double  u[N1][N1], ub[N1][N1], v[N1][N1], vb[N1][N1];
    FILE *fp1, *fp2;

    f = 0.02;
    k = 0.05;

    e = 0.00001;
    dx = 2.5/(N-1);
    dxx = dx*dx;
    dt = 1.0;
    d1 = 2.0*e/dxx;
    d2 = e/dxx;
    count = 0;

    srand((unsigned) time(NULL));

    for(i=0; i< N1; i++)
    {
        for(j=0;j<N1;j++)
        {
            u[i][j] = 1.0 - 0.01*rand()/(RAND_MAX);
            v[i][j] = 0.0 + 0.05*rand()/(RAND_MAX);
            ub[i][j] = 1.0;
            vb[i][j] = 0.0;
        }
    }

    for(i=Nh-10; i< Nh+10; i++)
    {
        for(j=Nh-10;j<Nh+10;j++)
        {
            u[i][j] = 0.5;
            v[i][j] = 0.25;
```

```
        }
    }

    while (count < 10000)
    {
        count = count +1;

        for(i=1; i< N; i++)
        {
            for(j=1;j<N;j++)
            {

                diff1 = u[i-1][j]+u[i+1][j] + u[i][j-1] + u[i][j+1]  -4.0*u[i][j];
                diff2 = v[i-1][j]+v[i+1][j] + v[i][j-1] + v[i][j+1]  -4.0*v[i][j];
                nl = u[i][j]*v[i][j]*v[i][j];
                ub[i][j] = u[i][j] + dt*(d1*diff1 + f*(1.0 - u[i][j]) - nl);
                vb[i][j] = v[i][j] + dt*(d2*diff2 - (f+k)*v[i][j] + nl);

            }
        }
        for(i=1; i< N; i++)
        {
            for(j=1;j<N;j++)
            {
                u[i][j] = ub[i][j];
                v[i][j] = vb[i][j];
            }
        }
        for(i=1 ; i<N; i++){
            u[i][0] = ub[i][N-1];
            u[i][N] = ub[i][1];
            v[i][0] = vb[i][N-1];
            v[i][N] = vb[i][1];
            u[0][i] = ub[N-1][i];
            u[N][i] = ub[1][i];
            v[0][i] = vb[N-1][i];
            v[N][i] = vb[1][i];
        }
    }

    fp1 = fopen("gsuData.txt","w");
    fp2 = fopen("gsvData.txt","w");

    for(i=1; i< N; i++)
    {
        for(j=1;j<N;j++)
        {
            fprintf(fp1, "%lf  ",u[i][j]);
            fprintf(fp2, "%lf  ",v[i][j]);

        }
        fprintf(fp1,"\n");
        fprintf(fp2,"\n");
    }
```

```
    fclose(fp1);
    fclose(fp2);
    puts("To plot Datafile using Gnuplot:");
    puts("set contour");
    puts("unset surface");
    puts("unset ztics");
    puts("unset zlabel");
    puts("set view map");
    puts("splot \"gsuData.txt\" matrix with lines\n");


    return 0;

}
```

Listing B.4: heat_flow.py

```
# Author: Stephanie Athow
# Date: 21 April 2015
# Problem:
#  Modify the heat flow example to deal with insulated conditions on the top
#  and bottom boundary. Insulation means zero flux or:
#     u[N][j] = u[N-1][j]
#  This implies that instead of a fixed valued ghost points on the top and
#  bottom, you modify the CA rule using the previous relation.

import numpy as np
import matplotlib.pyplot as plt

N = 100            # grid size (square)
time_end = 1500       # time step end

t_source = 10
t_grid = np.zeros( [N, N] )   # holds grid of temps
t_update = np.zeros( [N, N] ) # holds update values

# initialize heat source
for i in range( 0, N ):
   t_grid[i][N-1] = i*( N-1-i )

# run CA on grid for time_end loops
for i in range( 1, time_end ):

   # update top/bottom boundaries
   for k in range( 0, N ):
      t_update[0][k] = t_grid[1][k]
   for k in range( 0, N ):
      t_update[N-1][k] = t_grid[N-2][k]

   # update inside cells
   # loop rows
   for j in range( 1, N-1 ):
      # loop columns
      for k in range( 1, N ):
         if( k == N-1 ):
            t_update[j][k] = t_grid[j][k]

         else:
            neighbors = t_grid[j-1][k] + t_grid[j+1][k] + t_grid[j][k-1] + t_grid[j][k
            update_temp = neighbors / 4.0
            t_update[j][k] = update_temp


   # time update t_grid
   #del t_grid
   t_grid = t_update

fig, ax = plt.subplots()
ax.imshow( t_grid, cmap=plt.cm.gray, interpolation='nearest' )
```

```
plt.show()
```