CSC 492/592 BioInspired Computing

Homework 3

Mack Smith Derek Stotz

May 4, 2015

Contents

List of Figures

Document Preparation and Updates

Current Version [1.0.0]

Prepared By:
Mack Smith
Derek Stoz

Revision History

Date	***Author***	***Version***	***Comments***
5/2/15	*Mack Smith and Derek Stoz*	*1.0.0*	*Finished all programs and some portions of documentation*
5/4/15	*Mack Smith and Derek Stoz*	*1.0.0*	*Wrote the rest of the documentation*

1

Fractals - Text Chapter 7

1.1 Problem 7.10

1.1.1 Problem Information

Write a python program to use turtle graphics to draw the fractals in figure 7.24. This incorporates the bracketed OL-system described in chapter 7 to draw fractals. After recreating the fractals from the book, I came up with my own set of production rules to draw even more fractals. Unfortunately the ones I came up with were not as cool as the ones in the book.

1.1.2 Implementation

I wrote this program in python and used the turtle graphics module to do the fractal drawing. The functions I wrote for this program were the generate production rules function, and the draw fractal function. To generate the production rules I simply iterated through the initial string and replaced everything corresponding to the F-rule and G-rule. I generated the entire production rules string before I drew the fractal. Then to draw the fractal I passed in the production rules string as well as the rotate value and the draw length.

1.1.3 Issues

When tasked with coming up with my own production rules, most of the ones I came up with turned out to look like tumbleweeds. I don't know why most of them did that, but some others that I produced were very stick like. It wasn't a problem with my program, just my own lack of creativity.

1.1.4 Analysis

The code to produce the fractal rules and to draw the fractal were very simple and didn't require too much work. I made functions to call each of the different fractals with each of their own rule sets. The fractals themselves were fun to watch for a while but ultimately they took way too long to draw so I ended up getting bored and doing something else while they were drawing.

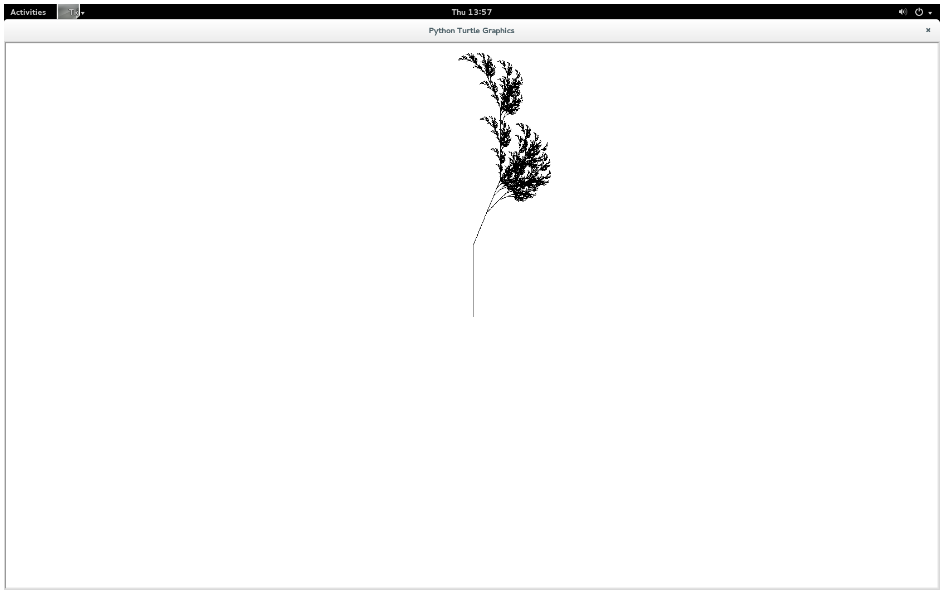

Figure 1.1: Fractal 1

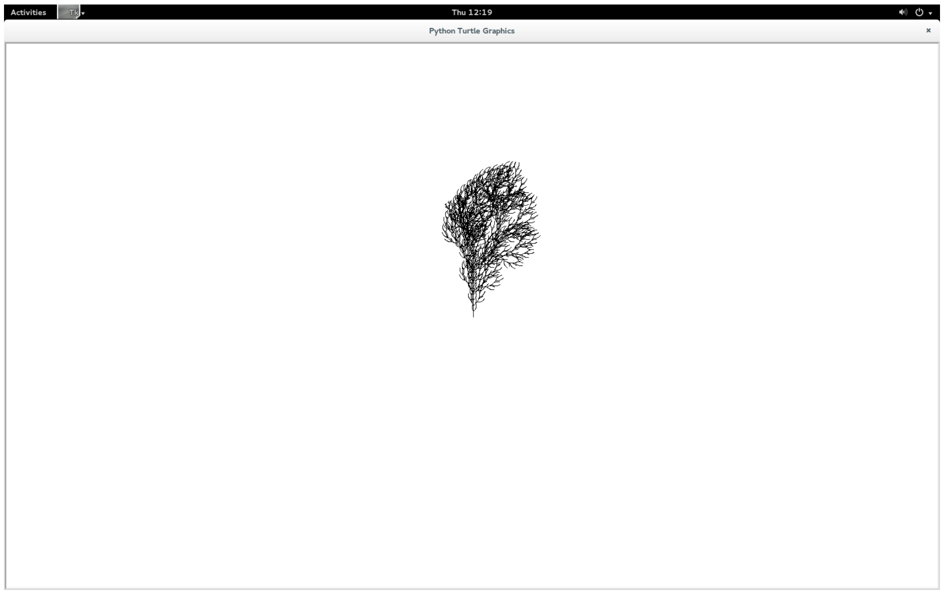

Figure 1.2: Fractal 2

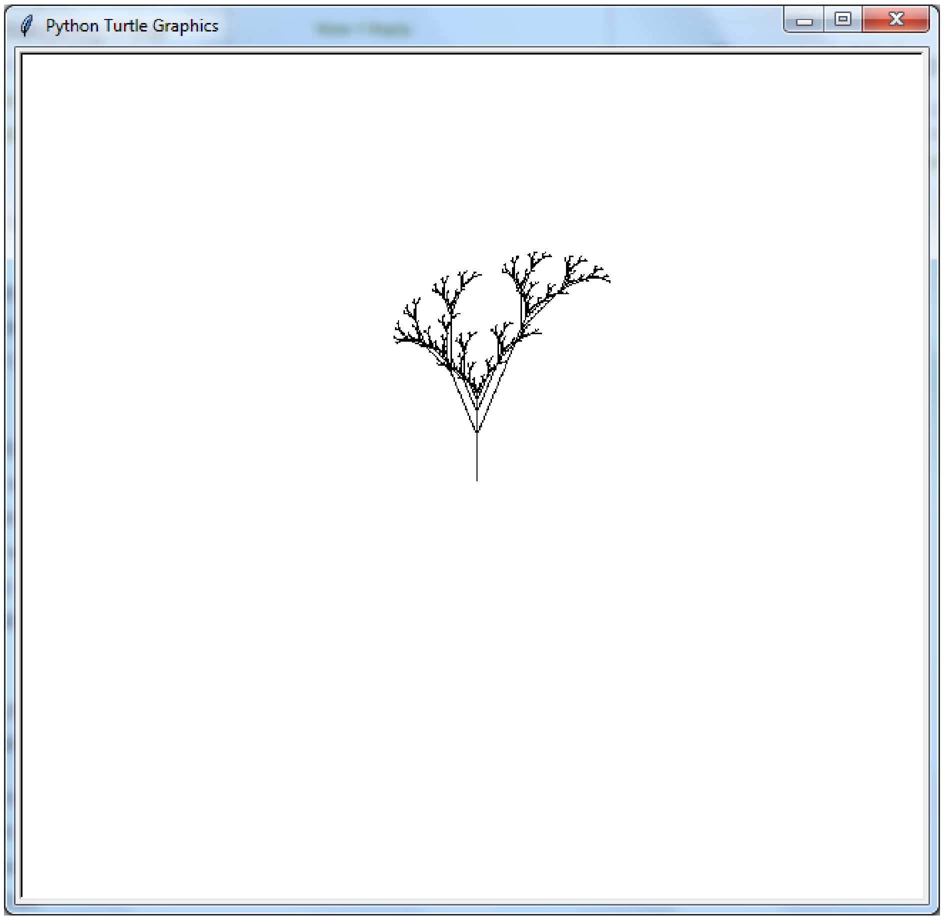

Figure 1.3: Fractal 3

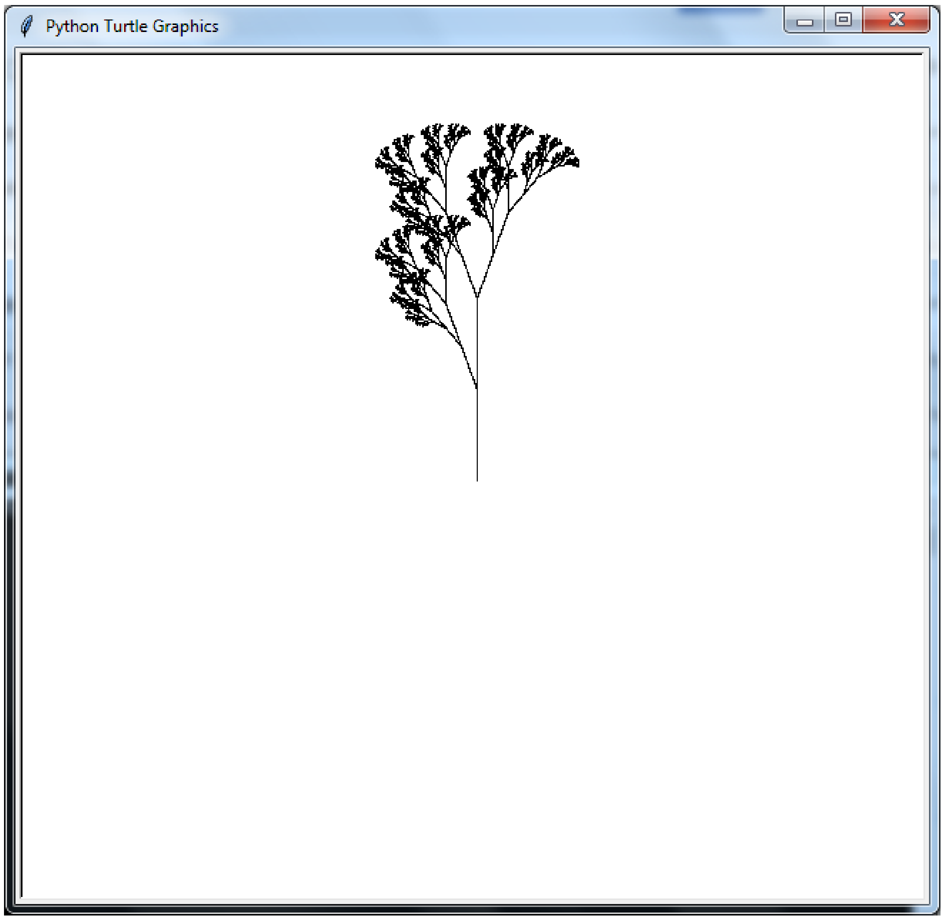

Figure 1.4: Fractal 4

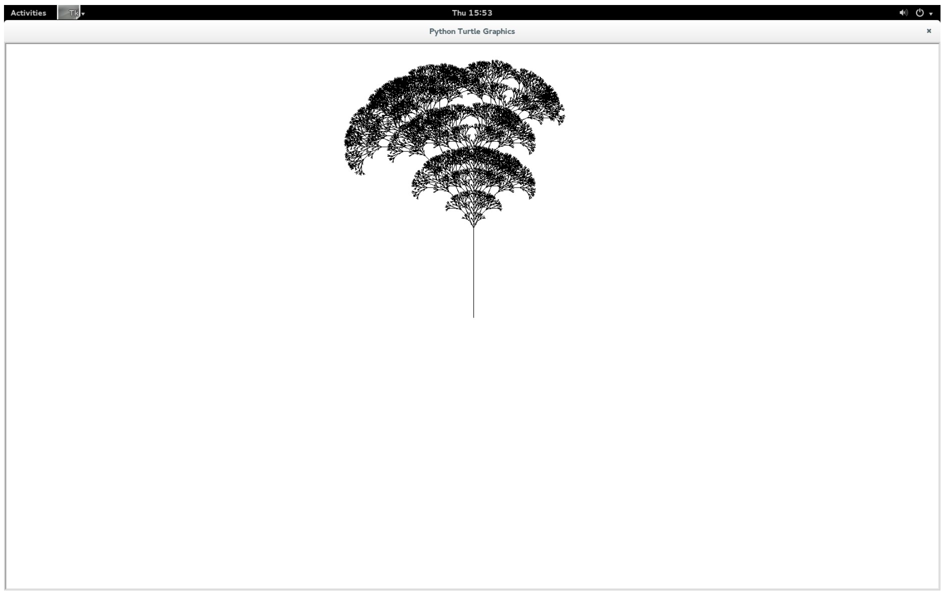

Figure 1.5: Fractal 6

1.2 Problem 7.15

Implement a RIFS to generate all the fractals whose codes are presented in Table 7.3

1.2.1 Problem Information

RIFS - Random Iterated Function Systems - are a method of sequentially drawing fractals which give varying results. Each iteration has a certain chance of making an affine transform on the current pixel and then updating the pixel to that transformed one.

Although RIFS systems are quite small code-wise, they can generate some very interesting shapes through emergent behaviour. Table 7.3 presented us with some classic fractals: The Serpinski Gasket, the Bernoulli Fern, The Square and The Tree. There was an error in the gasket's d column values - they should all be .5.

One major difference between the RIFS and IFS fractal generation systems is that while the IFS uses subimages or drawing strokes, the RIFS just uses individual pixel values. This makes the turtle a poor fit for output, and so instead we used pyplot's scatter graph. The results were simply lovely.

1.2.2 Algorithm Description

The RIFS algorithm we implemented came straight out of the book. The only real addition - other than the display functionality - was some user-friendly file parsing. The first three values in the input file are the start point coordinates of the fractal followed by the point size. The larger the point size, the denser the fractal is drawn.

The fractal building took the initial point, and then selected an affine transform loaded from the table file via roulette wheel selection. The p column in the file - the last column, that is - specified the chance that any given transform would be applied.

After an affine transform was randomly picked, so that the selected column j was chosen, a function was called which pulled the table values at that specified j into some transformation matrices, which were then multiplied with the current point. The resultant point was returned and then drawn to the screen, and the loop restarted. It ended when the maximum number of iterations was reached.

1.2.3 Results

Despite using such a simple, straightforward implementation, our results were exceedingly good. A great number of iterations to use is actually 1000000. A percentage is updated on the console as the fractal is built, and then the fractal will take a while to draw. The result is a nicely drawn fractal in a proportionally square plot.

1.3 Problem 7.21

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

1.3.1 Problem Information

The midpoint displacement algorithm is a common method of generating fractal landscapes in 2D and 3D. The book highlighted how to do it in 2D and gave some very explicit pseudocode for how to generate the landscapes with the algorithm. It was up to us to extend it to 3D.

This problem required some drawn 3D output and a study of the input parameters, which were the number of divisions and the maximum deviation per iteration. We shall refer to them as nmd and theta, respectfully. The pyplot 3D axis object was used for generating the 3D landscapes, and the command line arguments specified the two input parameters. Pyplot allowed the view to be rotated with the mouse.

1.3.2 Algorithm Description

Essentially, the algorithm has two parts: an iterative portion and a recursive portion. The recursive portion builds a list of deltas - deviations - which depend on the division number and sigma. They are stored for later and passed to the recursive function. The recursive portion also requires a start and end point and number of divisions, along with the current division number. It is called immediately after the delta list is built.

The recursive portion divides the distance between the two points in two and then sets its height to a random perturbation times the current division's detla value, which was calculated beforehand. After the division, it returns if the current division is equal to nmd. Else, it calls the next division on the midpoint between the two x values, the midpoint between the two y values, and the midpoint between the x and y values. The result is a random landscape generally grading in one direction.

1.3.3 Results and Analysis

We are not terribly happy with the ugly results, although the algorithm appears to be working correctly. Due to the poor fidelity of the 3D pyplot, it is very difficult to see much of anything unless large theta values are picked, and while the landscape is visible on the grid, the axis do not appear connected in the z-dimension, which is annoying.

An analysis on the input parameters is pretty basic. When nmd is increased, the hills and vallies become smaller in radius and more numerous on the grid. it also becomes exponentially slower, which makes sense.

When theta is increased, the hills and vallies become more pronounced. If theta is too small, then the hills and vallies actually become invisible. Due to the the nature of the algorithm, half of the hills and vallies are not visible at all unless the sigma is cranked up to a very high value.

As the code file says, a good trial run uses a width of 10, an nmd of 4 and a theta of 10.

2

Slides - Heat Flow

2.1 Problem

Implement the heat flow diagram in the text using an insulated top and bottom layer.

2.1.1 Implementation

Using python and the matplotlib module, I used Dr. McGough's code from the slides to implement the diagram. Working in the OPP lab however the computers there did not have the matplotlib installed, so I had to resort to using Python 2 rather than Python 3 which didn't appear to affect the final result much at all, if any.

2.1.2

Like I said, the computers in the linux lab did not have matplotlib installed so Python 3 would not run the code. Python 2 was used instead to fix this.

2.1.3 Analysis

Aside from the Python 2, Python 3 issues, this program was very easy to implement mostly because all of the code was from the slides, except for the two if statements I added for the insulation. Overall it turned out well and illustrated the heat flow accurately.

3

Slides - Gray Scott

3.1 Problem

Generate the lambda, theta, alpha, and mu Gray-Scott patterns described in the slides. These patterns are reminiscent of those found in nature, such as animal coat patterns as well as bacteria colonies.

3.1.1 Solution

Using the code provided by Dr. McGough, the Gray-Scott program was implemented in C due to performance issues with the python code. The C code used to run this program is honestly confusing as hell to me... it just looks like a bunch of random calculations. However, the data is then represented in a plot using Gnuplot.

3.1.2 Issues

Narrowing down the exact number for the F and K parameters was difficult. This resulted in some not exact matches to the slides in some cases, and in others it didn't resemble the slides at all.

3.1.3 Analysis

The patterns generated were not exact matches to the given templates. It was hard to get the right values for F and K which is what caused the differences. It was pretty amazing to see how little changes affected the final product in a huge way.

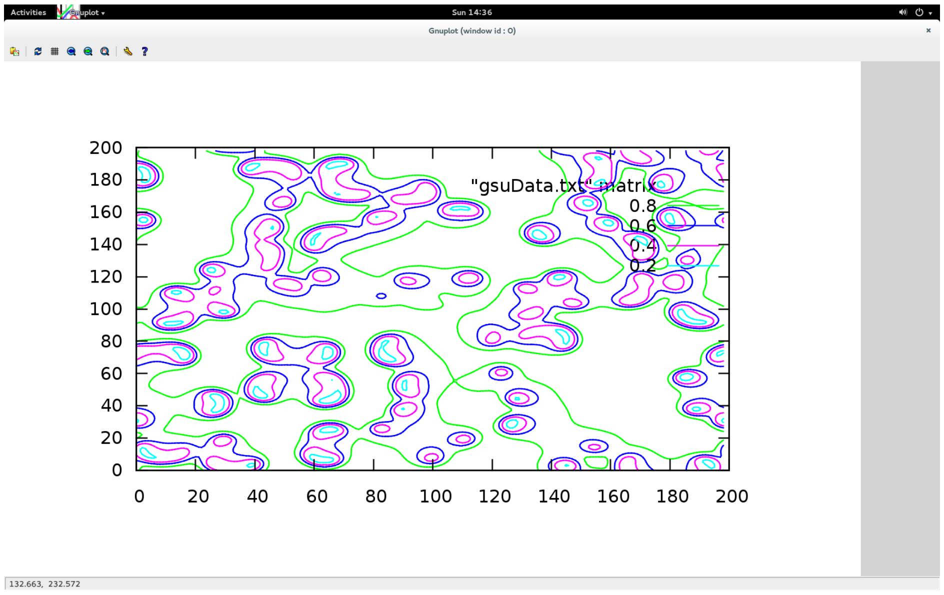

Figure 3.1: Alpha

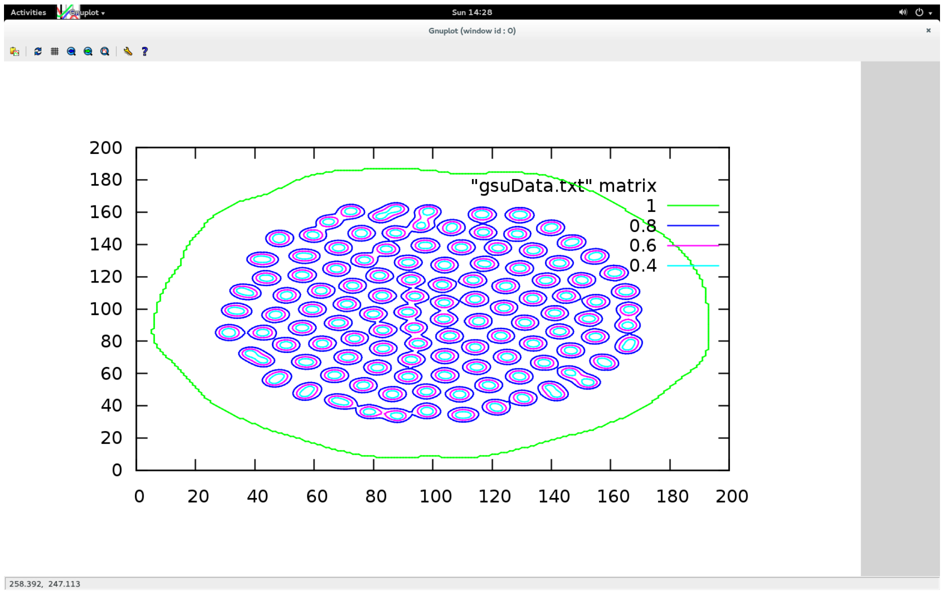

Figure 3.2: Lambda

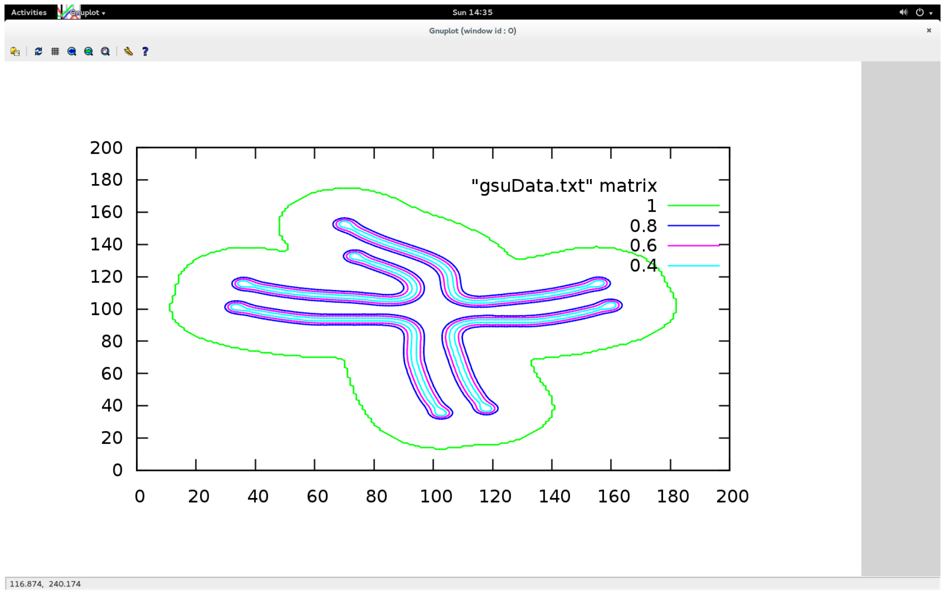

Figure 3.3: Mu

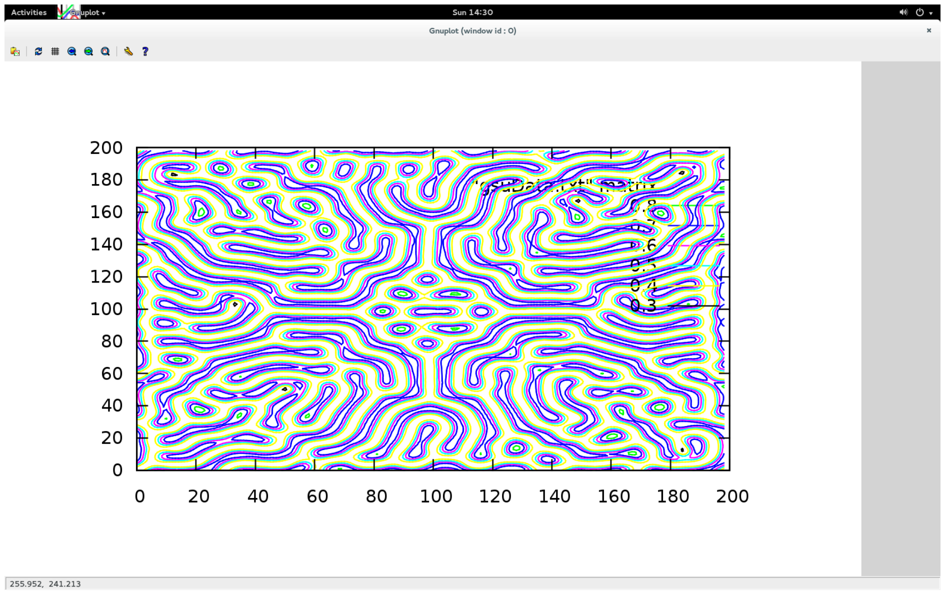

Figure 3.4: Theta

4

Cellular Automata - Text Chapter 8

4.1 Problem 8.3

Implement one of the Problems on the StarLogo website – we chose to do the rabbit and grass simulation.

4.1.1 Problem Information

The rabbit and grass simulation proposes a plane with randomly scattered grass and rabbits. Over time, the patches without grass grow grass and the rabbits move around, eating the grass off of the grassy patches. As the rabbits consume the grass, they gain energy. Once they reach a sufficient energy level, they bud off a second rabbit, which goes off on its own and eatis and reproduces by itself.

The Rabbits all move randomly and lose a slight amount of energy with every movement. If they run out of energy, they die.

There are a handful of parameters which can be passed into the simulation. The hatch threshold is the amount of energy required for reproduction in a single rabbit. The starvation rate is the amount of energy that bunnies lose at every movement. The grass growth rate is the chance that grass will grow in any given square in any given time step. The maximum iterations is the maximum number of steps that the simulation will run.

4.1.2 Algorithm Description

We created a Bunny class, which held the position and energy of the bunnies. In the time step, the grass matrix was updated by growing grass in each tile with a certain probability. However, before the grass was grown, the bunnies were iterated through. Their various bodily functions and movements were processed and their energy was modified, and then at the end of each bunny's iteration, it would be marked as dead if its energy

was below zero. At the end of the time step, all bunnies which were marked as dead were removed. At the end of each step, the pyplot was updated with the grass and grassless spots, drawing red spots where there were bunnies.

4.1.3 Results

This was simple and fun, and different settings yielded very different results. Setting the starvation rate to something very low - like .05 - and setting the growth rate to something also very low - like .001 - give a good example of a fluctuating population. The default parameters give a fairly generic example. If the growth rate was too high in relation to the starvation rate, then the bunnies would exponentially overpopulate. If the converse was true, then the rabbits would go extinct. Decreasing the hatch threshold speeds up the whole simulation.

4.2 Problem 8.4

Implement Conway's Game of Life

4.2.1 Problem Information

Conway's Game of Life is a well-known cellular automata simulation of which Derek was already familiar. This made testing and verification easy, although the main challenge was in implementing a fast solution.

Conway's Game of Life is a simulation with four simple rules. Firstly, any cell with more than three live neighbors dies of overcrowding. Secondly, any cell with exactly three neighbors comes to life. Thirdly, any cell with less than two neighbors dies of starvation. Lastly, any live cell with two or three neighbors lives on to the next generation.

4.2.2 Algorithm Description

We implemented a grid which was drawn by pyplot, setting live cells to black and dead cells to white. Following each of those rules iteratively, we updated the cell grid each generation.

The number of neighbors for each cell was calculated by convolving the grid with a 3x3 mask.

4.2.3 Results

The result was an adjustable game of life accurate to the specifications. The first argument should be start.txt, the second should be the generations/second - 10 is usually good. The third argument should be the maximum number of generations, which is entirely up to the user. Entering a very high number is advised. The starting configuration file is structured so the first line has the height and width of the playing field Subsequent lines are rows in the playing field. 0s are cells which are dead, and 1s are cells which are alive.

5

Immunocomputing - Text Chapter 6

5.1 Problem 9.1

Problems that a turing machine cannot solve:

1. Halting problem
2. Empty tape acceptance problem
3. Empty set acceptance problem
4. Regular machine recognition problem

5.2 Problem 9.2

Name four NP-Complete and four NP-Hard problems:

5.2.1 NP-Complete

1. Knapsack Problem
2. Traveling Salesman problem
3. Vertex cover problem
4. Hamiltonian Path problem

Disclaimer: the problem stated to NAME the problems, not describe them

5.2.2 NP-Hard

1. Flow shop scheduling
2. K-minimum spanning tree
3. Nurse scheduling problem
4. Quadratic assignment problem

Disclaimer: the problem stated to NAME the problems, not describe them

5.3 Problem 9.3

Maxam-Gilbert vs Sanger DNA sequencing methods

1. Maxam-Gilbert: The DNA strands are subject to a chemical treatment that fractures it in 4 points where different reactions occur (G, A+G, C, C+T). They are then placed in a sequencing gel where each segment can be visible and the sequence can be inferred.

2. Sanger:

   This method copies the DNA strand to be sequenced using four chemically altered bases. Each of the bases stops when it encounters a specific letter associated with the DNA proteins. After all of the copying is done there will be four strands each with one DNA letter on them. They are then put backed together like a puzzle to give the sequence of the original strand of DNA

3. Contrast: While the Maxam-Gilbert method was used for a long time, it is very time consuming and prone to human error when examining the sequence gel. The Sanger method is much more refined and can produce a more accurate result because of the way it is structured.