COSC 420: BIOLOGICALLY-INSPIRED COMPUTATION

PROJECT 3 - "HOPFIELD NETWORK"

CLARA NGUYEN

TABLE OF CONTENTS

1. SYNOPSIS

In this project, I explored how Hopfield Networks work, and go over how it works based on the experiments run on it. In most of the simulations done here, the Hopfield network consists of 50 patterns that have 100 neurons in each pattern. The goal is to get information by running the simulator multiple times. Then the data of each simulation is averaged. Most simulations are run 50 times before being averaged. In other cases (to determine different behaviour), the simulator may be run more times, upwards to absurd values.

2. DEVELOPMENT

The simulator was written in C++. The simulator was written with an object-oriented aspect in mind. As such, an entire object was created for a single run of a simulation. The object is configurable and expandable, which allows us to tweak it to get more interesting results for researching how Hopfield Networks work. The files are stored in the "src" directory. However, the makefile is in the root directory of the project. To compile, simply type "make" in the root directory of the project, and the makefile will go into the "src" directory and compile the executable itself. To run the simulator, supply the following arguments:

```
./hopfield patterns neurons simulations
```

For example, the default configuration consists of 50 patterns, 100 neurons, and 50 simulations. Therefore, the following must be passed into the program:

```
./hopfield 50 100 50
```

The program will output a valid CSV file containing information about the "p", the "Fraction of Unstable Imprints", and the "Stable Imprints". Pipe the information into a file, and then you can generate a graph with an external application such as Microsoft Excel.

3. HOW IT WORKS

```
initialize data structures for keeping statistics
for i from 0 to number of runs
  generatePatterns()
  for p from 1 to 50
    imprintPatterns(p)
    testPatterns(p)
compute averages over number of runs
normalize data for basins of attraction (527 students)
write data file
```

Figure 1. *Pseudo Code for the Simulator.*

The program's structure was written with pseudo code from Figure 1 in mind. The program abides by a few mathematical equations that were given to us in a lab writeup. It sets up a number of patterns, that consist of multiple numbers of neurons. Initially, the states of the neurons are set to be either -1 or 1 randomly.

3.1. IMPRINTING PATTERNS

The imprintPatterns function from the pseudo code is where the weights of neurons are computed. It will store the weights in a 2D array of equal width and height. The size of the 2D array is determined by the number of neurons in a pattern (e.g., a pattern with 100 neurons will have a 100x100 array of weights). The equation in Figure 2 will determine the weight at any point in a specific pattern *p*. The weight of *i* and *j* only have a value if they are not the same, as the weight of a neuron to itself is 0.

$$w_{ij} = \begin{cases} \frac{1}{N}\sum_{k=1}^{p} s_i s_j, & i \neq j \\ 0, & i = j \end{cases}$$

Figure 2. *Equation for computing weights at i and j in a specific pattern p.*

3.2. TESTING PATTERNS FOR STABILITY

The testPatterns function from the pseudo code is where the simulator will test whether or not a pattern of neurons is stable or not. The procedure for doing this involves computing *h*, which requires weights of each neuron. The equation in Figure 3 shows how to compute h_i, which is used in the formula featured in Figure 4 to generate a new state. The function will do this for every neuron in the pattern. If *any* neuron does not match its new state, the pattern is considered to be unstable. Otherwise, it is stable.

$$h_i = \sum_{j=1}^{N} w_{ij} s_j$$

Figure 3. *Equation for computing* h_i.

$$s'_i = \sigma(h_i)$$

$$\sigma(h_i) = \begin{cases} -1, & h_i < 0 \\ +1, & h_i \geq 0 \end{cases}$$

Figure 4. *Equation for generating the new state for stability comparison.*

The simulator takes all of the values of *h*, sums them up, and uses that to tell whether or not the entire pattern is stable.

4. EXPERIMENTS (UNDERGRAD PORTION)

The program is configurable to allow variety in testing. We can do the following:

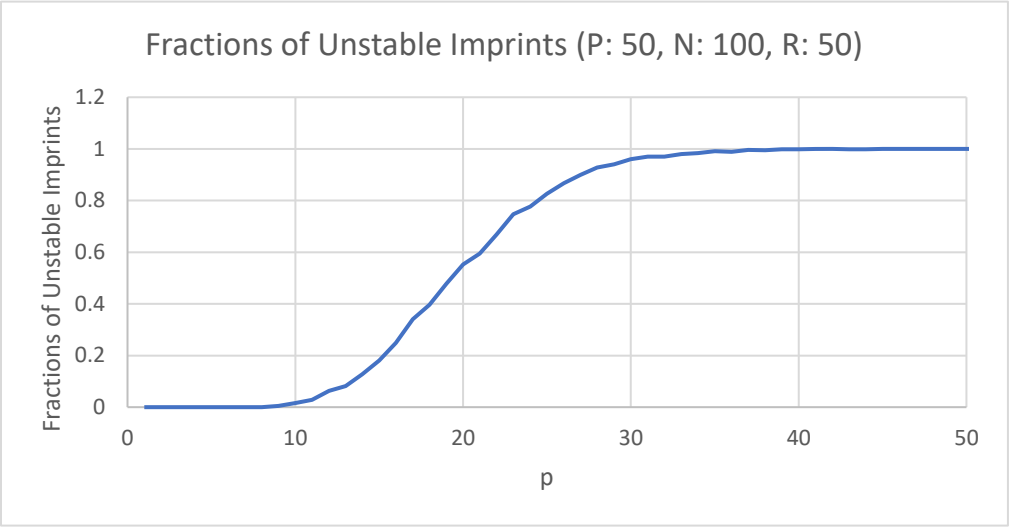
- Configure the number of neurons in each pattern
- Configure the number of patterns in the simulation
- Configure the number of times the test is run and averaged.

The experiments conducted put the simulator under various stress levels ranging from very

lightweight to very brutal. As such, I have labelled the experiments by number, specifying the configuration used in the simulator where **P** is the number of patterns, **N** is the number of neurons in each pattern, and **R** is the number of times the program repeated the simulation.

4.1. EXPERIMENT 1 (P: 50, N: 100, R: 50)

This experiment is the default configuration of the simulator and is the one used in the Project writeup.



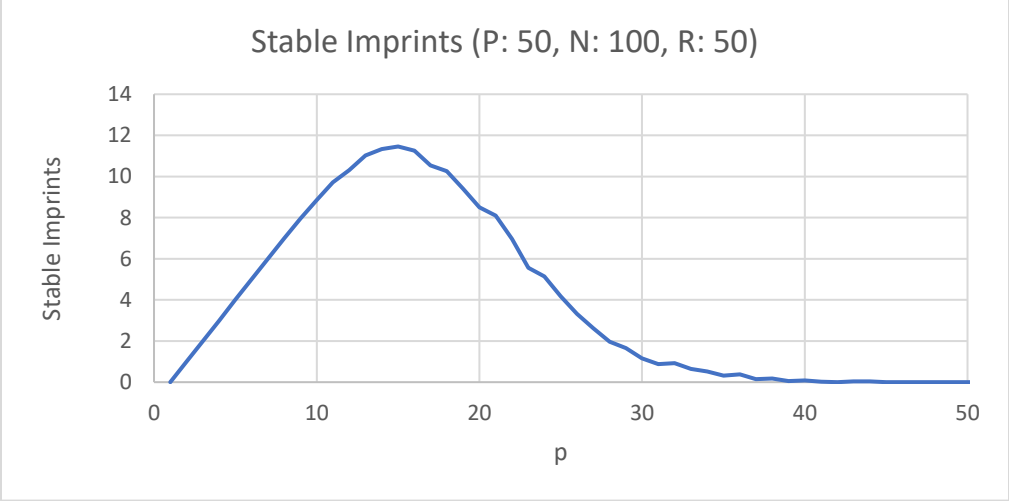
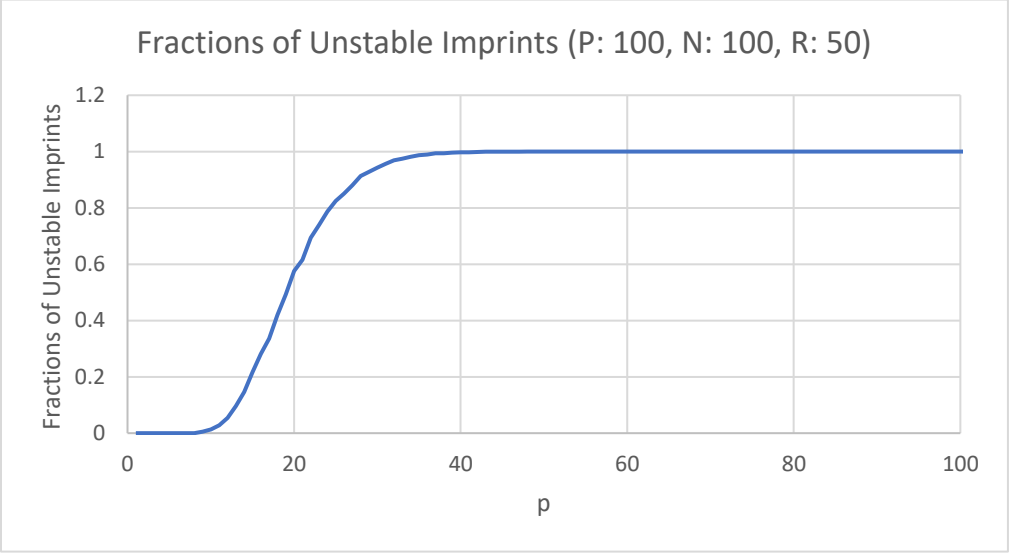


Figure 5 (Top) and 6 (Bottom). *Results of Experiment 1.*

This is an expected result, matching the results shown in the project writeup.

4.2. EXPERIMENT 2 (P: 100, N: 100, R: 50)

After Experiment 1, I decided to experiment with the values to see what happens when we have more patterns. For this experiment, we will simply double the number of patterns.



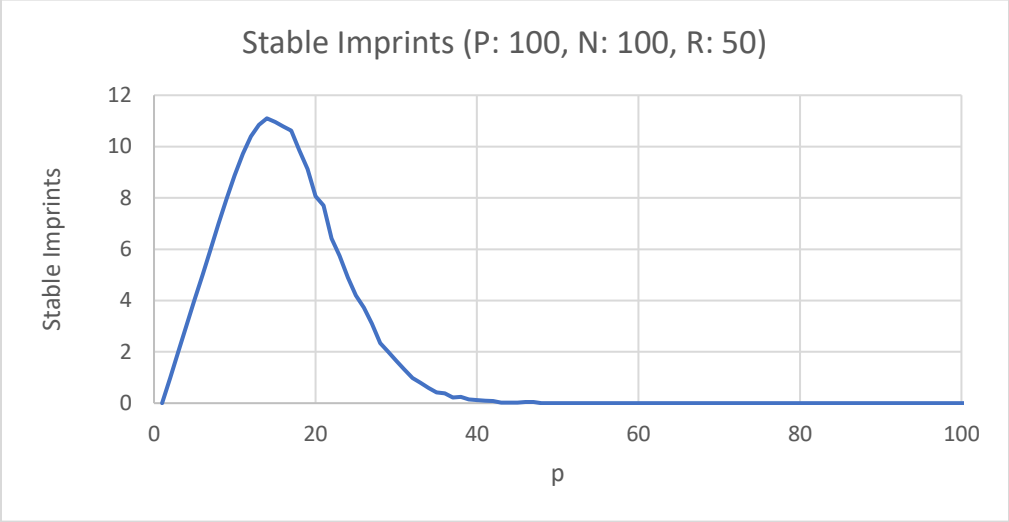
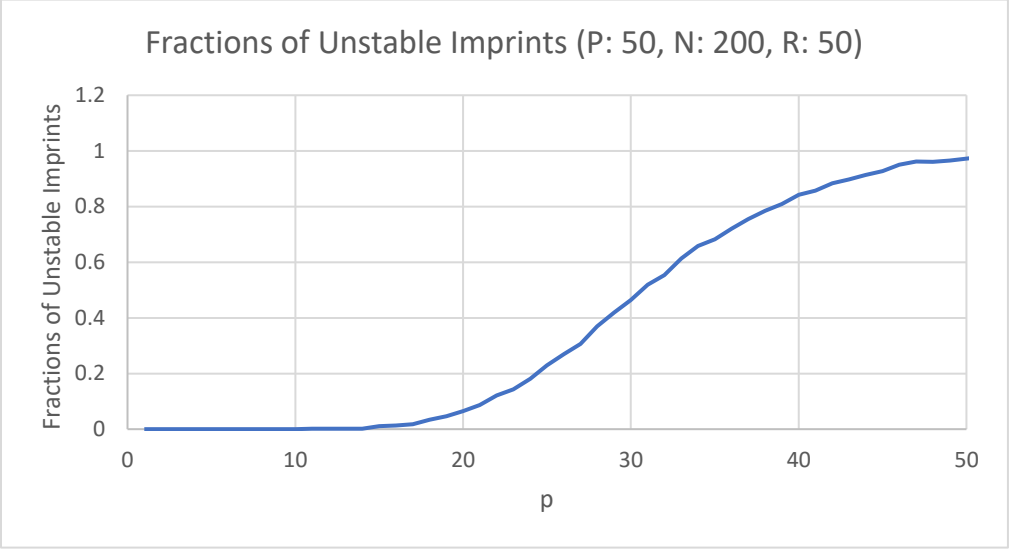


Figure 7 (Top) and 8 (Bottom). *Results of Experiment 2.*

This experiment ended up showing us that no data beyond around the same number of patterns as shown in Experiment 1. This is an expected result, as p goes beyond 30-35, the fraction of unstable imprints becomes 1 and mostly stays there throughout the remainder of the simulation. As for the Stable Imprints, it goes down to 0 when p is 46 and stays there for the remainder of the simulation.

4.3. EXPERIMENT 3 (P: 50, N: 200, R: 50)

With the same mentality as in Experiment 2, I wanted to see how the simulator performs with twice as many neurons as the default setup in Experiment 1.



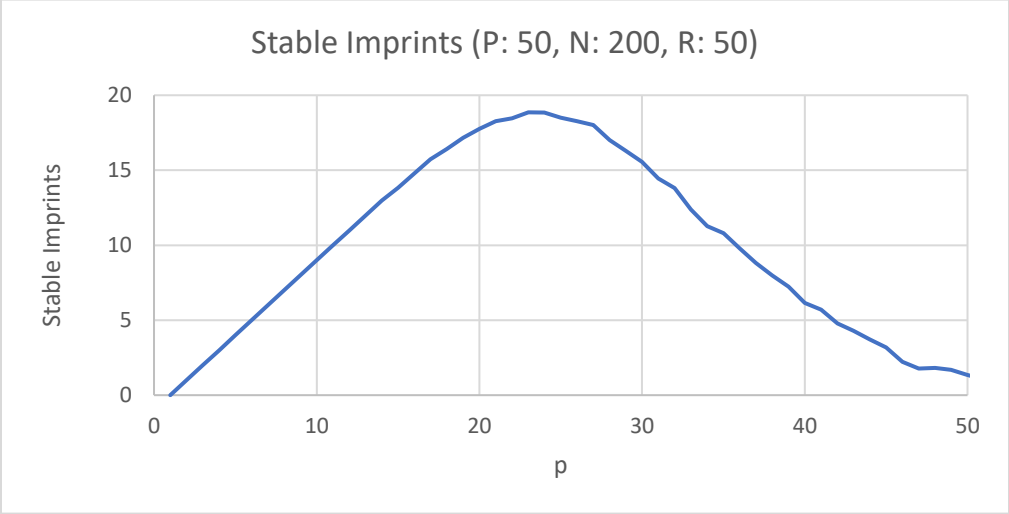
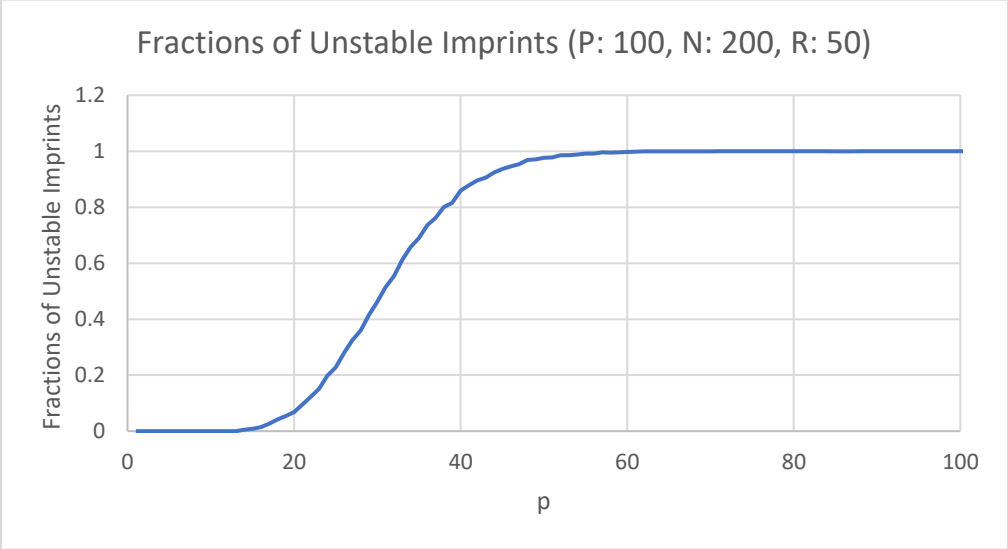


Figure 9 (Top) and 10 (Bottom). *Results of Experiment 3.*

Interestingly enough, double the neurons means that the graphs scale horizontally in proportion to the number of neurons added in. This is close to a 1-to-1 ratio. However, as we try more patterns and neurons, we will find that the slight offset in the ratio will cause the results to scale a bit less horizontally on the graph as the number of patterns grow.

4.4. EXPERIMENT 4 (P: 100, N: 200, R: 50)

With the results from Experiment 1, 2, and 3 in mind, we can take it a step further and multiply both the patterns and the neurons and see the results.



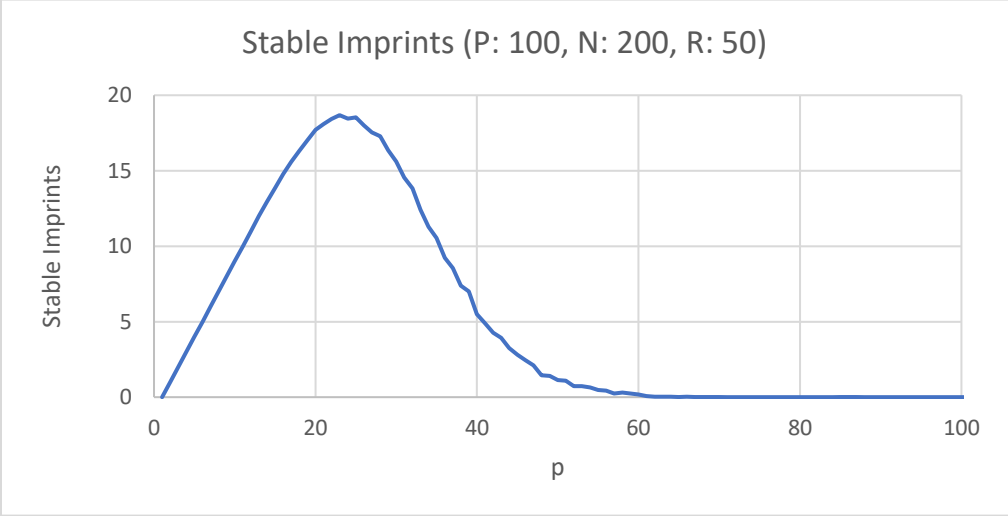


Figure 11 (Top) and 12 (Bottom). *Results of Experiment 4.*

As expected, the result looks almost the same as Experiment 1, except all of the values have nearly doubled. Though, the ratio isn't exactly 1-to-1. The peak is around 18 in Stable Imprints, meanwhile Experiment 1's peak is at around 11.6. We can further expand on this and get further data by increasing the numbers even more.

4.5. EXPERIMENT 5 (P:1000, N: 2000, R: 50)

To confirm the theory that the data scales almost linearly with the number of patterns and neurons, we can take the number of patterns and neurons a step further by multiplying them by 10.

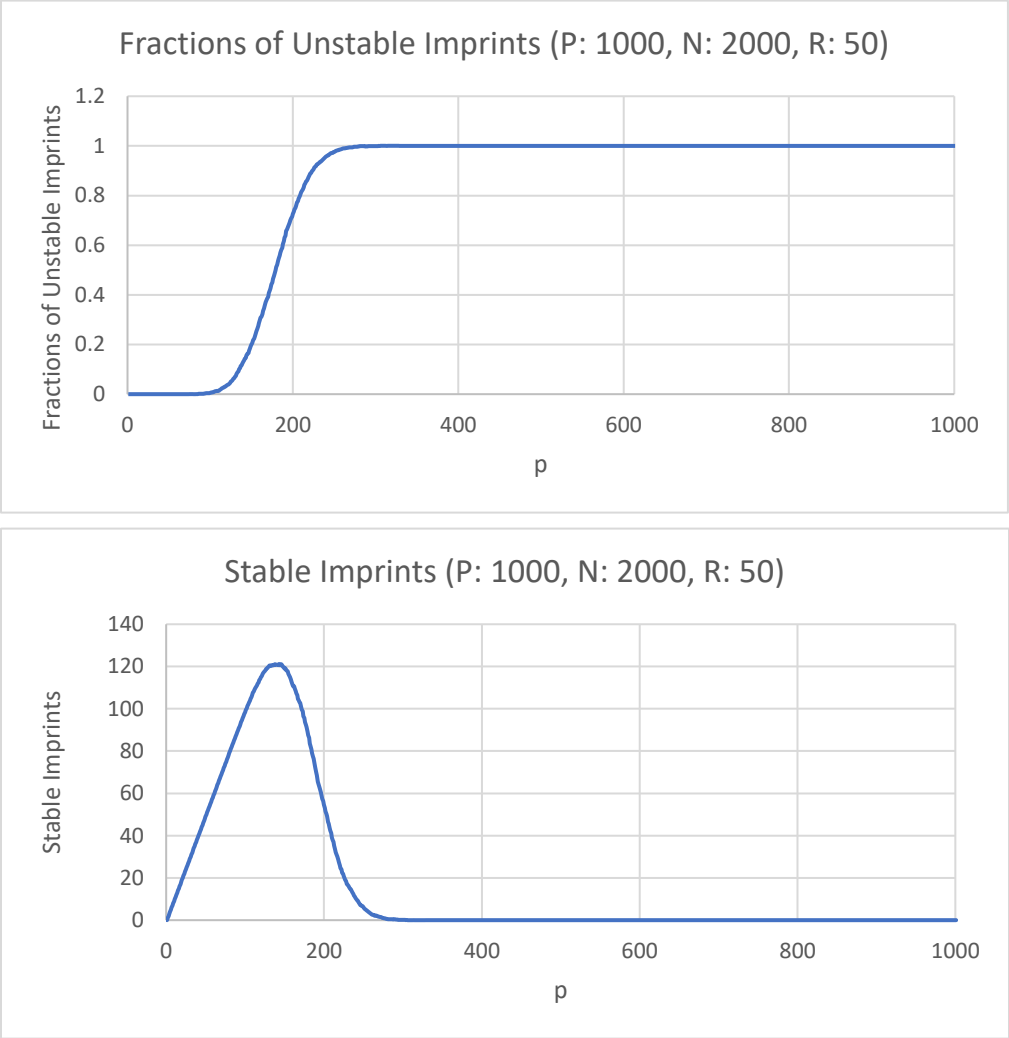


Figure 13 (Top) and 14 (Bottom). *Results of Experiment 5.*

This test was not an easy one to run. When single-threaded and compiled under "-O3", it took around 5 hours to finish. I rewrite the simulator with a parallel programming library known as OpenMPI in order to make the test run much faster and span multiple machines. With OpenMPI, it took under 45 minutes.

4.6. EXPERIMENT 6 (P: 5000, N: 10000, R: 50)

4.6.1. THE PROBLEM

This brutal test put the simulation to the limit in an effort of showing how accurate the data we are receiving is. The system requirements to even run this test required around 1.5865 GB of memory per machine (79.325 GB of RAM total), so I resorted to my OpenMPI implementation that I made specifically for Experiment 5. It allowed me split the job up into all of the CPU cores on the machine. However, that would only give me a 4-8x speed boost in computation. How can we speed this up? It turns out, we can take this level of multithreading to an entirely new level by gathering other machines in a cluster to help perform the computation.

4.6.2. THE SOLUTION

Nuke Hydra.

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
28128	ssmit285	39	19	541948	202644	3792	R	100.3	2.5	1164:35	hopfield_mpi
28129	ssmit285	39	19	541948	202632	3792	R	100.3	2.5	1165:15	hopfield_mpi
28130	ssmit285	39	19	541948	202612	3788	R	100.3	2.5	1164:35	hopfield_mpi
28131	ssmit285	39	19	541948	202640	3784	R	100.3	2.5	1164:35	hopfield_mpi

Figure 15. *Output from "top", showing the simulator running 4 threads on one machine.*

In a desperate need of computational power, I resorted to using all 31 of the Hydra and Tesla computers (62 total) and unleashed 4 threads on each. This totaled to be 248 threads all computing values for the simulation. This was to account for if any of the machines shut down mid-simulation. 4 of the Tesla machines shut off mid-simulation. However, I only needed 50 of them machines to complete the simulation so I was good.

On a single core of a single machine, the process would have taken around half a month to run this simulation. With OpenMPI and 62 computers, it took around 13 hours to compute.

4.6.3. The Experiment

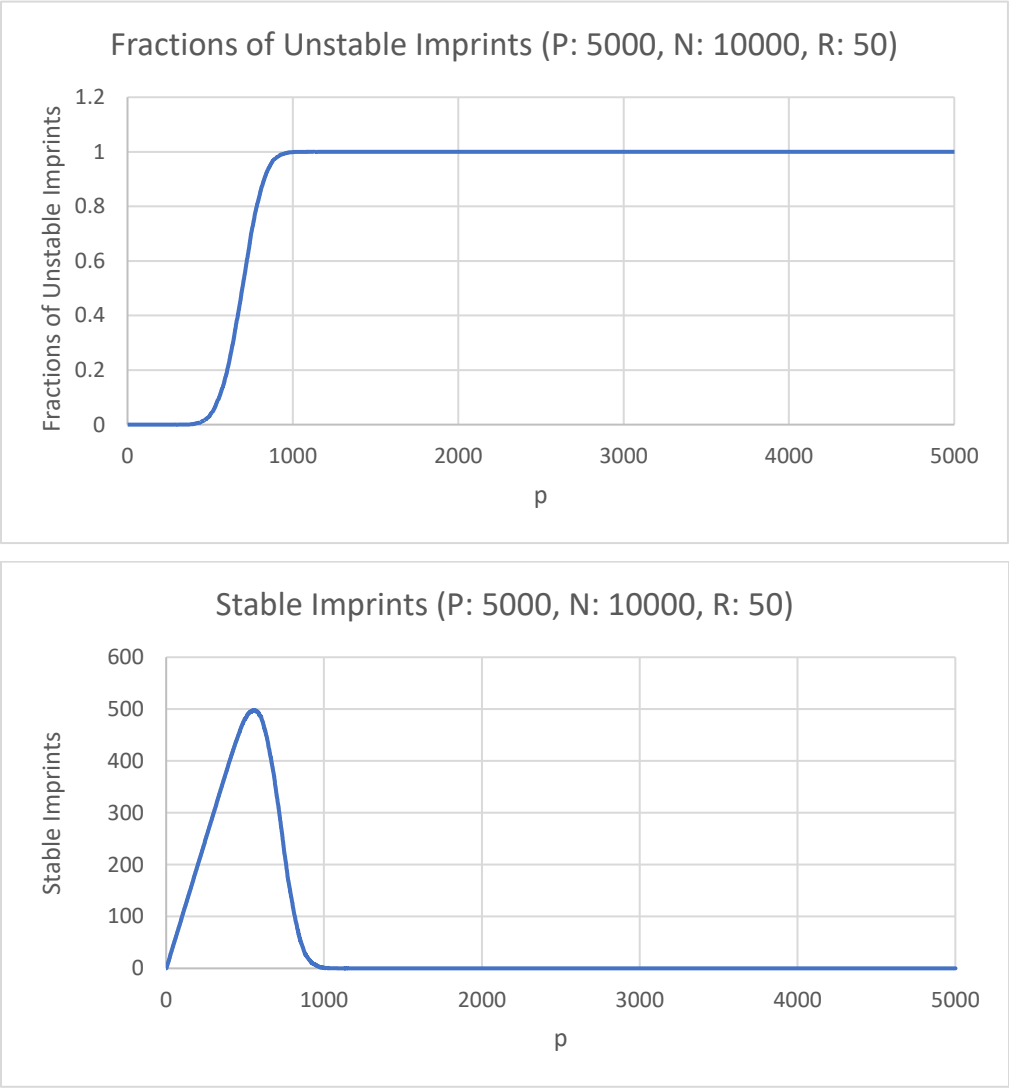


Figure 16 (Top) and 17 (Bottom). *Results of Experiment 6.*

The graph slope is getting more steep as we increment the number of patterns. It is really becoming apparent as we keep multiplying the neurons and patterns. The results of this experiment, despite the amount of time it took to compute the values, is unsurprising. Though the data generated is extremely precise, as we iterated through 100x the original guidelines stated for the Hopfield Network.

5. OBSERVATIONS (UNDERGRAD PORTION)

5.1. GENERAL

After a certain number of simulations, the fraction of unstable imprints will *always* go to 1. The only times that it doesn't is if you cut the simulation off at an earlier *p* value. Otherwise, it will eventually go to 1. The opposite is true for the number of Stable Imprints. It will eventually go down to 0, no matter the parameters passed into the simulator.

5.2. THE THEORY FOR BRUTAL TESTING

For the first part of the assignment, a lot of data was discovered following the experiments. The main purpose of running the brutal experiments was simple... I theorized that the precision in the graphs would increase as there were more patterns and neurons. This ended up being the case, however, it didn't scale as an exact 1-to-1 ratio. More importantly, with this data, we can also conclude statistics for increasing any of the parameters passed into the simulator.

5.2.1. INCREASING NEURON COUNT

As the neural size increases, more patterns are in the system are stable. So if we wanted more stable patterns, we can simply increase the number of neurons for each pattern.

5.2.2. INCREASING PATTERN COUNT

Pattern count is simply the *p* variable. This value doesn't affect the neural network after the number of stable imprinted statements hits 0. If the number of imprinted statements at the pattern count is not 0, the simulation is cut off early. This is shown in Experiment 3, where the simulation is cut off before the values have stabilized.

5.2.3. INCREASING REPETITIONS

The simulator repeats the simulations (by default) 50 times, then averages their values to generate the final CSV file for an experiment. Increasing this value will result in more accurate information that is being written to the CSV file, as there will be more data to average out.

5.3. TAKING IT ONE STEP FURTHER

There is one challenge left. As we saw in experiments 4, 5 and 6, the pattern count should be lower by a certain value in order for the graph to look the same as in experiment 1. Furthermore, notice that, after a certain point, the fraction of unstable imprints is always 1, and the number of stable imprints is always 0. There exists an estimated ratio that we can use to predict when in the simulation that we hit this point.

5.3.1. THE MAGIC FORMULA

At first, the ratio was assumed to be a constant. So I looked at the data in Experiment 1 and Experiment 6 and found 2 spots late in the simulations where the numbers were the same. For Experiment 1, this was p = 43. For Experiment 6, this was at p = 1148. They both equaled 1 and were assumedly not going to change beyond that point. We can then generate the constant like so:

$$p_r = \frac{1148}{43 * (\frac{5000}{50})} = 0.266976 \ldots$$

Figure 18. *Equation to compute ratio to ensure graph consistency.*

While this ratio works with Experiment 6, as around 75% of the experiment is the same value, this does not apply to other experiments such as Experiments 4 and 5. My next guess was to use a

Power Regression in order to give a equation that predicts a number of patterns needed in the simulation based on the number of neurons. That equation was generated by the following table of data:

Experiment #	N	FOUI = 1	Ratio
1	100	43	1
4	200	69	0.802325
5	2000	387	0.45
6	10000	1148	0.266976

Table 1. *Data used for Power Regression. N and Ratio were X and Y respectively.*

Power Regression was performed and the following equation was returned:

$$P = 1.91524962n^{0.697007102}$$

Figure 19. *Equation to predict number of patterns needed to run a simulation.*

We can combine the equations in figures 18 and 19 to generate a ratio of how many patterns in a given simulation are actually useful. If a user gives a simulation with an absurd about of patterns, we can correct their choice with this ratio.

$$P_r(P,n) = 1 - \frac{1.91524962n^{0.697007102}}{43 * \left(\frac{P}{50}\right)}$$

Figure 20. *Equation to give ratio of patterns that have useless information.*

P *is the number of specified patterns, and* ***n*** *is the number of neurons.*

Our simulator accepts 3 inputs: Patterns, Neurons, and Repetitions. With this formula, we can take 2 of those values and figure out the following:

1. Ratio of patterns that actually mean something in our simulation.
2. Number of patterns before we can cut off the simulation without losing any data.

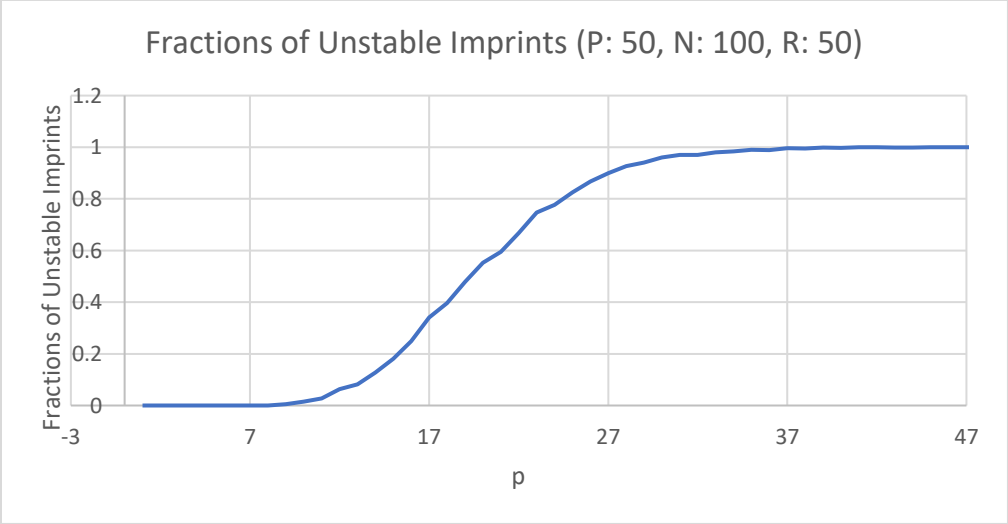
5.3.2. PROOF: EXPERIMENTS 1 & 4-6 WITH RATIO CORRECTION.

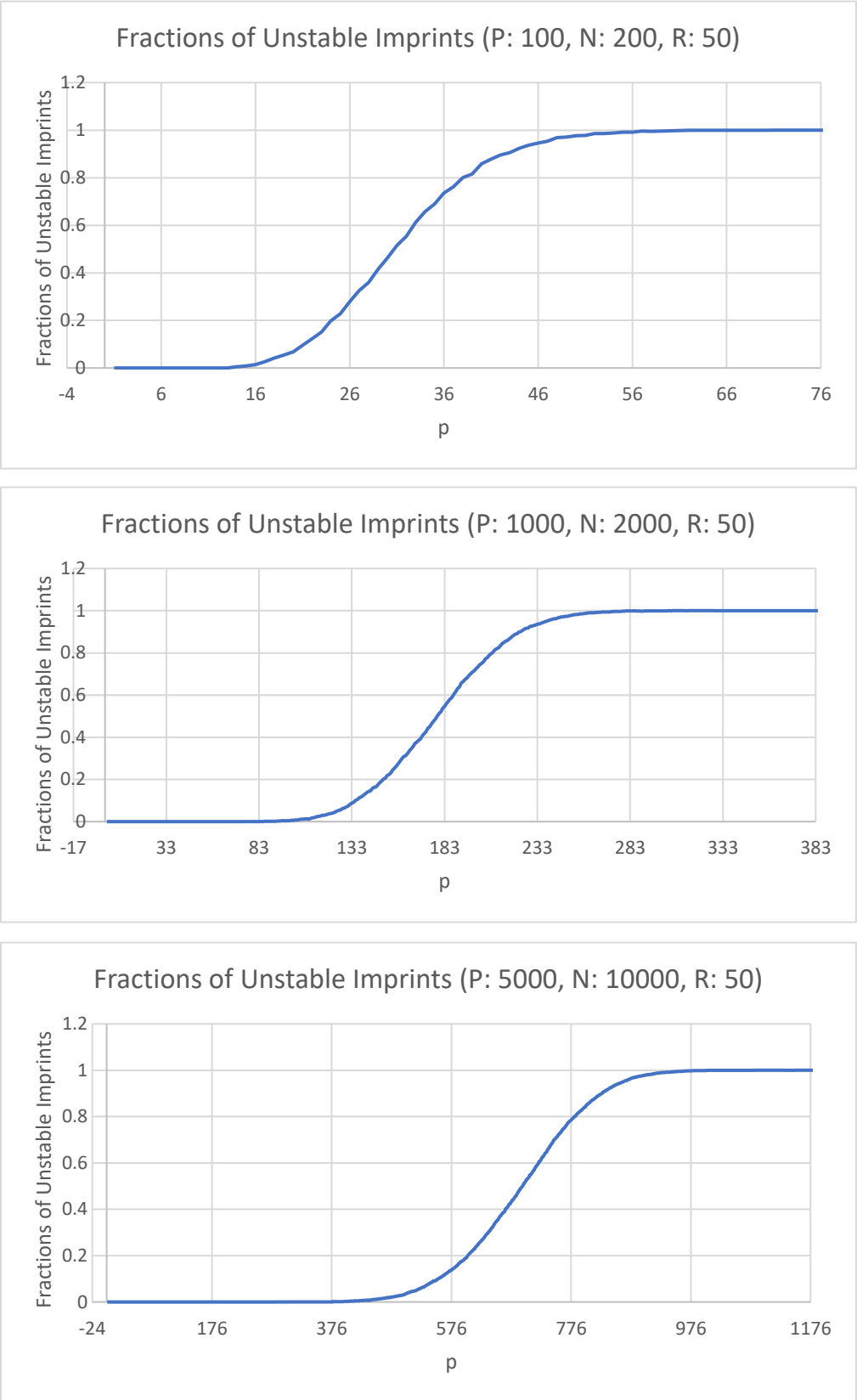
Let's look back at previous experiments such as Experiment 1, 4, 5 and 6. We can use the equations in Figure 19 and Figure 20 to predict when to cut off the simulation.

Ex. #	N	p	% of useless info (Figure 20)	Min Patterns (Figure 19)
1	100	50	0.00%	47
4	200	100	10.5539467%	76
5	2000	1000	55.4786541%	383
6	10000	5000	72.6608293%	1176

Table 2. *Prediction of percentage of useless information and minimum patterns.*

Let's cut off the grids from the experiments and see the results.





Figures 21, 22, 23, and 24. *Results for Experiments 1, 4, 5 and 6 after useless information.*

As expected, the graphs cut off right at the very end of the simulation. If there was more computational power at this time, more values could be shown.

5.3.3. WHY BOTHER?

This is to allow for program efficiency. Look back at the graphs for experiments 4, 5 and 6. You should notice that, beyond a certain point, it's just a flat value. This is wasted computational time. However, with the ratios we have, we are now able to predict when the neural network will constantly be unstable. With that, we can cut the simulation off early, and guarantee that we are not losing any data. We can also use this to inform the user how many of their patterns will contain useless information before they even run the simulation.

6. EXPERIMENTS (GRADUATE PORTION)

In the graduate portion of the assignment, we had to keep track of basin sizes as well. This is done by keeping a 2D vector in each simulation that holds data about each size for each pattern. This is done for each simulation, and then the values are averaged. However, for the sake of matching the lab writeup, the experiments will consist of both the regular and normalized versions of each graph.

It is unusual that the project writeup decided to show them backwards, however, the simulator computes the values, then normalizes afterwards by dividing the entire vector by the pattern it's on. This is because, as observed, the basin size is guaranteed to not pass the pattern number that it is on. So when divided by that number, it's guaranteed to be between 0 and 1. Just like the undergraduate part in Section 4, we will experiment with a few options with the basil computation.

6.1. EXPERIMENT 7 (P: 50, N: 100, R: 50)

Let's try to match the lab writeup's graphs. That's a pretty reasonable goal. For this experiment, we will match the original lab requirements, being 50 patterns of 100 neurons each, being repeated 50 times.

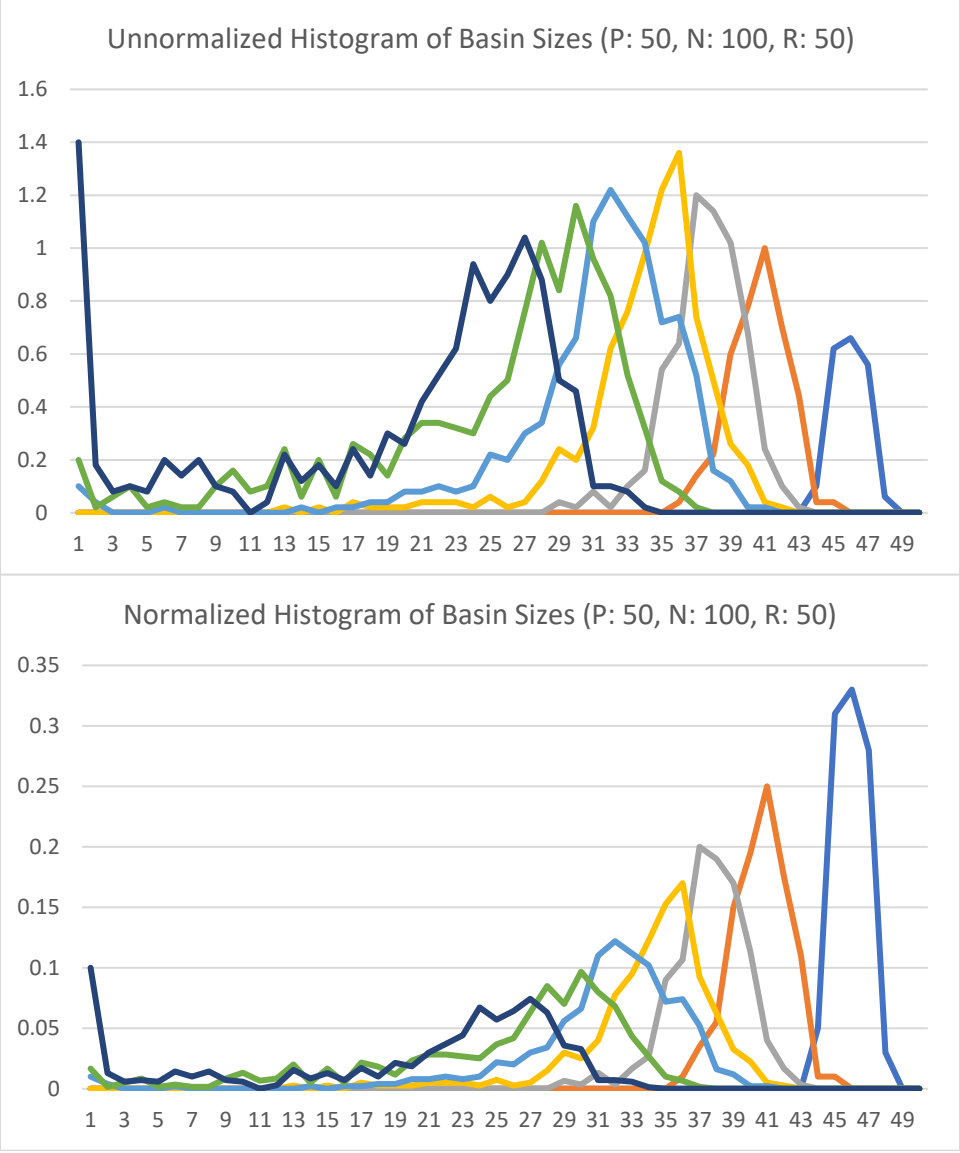


Figure 25 (Top) and 26 (Bottom). *Histogram graphs for Experiment 7.*

The graph generated matches the layout of the one in the writeup. Now let's take it a step further.

6.2. EXPERIMENT 8 (P: 100, N: 100, R: 50)

So to begin a *true* analysis of how this works, let's try doubling the number of patterns inspected.

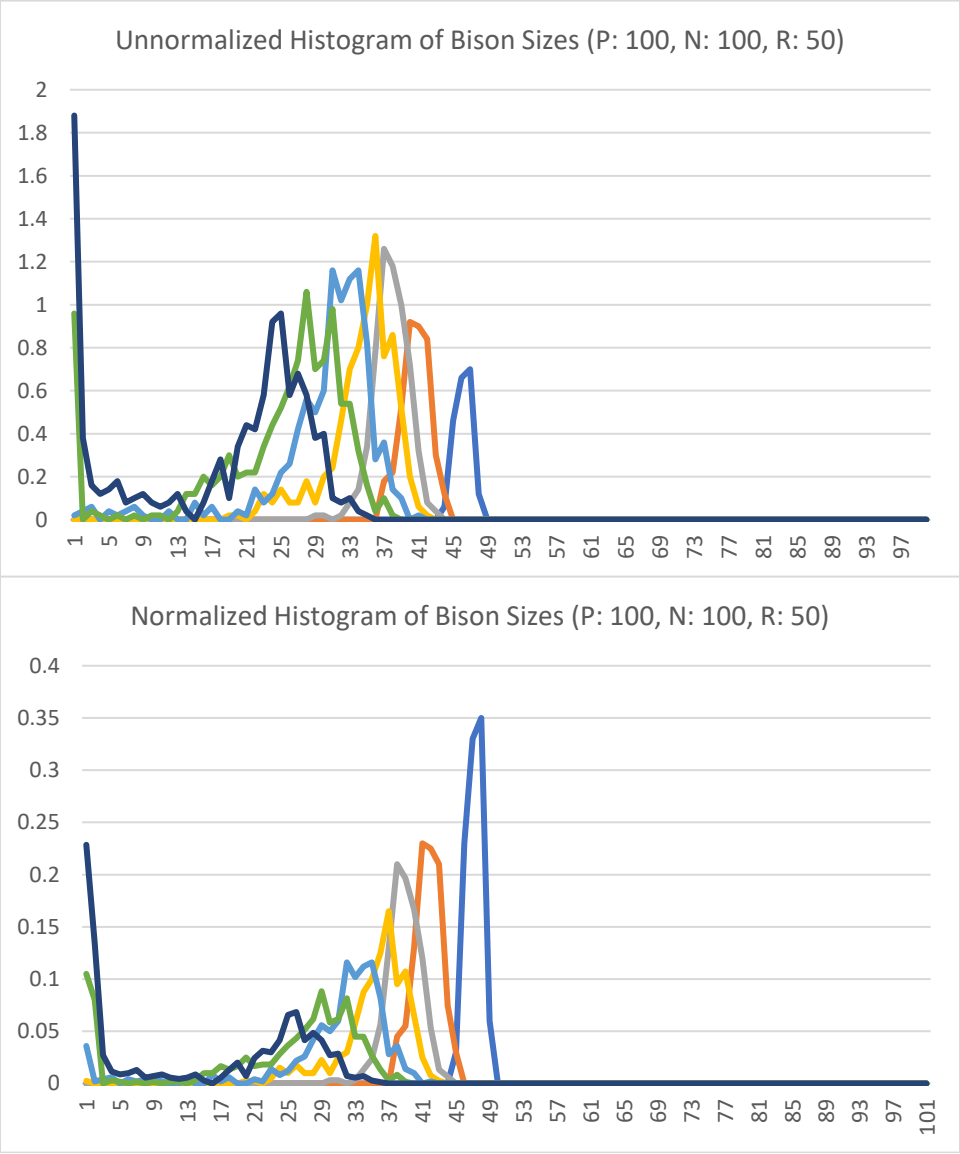


Figure 27 (Top) and 28 (Bottom). *Histogram graphs for Experiment 8.*

Just like the undergraduate portion of the experiment, the number of patterns simply increased width (or range) of the graph. The output is similar to the output in Experiment 7, so this doesn't have any dramatic effects at all.

6.3. EXPERIMENT 9 (P: 50, N: 200, R: 50)

Just like Experiment 3, let's double the number of neurons instead.

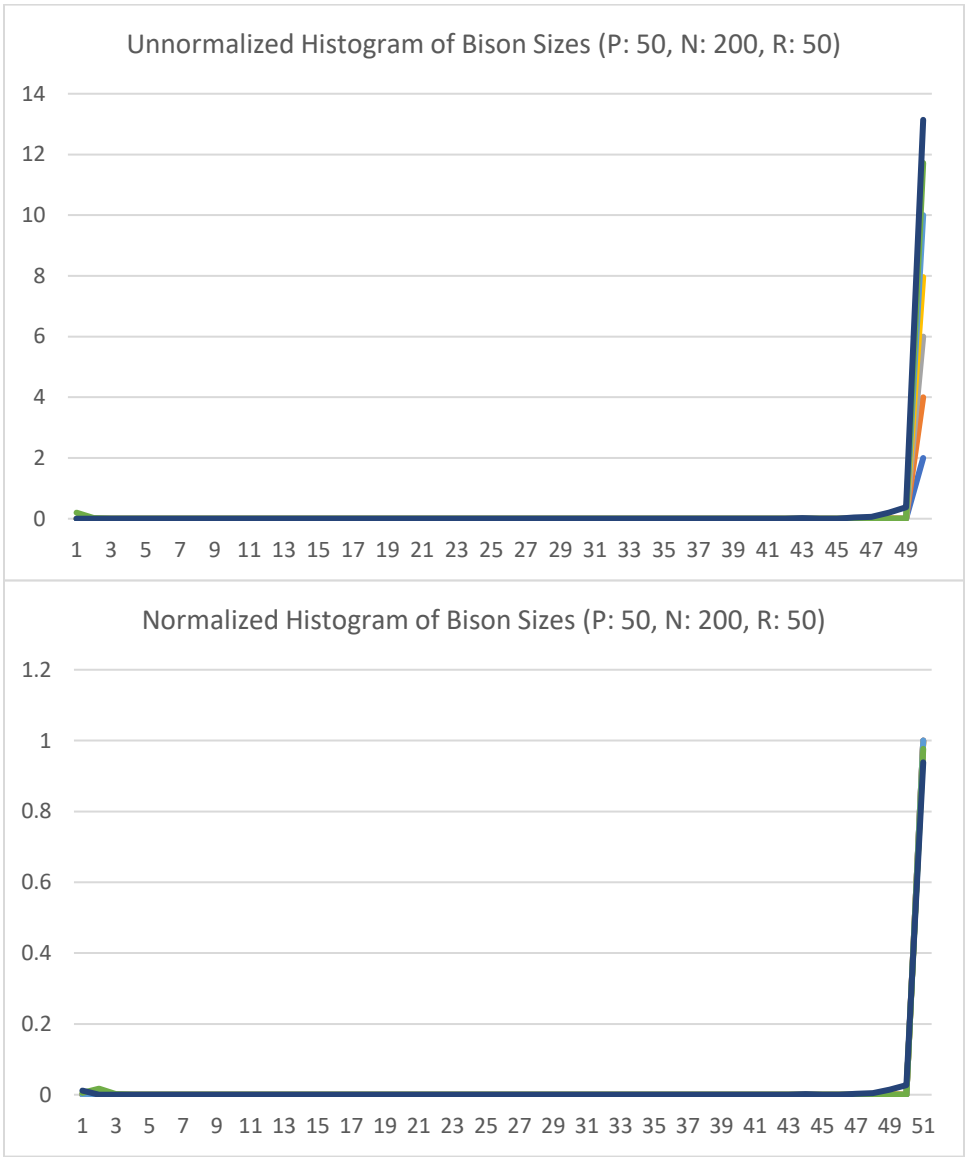


Figure 29 (Top) and 30 (Bottom). *Histogram graphs for Experiment 9.*

This is where some interesting information starts to emerge. It seems that doubling the number of neurons has caused the bison sizes to stretch out a bit. However, it appears that it also delays when they start filling up the graph.

6.4. EXPERIMENT 10 (P: 100, N: 200, R: 50)

Experiment 9 consisted of barely anything, so let's use what we know and double the number of patterns. That should extend the graph out enough to let us see some information.

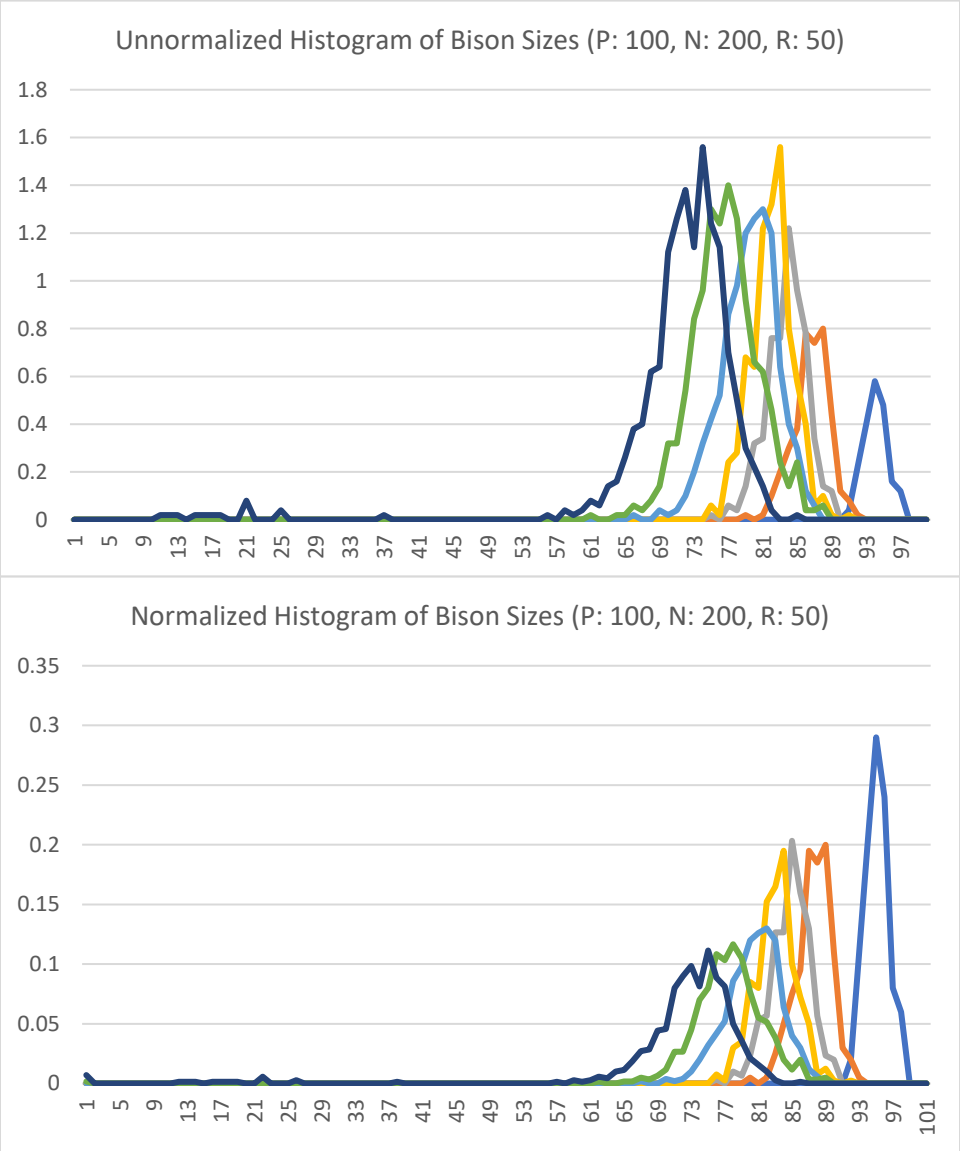
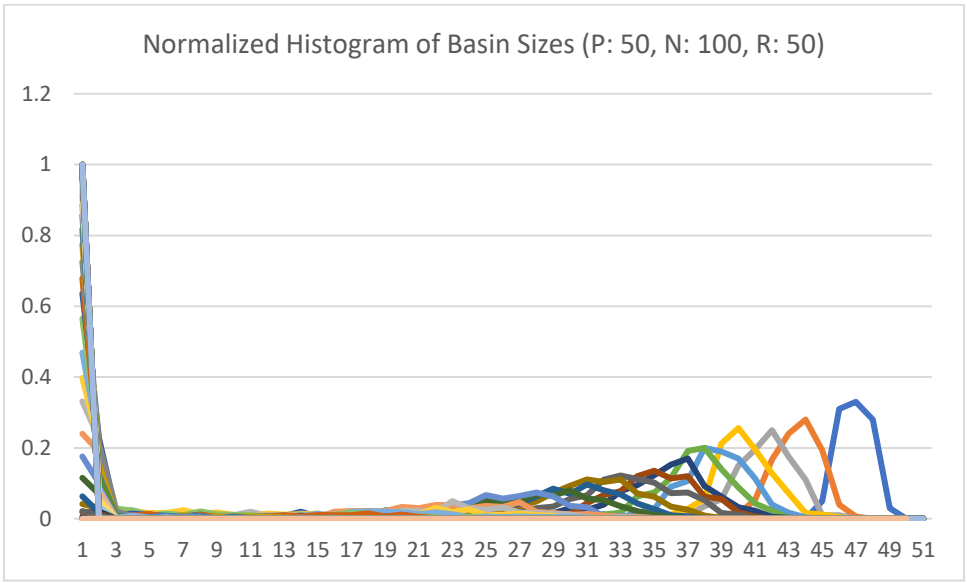


Figure 31 (Top) and 32 (Bottom). *Histogram graphs for Experiment 10.*

We now have information that looks like it came straight from Experiment 7, but was slid over by 50. The range of the basil sizes are still the same, so the graph wasn't stretched by any values.

7. Observations (Graduate Portion)

Unlike the undergrad portion, this portion of the project did not get an OpenMPI port. Nor did it undergo extreme tests with insane amounts of neurons or patterns. One of the reasons for this is because the values would be within 0-1, so it's guaranteed that it won't scale the same way as the graphs in Experiment 1, and 4-6 did. Furthermore, Experiment 10 displayed that the values simply slide over as the number of neurons increase. They do not multiply.



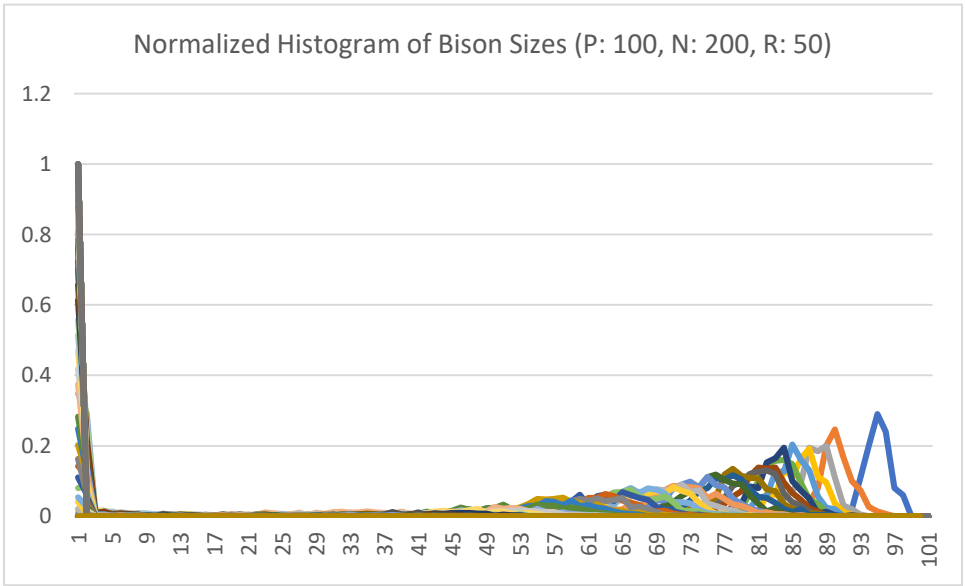


Figure 33 (Top) and 34 (Bottom). *Experiments 6 and 10 with all p values shown.*

From Figures 33 and 34, if we chose to include *all* data from Experiment 7 and Experiment 10, we can clearly see that Experiment 10 has more data in it. More importantly, the X axis is indeed stretched out to 101 as opposed to 51. However, the y axis stays mostly the same regardless of changing any of the parameters. Take note, though, that this their normalized forms. Figures 35 and 36 are their original forms:

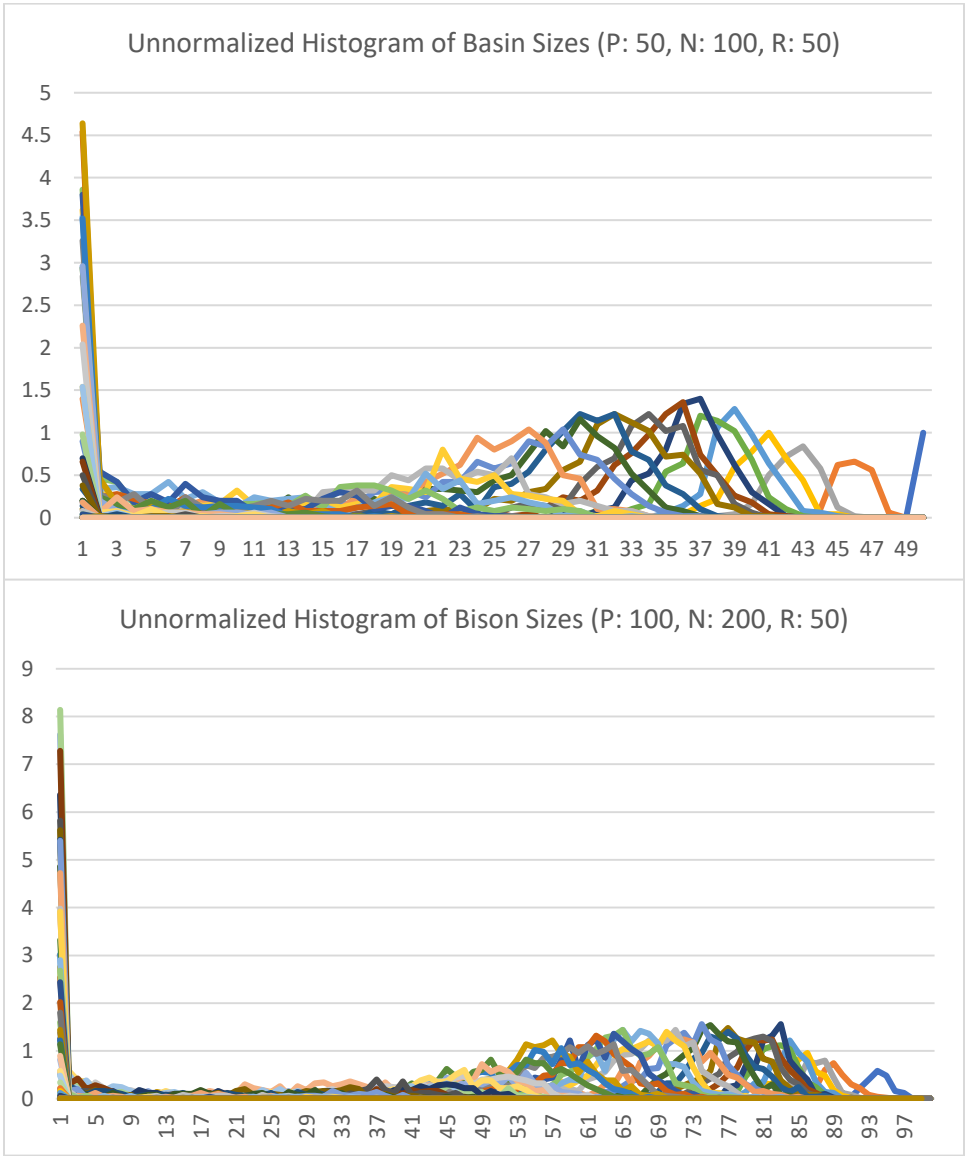


Figure 35 (Top) and 36 (Bottom): *Non-normalized versions of Figure 33 and 34.*

Even without being normalized, the values still didn't change by much. Both peak at around 1.5 (with the exception of the first p value in Experiment 10 going up to over 8). The mess just seems more detailed when there are more neurons and patterns, but it appears to only stretch horizontally, not vertically on the graph.

8. PROJECT FILES AND SUBMISSION INFORMATION

8.1. C++ SOURCE CODE FILES

You can find the source code to the single-threaded and OpenMPI multi-threaded programs in the "src" directory of this submission. To compile, have the makefile **outside** the "src" directory, as it will traverse in the directories and compile for you. In order to compile hopfield_mpi, you MUST use **mpic++** to compile it. It is on Hydra, but you may have to include some additional directories in order to run the executable.

8.2. PRE-COMPILED BINARIES

For the sake of convenience, there are some pre-compiled binaries that will run on an x86_64 processor (Hydra included) just in case there are difficulties trying to compile this yourself. "hopfield" and "hopfield_mpi" are included.

8.3. SHELL SCRIPTS

The "scr" directory contains a few shell scripts written by me to make the task of automating the experiments easier. "killall_on_hydra.sh" and "killall_on_tesla.sh" are obvious. "nuke.sh", "nuke2.sh" and "nuke3.sh" are dangerous scripts that will execute "hopfield_mpi" on all 31 Hydra and/or Tesla machines. Don't use it unless you want to run an extreme test like Experiment 6.

8.4. EXCEL DOCUMENTS

In the “experiments” directory, there is an “Experiments.xlsx” file, which contains all data used in this report, along with all of the graphs generated. Overall, it contains the following:

- Experiments 1-6 (Regular) and Experiments 1-5 (OpenMPI)
- Experiments 7-10 (Regular)

8.5. CSV EXPERIMENT FILES

In “experiments/csv”, there are 4 categories of csv files:

- **undergrad_reg** – Experiments 1-6 without OpenMPI
- **undergrad_mpi** – Experiments 1-6 with OpenMPI
- **graduate_reg –** Experiments 7-10 without OpenMPI
- **graduate_basin –** Experiments 7-10 Basin Information

There also exists a “experiments/misc” directory which contains other files that are not particularly useful.

8.6. DOCX MASTER FILE

This document itself has a master file that is included in the “docs” directory under “3 - Hopfield Net.docx”. Of course, this file (“3 - Hopfield.pdf”) is in the root directory of this submission.