USING STATISTICAL DISTRIBUTIONS FOR GENERATING RANDOM TEST DATA

by

JAMIE ZIMMERMAN

A THESIS

Presented to the Department of Computer and Information Science
and the Robert D. Clark Honors College
in partial fulfillment of the requirements for the degree of
Bachelor of Science

June 2018

An Abstract of the Thesis of

Jamie Zimmerman for the degree of Bachelor of Arts

in the Department of Computer and Information Science to be taken June 2018.

Title: Using Statistical Distributions to Generate Random Test Data

Approved: _____

Dr. Michal Young

Many open-source software programs lack rigorous, wide-reaching testing. This is primarily because the testing process is deeply dependent on human invention and manual writing. Therefore, writing system tests is often avoided because of its financial and temporal expense on the software development lifecycle. However, the data for some system tests can be generated from statistical distributions that exhibit certain trends or patterns when interpreted in the context of the program. Therefore, this approach can tease out specific behaviors in the program that warrant being checked. This project aims to automate the creation of those data points to streamline the process of writing large test suites.

Acknowledgements

It is with the deepest gratitude I salute Dr. Michal Young for his wisdom, guidance, forgiveness, and intellect. I thank him sincerely for our weekly meetings for which significant portions were devoted to chats about bicycles. I also thank Dr. Hank Childs for his seemingly inexhaustible generosity in taking on another student's senior project. I express to Dr. Rebecca Lindner appreciation for her constructive motivation to always improve my writing communication, bright attention, and encouragement to remain a scholar in the Honors College, well before this project had materialized and when I thought I had no chance of completing it. I am forever thankful to my three advisors for encouraging me to strive for the highest.

Table of Contents

List of Accompanying Materials

1. GenSequence: https://github.com/TestCreator/GenSequence

2. GenPairs: https://github.com/TestCreator/GenPairs

List of Terms

SUT - Software Under Test, the program to be tested

Production Rule - a recursive rule that describes what the left-hand symbol could possibly be. For example, an A could be a B, denoted by A $\rightarrow$ B. Whenever an A is seen, one can choose to replace it with a B.

Test Vector - A symbolic description of a test case, lacking exact data points but having an English language descriptor

Fault - a misbehavior in a software program, also known as a bug. The behavior can vary from a program crash to an unexpected, nonsensical output

CSV file format - a comma-separated value file that is viewable in Microsoft Excel

List of Figures

List of Tables

Chapter 1

Introduction

Software testing is a crucial part of systems that run our lives. Industries that rely on predicting consumer habit trends or dispatching taxi cars are deeply affected when their software fails to provide accurate results. The gravity of this trust is even greater in safety-critical programs controlling gas-leak shut off valves or anesthetic delivery machines. Consequently, before these pieces of software are deployed, they must be checked and tested as rigorously as necessary, but no more. The breadth and completeness of testing is directly proportional to how much damage could occur if the software malfunctioned. The software must eventually reach production (used in the hands of the customer or used in industrial practice "for real") and cannot spend too much of its lifetime under test. Therefore, cost and convenience of testing is an important consideration in determining the rigor of testing. Neither cost nor convenience is really ever that low because few tools exist to make that job easier, so software testing remains largely overlooked in industrial practice; developing new techniques is too expensive and time-consuming and that is why engineers don't do it.

A key component of getting correct results is knowledge of what that correctness looks like. Developers and testers must be careful not to confuse code that compiles and runs without errors with code that garners accurate results. To find accurate results, testers need an oracle, which is a way of determining how close

the actual result of a program is to the expected result. Figure 1.1 illustrates the divergence. Unsurprisingly, attaining and checking this oracle is difficult. Given a particular input to a program, a tester must know what the expected result even is. Then, she must know how to read or understand the actual result and be able to compare it to the expected. Consequently, many companies often build the role of tester into the role of developer, since the developer has the most intimate knowledge of the design and implementation of the software, and therefore has a better sense of what the output of the program garners. Even then, it may be impossible to describe the ideal expected output of the program.

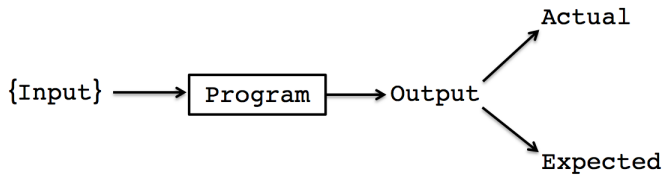


Figure 1.1: Basic Workflow of a software program

However, there are some basic oracles and trends that can be tested that can offer high confidence that the program functions as anticipated. For example, it might be difficult to know what a piece of software should do, but significantly easier to know what the software should not do. Online banking software should not deposit millions of dollars into the account of a blacklisted felon. Another major challenge in software testing is the need to know all the possible situations you may want to test for. In the automobile simulation, a tester does not want to just test for braking and accelerating, they might want to test turn-signaling and beeping and brake-lighting, and also combinations of those tests to make sure that the beeping sensor does not accidentally disable the brake pedal capability. Knowing and describing every single situation may be unknown to the tester; they may not even think that that is a situation they have to test for. What is known as the "happy path", the operational choice that most represents the typical testing scenario, is certainly necessary for

testing the programs reliability, but it does not find the bugs. Outlier situations find the bugs. Identifying all these possible outlier situations is **Problem One**.

The next problem is writing the actual test data itself. A symbolic description of the test must be turned into an actual concrete input to the software. A test vector checking that beeping does not disable braking must be turned into "beeping=5s&&braking=true" or whatever discrete format the SUT requires. This is not a step that should be done manually. For example, VisIt, a graphical visualization tool [3], is capable of handling several gigabyte files, and no tester wants to or even could write three gigabytes of data points by hand just to test something. This is **Problem Two**.

It is significantly easier to describe the trend of a certain test than it is to create the hundreds (if not more) data points that fit that description. Therefore, the goal of this paper is to bridge that gap. It aims to aid the tester in designing and creating test suites so as to provide automation in the testing process. The reality of the software development environment means that solutions must be adopted to ensure accurate software while still maintaining a practical timeline and budget.

Finally, it is important to consider the necessity of generating this data in the first place. Gotterbarn remarks, "Insufficient data is not a problem" [2], implying that data exists for most programs we want to test. For example, consider an ocean temperature monitoring software that has predictive power in future local hot spots or cold zones. Ocean temperature data points do exist, and the missing points can likely be interpolated quite easily. But this is a sample size of one. To be sure the temperature projection software is robust, the tester would want to study a variety of circumstances and possibilities situations that do not even exist. They might perhaps want to study temperature diffusion trends as a result of a significant event 30 years from now. Data for that particular test case has to be created.

Chapter 2

Literature Survey

The essence of the oracle problem is best captured in the definition of non-testable programs, provided by Weyuker:

> A program should be considered non-testable [if] (1) there does not exist an oracle; (2) it is theoretically possible, but practically too difficult to determine the correct output [1]

There are a variety of reasons why the oracle does not exist or can be exercised in a practical capacity. Weyuker describes that some software systems are like magic calculators they are meant to inform us of the answer. She additionally mentions that some programs produce output that is impossible to read, either because of its volume or complexity. Here lies the core of software testing challenges. It is impossible to solve for every oracle or be able to describe every detail of a complex oracle every system is entirely different and no generic template could create a standardized oracle. However, it is possible to avoid oracles altogether, generate them partially, or give hints about what details are most important to check.

One method of avoiding oracles altogether is metamorphic testing, which exploits the relationships of different executions of different input data. For example, upon white box inspection of a system, testers can see that outputs should relate directly to their inputs. They may not know much else about the system, or what its output means, but they know what differences they should see upon two different executions

. Upon a certain execution they get f(5) = 20, and since they know the input-output relates directly, then they should predict that f(6) ¿ f(5). Though exceptional in avoiding oracles, ascertaining metamorphic relationships is as challenging as the oracle problem.

Other researchers have proposed methods that do not attempt to provide or calculate an oracle, but rather aid the tester, informed of the structure of the system, exactly what details she should watch and constrain about the oracle. This takes the form of determining, through a series of tests, which variables in the system are most effective in revealing faults or bad behavior, and then providing them to the tester to define the "correct" value. One research group used mutant generation to find which variables killed the most mutants and, therefore, found the most faults . An alternative solution identified chains of dependent variables using probabilistic substitution graphs to find the most important variables . These works make the best advances in the reducing the human effort needed to define and fulfill a complicated oracle.

One group from Columbia University developed a similar approach when testing Machine Learning models, called "parameterized random data generation". They identified equivalence classes in their test data, which they then used as constraints for random data. This allowed them to develop huge data sets that still followed patterns that their model expects. They made a significant advancement in test data generation by adding more specification to what they want in their test data than just specifying data type or range of values. However, their approach is designed for Machine learning applications, so the values produced are restricted to positive integers only. Moreover, they have only used uniform distribution as their random selector.

Another group used Perlin noise as a data generator for images representing the spread of non-native pests in British forests. They layered multiple realistic images together and used spatial statistics to "optimize" their images. In other words, they identified traits from real test images, like a histogram of pixel values, and applied them to their own images to enforce similar traits. Most importantly, their tool generated images extremely efficiently and therefore quickly, making their tool exceptionally streamlined. However, their tool did not aim to provide any oracle information, and instead opted to using metamorphic relationships between real images and their fabricated images to find faults.

One group with members in Brazil and Luxembourg used their previous work on a method that identified how close test data approached synthetic bugs to create a new heuristic that "finds" data points that could reliably strongly kill the mutants in a program. Their approach works backwards by using a search scheme to identify data points at potential test candidates.

However, a more focused approach on database testing frameworks shows similarities to my own project. Generating test data for database-driven applications is still a niche research area and therefore lacking in advancement. One paper described the typical approaches for commercial data generation tools are actually quite limited in their power, capable of generating data that passes syntactic level checks, like type-checking, and "not null, unique, primary, and foreign key constraints" . This is essentially generating data that is database compliant but has little semantic sense.

Chapter 3

Proprosed Argument

I believe that I have designed an approach to these problems and have architected a simplistic solution that automates the steps described in the introduction. I hypothesize that Problem One is solved by the advent of combinatorial testing, which identifies most, if not all, of the test cases necessary for confidence in working software. My tool, Parmgen, has solved the second problem, translating the English description of a test case into a concrete format that can be used as input into the program. I ask the user for smaller, more simplistic descriptions of their input data, which requires less work than writing every data point by hand. Then I can use Python's random statistical distributions to generate those points and write them to a convenient location for the testers use. This tool is written in the Python language, which is useful for prototyping the MVP (Minimum Viable Product) of this tool. I call this entire pipeline GenSequence.

Chapter 4

Background on Design Decisions

4.1 Parameters

The "input" to a program is a set of parameters, named variables that represent specific data points. How those parameters are generated, and what literal form they take, affects the outcome of the program. Large programs with larger inputs may benefit from having their input parameter data sets generated by statistical distributions.

I was inspired to create this tool by testing a project I helped build in my software engineering class, which took in the records of each student in a class of 40 students, describing things like their skills in certain technologies and their available free times during the week. This project made a fairly sophisticated decision about how to form optimal groups of students to work in teams, ensuring that each group has complementary skills and overlapping free times. During testing, it is important to consider bizarre scenarios, like what should happen when no student has any free time at all, or when half the class can only meet early in the week and the other half can only meet late in the week.

These parameters can follow certain trends that can be mimicked by repeated trials of statistical distributions. The input to this program is merely an Excel csv file of a record table, such that every row is a student's entry and every column is all the entries for certain parameters that really effect the outcome of the program.

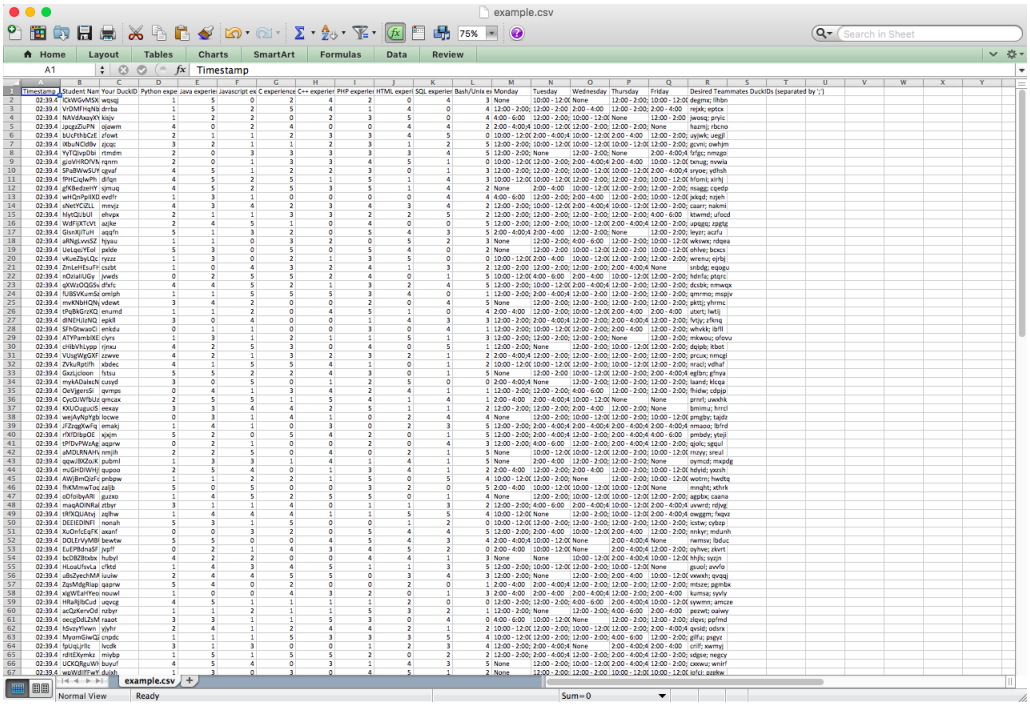


Figure 4.1: A CSV file of a test case input into the Team Builder program

Figure 4.1 shows the CSV file of the literal input into the Team Building program. The software takes in the filename, opens the file, and parses the data. Each row is a student, and each column is a parameter, such as free time availability on Monday or self-rated HTML experience level.

4.2 Pairwise Testing

Using a pairwise testing tool developed by the authors advisor , Michal Young, we can create symbolic test vectors that describe what each test case should generally look like, using English adjectives to describe what the input should be. Pairwise testing rests on the hypothesis that a majority of faults in a program are a result of combinations of parameters that work with one another to create or exacerbate a problem. Creating the minimum number of test cases that satisfactorily describe all combinations of possible parameters creates the maximum code coverage. The success of finding all faults increase with greater numbers of combinations of parameters;

that is to say that all combinations of four parameters will find more faults than all combinations of three. However, pairwise testing finds all combinations of two parameters and is often sufficient for maximum code coverage while still reducing the time, and therefore cost, of creating or running all test cases. This is even more likely when the total number of parameters is also somewhat low. Combining only two parameters is a matter of convenience and time, since more test cases require more time to create.

While translating a symbolic test vector to concrete data may present a challenge, it is extremely valuable in reducing the uncertainty of knowing what sort of trend the input test data follows. Say, for example, a tester needs to generate a concrete point for this test vector (Table 4.1)

Item Purchased	*Price*	*Delivery Method*
Large Item	Expensive	Ultrafast

Table 4.1: Single Symbolic Test Vector

She knows that some test case will send a large item by ultrafast means. In our generation of concrete data (described in the next section) in this test vector, she might come across the test case in Table 4.2

Motorboat	18700.00	Jet Airplane

Table 4.2: Single Concrete Test Case

The interaction of these two parameters, a very large object sent by ultrafast means, ought to trigger a bug in the system, because perhaps her company does not have that capability. If this situation is not handled correctly in the software, the user of the software, the customer, will be misled about the quality of their shipping network. This example is very trivial, but it exemplifies the problem well, and shows how pairwise testing methods can bring this awareness to the tester and developer and help them improve their software. Now imagine a much longer and complicated

test vector in a very advanced version of the companys ecommerce website; maybe now it has millions of possible item categories and hundreds of shipping methods and can also handle coupon codes and student discounts and special offerings: many more parameters to consider. It is helpful to know that the test vector was created from a description including Large Item - Expensive - Ultrafast. The symbolic test vector describes the input well enough for the tester to understand and use common sense to predict what the output ought to be. Then if she tests that test case and finds that the customer receives a message that sending a boat through lightning speed mail is impossible, she gains more confidence that her program works correctly.

The pairwise testing tool is designed such that the tester describes all the possibilities for what each parameter could possibly be, and then the tool generates all the test vectors that describe what the parameter ends up becoming. When using this tool, testers should list every kind of statistical distribution that they would be interested in seeing, or what makes sense in the context of their program.

4.3 Context-Free Grammars

The aforementioned skipped step uses context-free grammars implemented in the Python templating tool Mako to generate concrete test data. We call this feature Makogram. Say a test case has the following production rule:

⟨*Case*⟩ ::= ⟨*Item*⟩ ⟨*Price*⟩ ⟨*Delivery Method*⟩

What constitutes an item? Item might have a production rule

⟨*Item*⟩ ::= '`Ship`' | '`Book`' | '`Loaf`'

The grammar tool used is constructed such that a non-terminal can be the result of a function, and that result can be a randomly generated data point. For example,

⟨*Price*⟩ ::= @ lambda x: random.choice([i for i in range(x)])(10)

Meaning that the data point for the Price parameter would be some number chosen at equal randomness from any number between 1 and x (in this case 10).

A terminal symbol can be written to be the returned result of a much more sophisticated function - one perhaps that returns a Gaussian distribution of data points. For example, consider an earthquake modeling program, which takes a set of data points for its magnitudes parameter, and it can have all sort of descriptions attached to it: Gaussian, uniform, cardioid, etc. All of these descriptions will occur somewhere in a test vector, so each is mapped to the corresponding function. When the magnitudes parameter is set to be generated from a bell-curve distribution, the bell-curve random generator will generate that set of points.

Generating good test data for actual magnitudes of an earthquake must be based on the knowledge that a magnitude cannot be negative or greater than 10.0. This is the limit of the program. It cannot make logical guesses about the natural language of the parameter. But the tool is written to minimize the number of times the tester must identify this constraint - just once, in the beginning.

4.4 Law of Large Numbers

The Law of Large Numbers states that the actual outcomes will approach the expected outcomes as the sample size increases to infinity . One canonical example is a series of coin flips, with an equal probability of flipping heads as tails. It is expected that exactly 50% of the samples will be heads and 50% tails. In a sample size of only one hundred flips, the percentage of heads to tails may only be 46.0% - 54.0%, but a sample size of ten thousand would come far closer to an even 50-50 split, possibly 50.37% - 49.63%.

This principle applies to all kinds of probability distribution types, not just coin flip probability. Students grades tend to figure towards a bell-curve distribution, but a graph of 50 students grades will look more misshapen than a graph of thousands

of students grades. Moreover, the principle applies to many programming languages utilities that support randomness. The Python language contains a built-in module called random which provides a variety of generators for all flavors of probability distribution. Individual points are generated according to a particular distribution, and the generation of a large enough sample size is guaranteed to follow the expected trend.

How large is large? Bernoulli proved that the actual outcome of a simulation would approach the expected outcome as the sample size grows to infinity. So in general, the greater the sample size the more accurate the results.

If the use case arrives on the order of hundreds of data points, a default sampling size on the order of ten thousand randomly generated data points should suffice. Because the initially generated set of points is far too many, I have devised a scheme to reduce the sample set to the appropriate size while still encapsulating the distributions guaranteed by the Law of Large Numbers. For each parameter, I generate the too-large sample size, sort it in increasing order, and selectively choose every nth point to include in the reduced set of points, such that the size of the reduced set is the desired number how ever many the test case needs.

This selection scheme is yet to be proven rigorously, but I hypothesize that it suffices for now, and does indeed capture the essence of the Law of Large Numbers. Pictorial representations show that this scheme is not perfect, but is certainly far better than choosing exactly the desired number of points. Visibly a sample size of 300 systematically chosen from a large set follows the curve of 10,000 points much more closely than 100 initial points.

4.5 Cardioids

In some testing situations, it may be useful to consider the relationship between two parameters, a relationship that mimics real life. For example, a physics-modeling

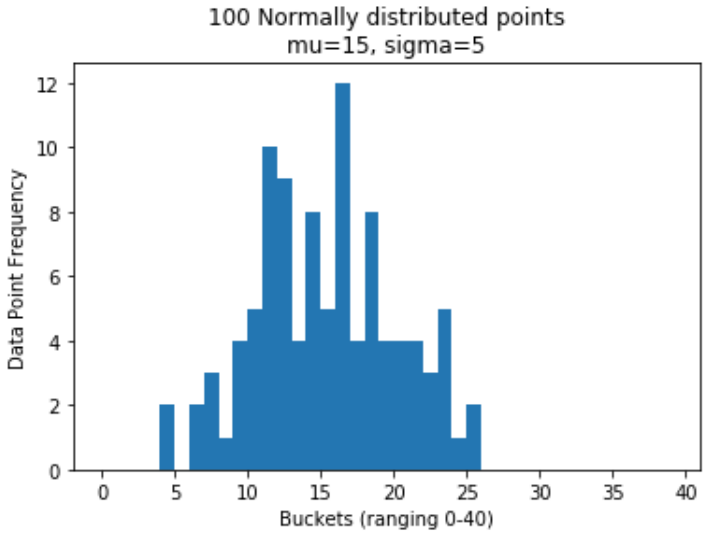


Figure 4.2: Histogram of point values generated by normal distribution - sample size 100

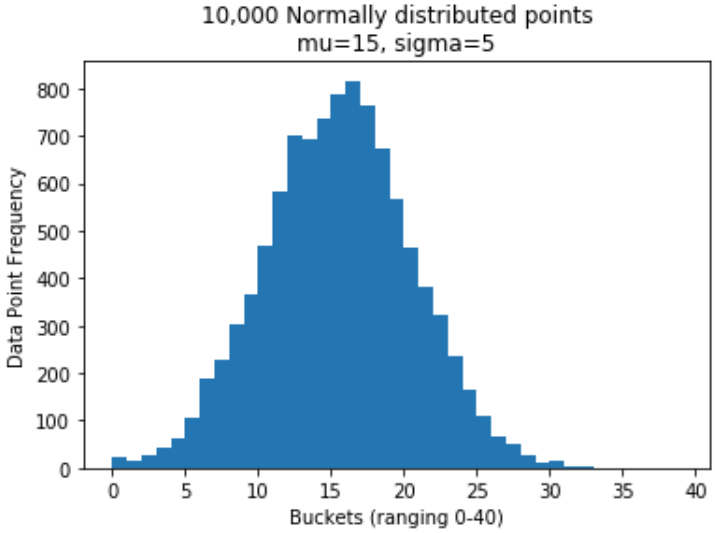


Figure 4.3: Histogram of point values generated by normal distribution - sample size 10,000

simulation might have a loose property that larger objects move slower than smaller objects. These two parameters, size and velocity, are interdependent.

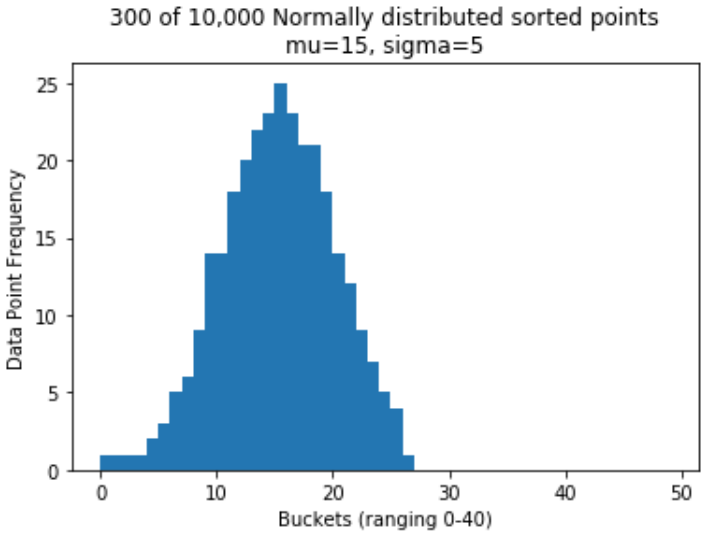


Figure 4.4: Histogram of point values generated by normal distribution and reduced to sample size of 300

A Cardioid[1] object acts as a two-column parameter. The end user specifies the relationship between the two sub-parameters by providing descriptions of what data point range pairings are preferred (called favorites) and what data point range pairings should only occur infrequently (called outliers or non-favorites). The data generation step will look at these descriptions and generate the data point pairings such that 90% of the samples are favorites and 10% are outliers. For example, the tester will specify favorites for the cardioid relationship between size and velocity: large and slow, medium and medium, and small and fast. Non-favorites are outlying particles: small and slow, and large and fast.

The cardioid relationship spanning two parameters occurs only if the tester specifies it as a possibility in the test vector creation step. There are certainly situations where it does not make semantic sense to use cardioid relationships. But, it does allow testers to generate tests that maintain realistic sense. It also allows the user to

[1]The name cardioid was chosen for its polarity. Its figure encompasses an area such that a majority of the space (about 90%) sits on one side of an axis and a small section (about 10%sits on the other.

include their own test data generation constraints if statistical randomness does not suffice.

4.6 Preprocessing

An important arm of the automation pipeline is developing a baby-sized language that is easy to write yet fully describes everything a tester wants out of this tool. Then a preprocessor can read this language and generate the code that, when executed, will create the data. Writing the data-generating code itself is complex enough that it limits automation, and building a machine that performs this step systematically significantly increases ease of use. I have fully developed the language into basic syntax and grammar rules, and I have constructed a simple preprocessor to parse it. It is very lightweight in that it compiles quickly, but does not have advanced error handling or give helpful compiler errors. The current iteration implements Ply (Python Lex-Yac). Ply works in two general steps. First it reads the input and identifies characters or words into tokens - essentially describing the type of each word or character. This is the lexing, or tokenizing, step. Next, the parsing step, parses the tokens altogether by identifying the grammar rules that combine them. Once the information has been identified as a derivation of a grammar rule, it is then available for execution.

The preprocessor is not completely automatic, and leaves a small portion of the data generating code up to the user. This is a complication of Mako templating, as I use Mako for both context-free grammars and specification injection.

Chapter 5

Methods

One key part of this project is deciding if it accomplished its goal and answered its research question. Is using random statistical distributions a valid approach to writing test data? The answer to this question must be supported by observations on the performance and quality of GenSequence.

Testers might use this tool at any stage during the software development cycle. Perhaps their application is highly advanced, has been neatly designed, and has fixed bug reports found through other means of testing or from end user reports. Therefore, that piece of software simply may not have that many bugs. That is unlikely, but it is possible. So measuring the usefulness of GenSequence on fault finding ability alone would depend on the state of the software it is testing, giving the indication that GenSequence is inadequate when really the software is just really well done.

The intention of this project was to streamline the testing process, and provided automated methods of data creation. The usefulness of this program is measured by the ease of test suite creation. Ultimately it is a testers decision as to how useful this project is, but these goals are nearly a self-fulfilling prophecy. More features added to the project only increases the ease of use, since a machine does more of the heavy lifting than a human brain otherwise would have. Nevertheless, there is further discussion later.

Finally, it is important to determine how valid the random statistical distribution approach is. A few case studies will be used to determine if knowing how a parameters data was generated helps clarify the expected result. First I will test my testing tool against a planetary orbits simulation. Next I will observe my tools data representation in a earthquake analysis program. Then I will identify open-source projects that have limited testing framework but are legitimate enough to benefit from test data creation, and generate test data for them using my tool. I will also consider these metrics: the length of the script generating data, how much code a tester would have to write, and how quickly the data is generated (more data takes more time).

Chapter 6

Results

6.1 Celestial Body Simulation

Initial use of GenSequence on the planetary orbits program proved some promise in the capability of this tool. I used the tool to generate 30 test cases. Test case 13 was programmatically named:

```
13-70-mass|right_slanted-position|right_-
slanted-velocity|uniform-diameter|left_slanted.csv
```

13 describes the test case number, 70 describes the number of rows in the test case. In this simulation, each row represents an instance of a planetary body.

Planetary masses were generated with a right slanted distribution. Position, the location of each body, was generated with a right-slanted triangular probability. The velocities of the bodies were generated uniformly. The diameters, the visible size in the program's GUI, were generated by a left-slanted distribution. The starting point of this program's test data showed this visualization:

Figure 6.1 accurately reflects the visible parameters position and diameter. The positions were generated by a right slant, and ostensibly so, the bodies are mostly located in one region. The diameters were generated by a left-slanted distribution,

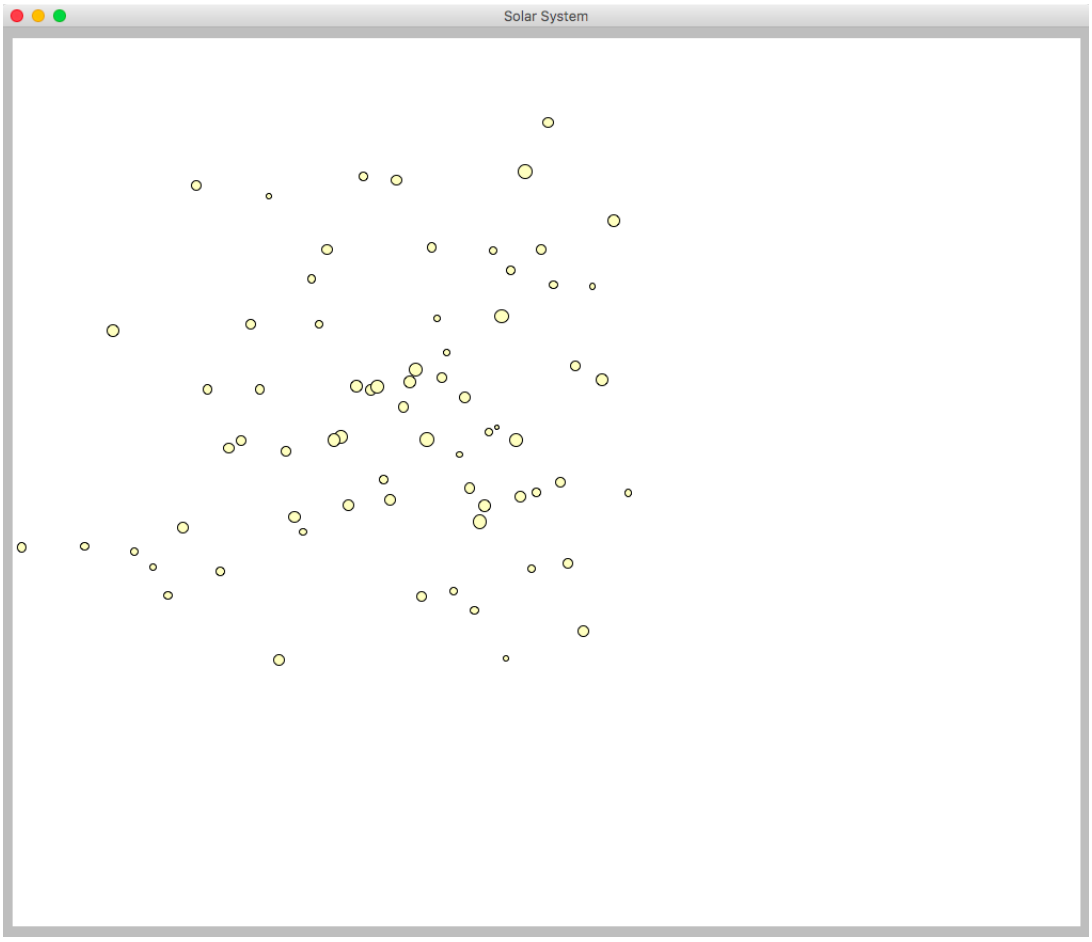


Figure 6.1: Starting Locations of 70 celestial bodies

clearly slanted towards a smaller diameter, with most samples appearing small or medium size, but a notable number appearing fairly large.

The program by nature tracks and draws the motion of each planetary body through time, giving a useful summary of the programs execution by the end of the simulation (665 time steps). Running test case 13 garnered Figure 6.2.

Spending a minute closely observing these paths registers some important observations about the natural behavior of gravity acting on planetary bodies. Some disobeyed their trajectory and were redirected by the large mass in the center. The initial velocities manifested themselves as well. The paths are of all different lengths, which makes sense since they were generated by a uniform distribution.

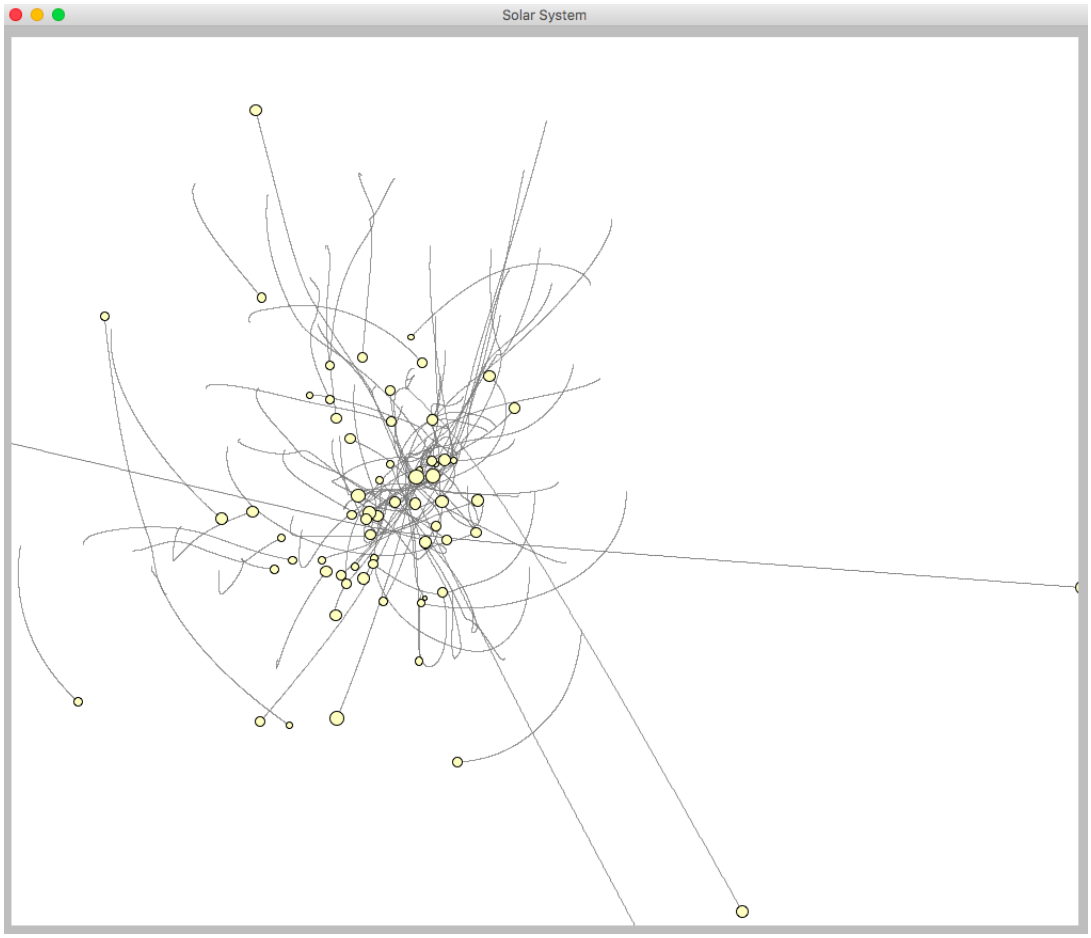


Figure 6.2: Ending Locations of 70 celestial bodies after 665 time steps

6.2 Earthquake Analysis and Visualization Program

The earthquake analysis program performs basic statistical analysis of magnitudes, locations, and depths on a map, and also plots quake events on a map. Bigger dots represent a greater data point value (This is important to note because dots representing magnitudes describe their intensity not their destruction coverage). One test case had a file name (test vector) of:

```
6-70-magnitudes|cardioid-latitudes|left_-
slanted-longitudes|right_slanted-depths|cardioid.csv
```

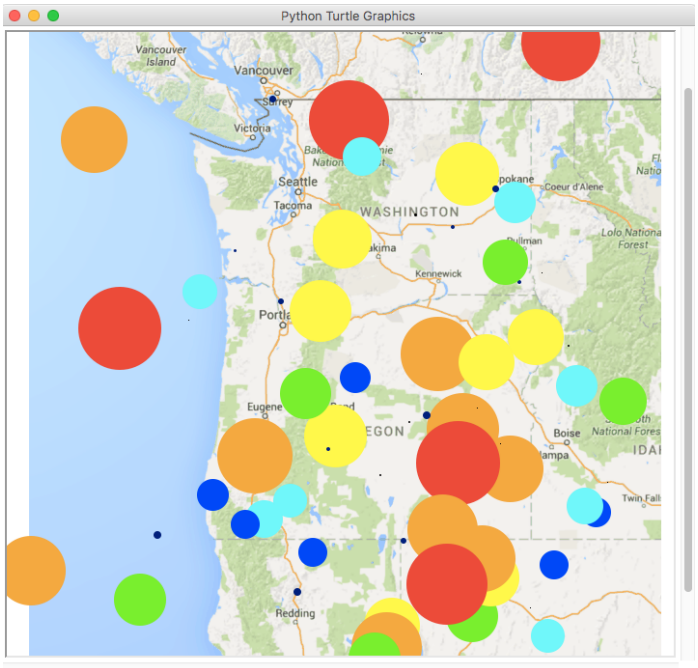
Observing the cardioid relationship between latitudes and longitudes is confirmation of GenSequence working. GenSequence was programmed with the following information:

```
#magnitudes ranges
Micro = Range(0.0, 2.0, exclusive_upper=True)
Feelable = Range(4.5, 7.9, exclusive_lower=True)
Great = Range(8.0, 9.5, exclusive_lower=True, exclusive_upper=True)
#depths ranges
Shallow = Range(0.0, 5.0, exclusive_lower=True, exclusive_upper=True)
Mid = Range(5.0, 15.0)
Deep = Range(15.0, 30.0, exclusive_lower=True)
...
# specify the relationship between magnitudes and depths
MagsDepths = Cardioid(Mags, Depths)
MagsDepths.setFavorites([(Micro,Shallow), (Great,Deep), (Feelable,Mid)])
MagsDepths.setNonFavorites([(Micro,Deep), (Great,Shallow),
    (Feelable,Deep), (Feelable,Shallow)])
```

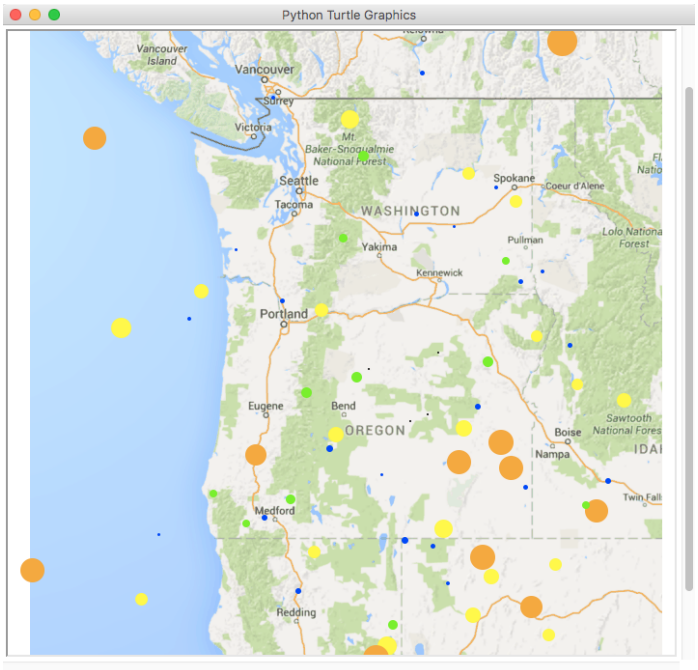
This is the literal implementation of the tool describing the most frequently occurring data pairs in this test case. Low intensity magnitudes should be near Earths surface and very intense magnitudes should occur deep below the earths surface. Moreover, it is not very frequent that low-intensity earthquakes occur very deeply, and high-intensity earthquakes occur very near surface. [1]

[1] This assumption may be entirely false about the nature of earthquakes. This trend is one I invented to add functionality to GenSequence, one that could easily be reversed by any seismologist user of the program.

The plots of magnitudes and depths (Figure 6.3) show this constraint propagating:



(a) Magnitudes



(b) Depths

Figure 6.3: Plotting Magnitudes and Depths from the same test case

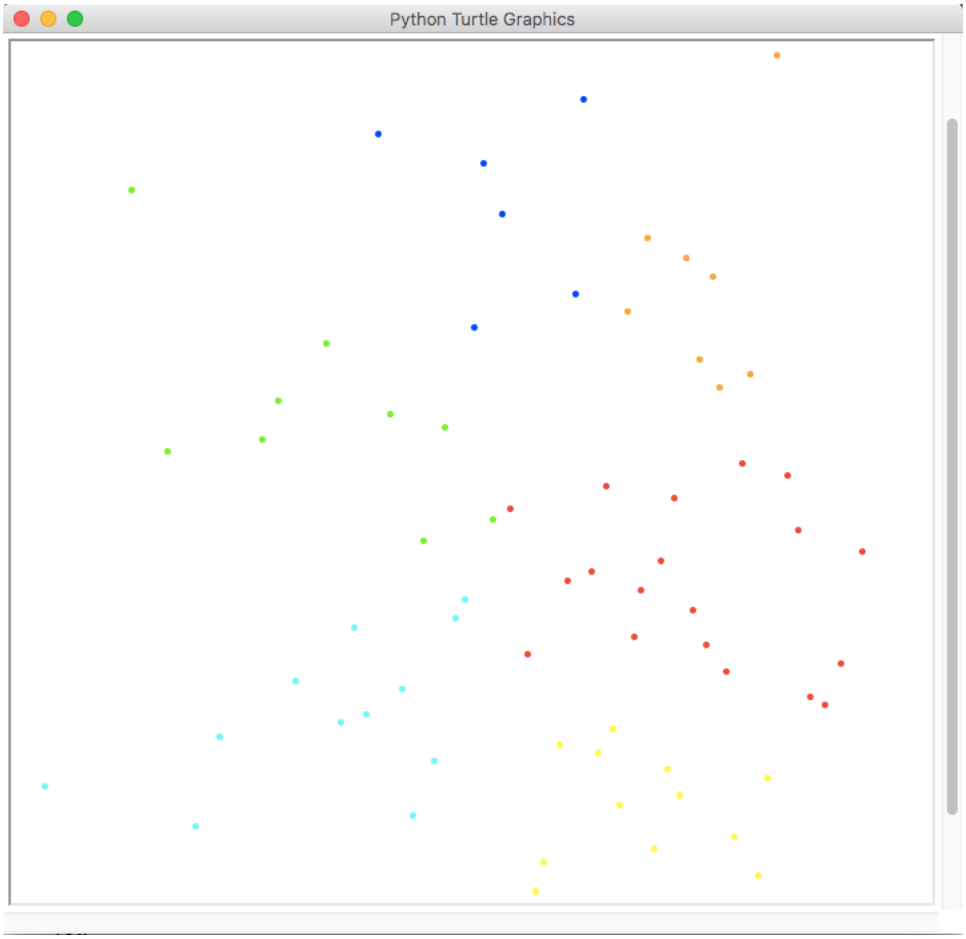


Figure 6.4: K-Means Clustering of Earthquake events from case 13

The correlation is rather difficult to piece but does reflect exactly what is expected. The large red dot due west of Portland has a matching yellow depth dot. The small blue event due west of the Oregon-California border has a matching blue dot in the same place. One outlier is a sizable orange earthquake covering the Umatilla forest that has a very tiny blue depth dot.

Considering latitude-longitude oriented north-up, west-left, east-right, south-down, and given the left-slanted-ness of the latitudes and right-slanted-ness of longitudes, this test case should mark the locations of the earthquakes mostly drifting towards the lower-right corner. This is in fact what the program generates (Figure 6.4).

Chapter 7

Concluding Thoughts

The research in this field has progressed quite a lot, but for many different applications. The application I have built is fairly specialized, and therefore limited in scope. It cannot be used to evaluate specific SUTs; it just does not have the functionality to do so. After all, as one paper noted, "Test data generation is an undecidable problem, meaning that it cannot be completely solved. Nevertheless, this does not mean there is no algorithm that can find a plausible but partial solution to satisfy a specific test goal". The application I have built is just one way to aid in testing a specific type of program.

Database-driven applications literally control the world. The record of every transaction, every payment, every credit report, and every bill can dictate a persons entire life. The infrastructure surrounding that data must not expunge or fabricate any of it, and must maintain its integrity. Software that accesses, controls, and manipulates this huge amount of data carries substantial responsibility. Moreover, software that can read, interpret, and identify trends in a huge wealth of global data has amazing power in informing us of what happens in the world and how we can make better decisions. It is therefore of utmost importance to design that software well enough to trust its results.

GenSequence has addressed those applications. It functions simply but provides automation during the testing stage and generates reliable data that is what is says

it is. If the context of the program warrants consideration of statistically unlikely but possible scenarios, GenSequence can make that happen. It primarily states its power by its ease of use and nearly end-to-end automation.

The simplicity of the programs for which I designed the tool may have been too great for GenSequence to help me find any actual misbehavior. Moreover, any open-source software I generate tests for may not have any flaws. Unfortunately I have no way of knowing if the tool cannot find bugs or if the programs just do not have any. This is the major difficulty with constructing software-testing tools. Numerous articles have declared their victory in finding bugs and attribute it to their testing tool. However, measuring GenSequences power in this way has the potential to return with a resounding no, that GenSequence is not able to do its job. Consequently, my tools ability to find flaws is left inconclusive, but it has potential to change testing processes in a very significant way. GenSequence has addressed the major complaint that creating test data takes too long and that the data is too unreliable.

Bibliography

[1] David Chays, Saikat Dan, Phyllis G. Frankl, Filippos I. Vokolos, and Elaine J. Weyuker. A framework for testing database applications. *SIGSOFT Softw. Eng. Notes*, 25(5):147–157, August 2000.

[2] Don Gotterbarn. The creation of facts in the cloud: A fiction in the making. *SIGCAS Comput. Soc.*, 45(3):60–67, January 2016.

[3] Lawrence Livermore National Laboratory. About VisIt.