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. What an honor it has been to enjoy the time offered by a true polymath. Finally I thank him sincerely for a year of weekly meetings for which significant portions were devoted to chats about bicycles and coffee. 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, her bright attention to detail, and her encouragement for me 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

1 Introduction **1**

2 Literature Survey **5**

3 Proprosed Argument **9**

4 Background on Design Decisions **11**

4.1 Parameters 11

4.2 Pairwise Testing 12

4.3 Context-Free Grammars 14

4.4 Law of Large Numbers 16

4.5 Cardioids 18

4.6 Preprocessing 19

5 Methods **21**

6 Results **23**

6.1 Celestial Body Simulation 23

6.2 Earthquake Analysis and Visualization Program 25

7 Concluding Thoughts **30**

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

1.1 Basic Workflow of a software program 2

3.1 Basic Diagram of User's workflow with GenSequence 10

4.1 A CSV file of a test case input into the Team Builder program 12

4.2 Histogram of point values generated by normal distribution - sample size 100 17

4.3 Histogram of point values generated by normal distribution - sample size 10,000 18

4.4 Histogram of point values generated by normal distribution and reduced to sample size of 300 18

6.1 Starting Locations of 70 celestial bodies 24

6.2 Ending Locations of 70 celestial bodies after 665 time steps 25

6.3 Plotting Magnitudes and Depths from the same test case 27

6.4 K-Means Clustering of Earthquake events from case 13 29

List of Tables

4.1 Single Symbolic Test Vector 13
4.2 Single Concrete Test Case 13

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, the tradeoff of cost and convenience of testing is an important consideration in determining the rigor of testing. Cost is never low, and convenience is never high, so software testing remains an lengthy and sometimes ad hoc practice in many industrial applications. Numerous papers agree that software testing is the most arduous part of software development [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**]. This is not surprising since testing the quality of software is inherently an impossible solution. In fact, Misailovic et. all reports that 50% of the software development process is software testing.

Firstly, testing the software means knowing what the software *should* do and what output it *should* generate. A key component of getting correct results is knowledge of what 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 ?? 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.

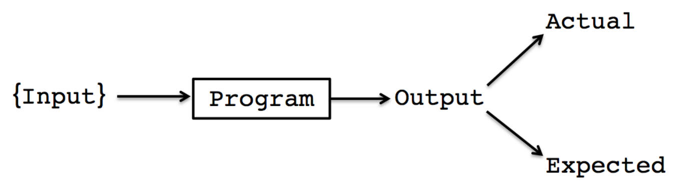


Figure 1.1: Basic Workflow of a software program

Then, she must know how to read or understand the actual result and be able to compare it to the expected.[1] In immensely complex scientific software or finanical programs swathed in accounting terms, the output of the program might be an unknown language to the tester. Even if the tester is a domain scientist, the output may be too complicated to read quickly and identify the trends that indicate she received a postive test result. Lindvall et. al describe this problem in detail when testing autonomous drones for example [?].

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. But these models must be tested. What is known as the "happy path", the operational choice that most

[1]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 exhibits. Even then, it may be impossible to describe the ideal expected output of the program.

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**.

With human's prolific ability to create and store data quantifying the world around them, it is hard to imagine that a tester could not find data that they could use. Gotterbarn agrees that "Insufficient data is not a problem" [**?**]. 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 hypothetical significant event 30 years from now. Data for that particular test case has to be created.

This is **Problem Two** – 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 [**?**], is capable of handling several gigabyte files. No tester wants to or even could write three gigabytes of data points modeling a real-world situation by hand just to satisfy one test case. Nearly every software testing paper agrees that writing data by hand seriously hinders the testing process [**?**, **?**, **?**, **?**]. Software testing is hard and takes a ton of work and that's why people don't do it [**?**, **?**]

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 project 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. Since Kassab et.al reported that 43% of software engineers did not have enough time to test their code before its release date, automation, simplicity, and ease of use are key tenets of this project. Making a tester's job easier is much more likely to help them do it well, and making this tool accessible and easy to use is a low-cost, high-convenience solution that not only codifies a standardized testing procedure but shortens the duration of the testing phase and ensures resilient software before its deployment date.

Chapter 2

Literature Survey

One key problem of software testing is knowing what to expect in the output of any given program. 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 [?]

There are a variety of reasons why the oracle does not exist or cannot 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. It goes without saying that this is even more impossible when there are time and resource constraints on the software development timeline. Consequently, a number of researchers have attempted to automate the creation of oracles [?]. For example, one research group used natural-language processing techniques to extract Java comments and design test cases that generate exceptions [?]. Another group identified and compared outputs of redundancies in software, using them as pseudo-oracles [?]. However, the problem of generating complete, automatic oracles remains unsolved. While my project will not make any attempt to create oracles, it will provide information about the input. The test data will be constructed backwards in a sense. Instead of creating random data and then trying to discover its patterns, I will describe a trend and create data that fits that trend, in the hopes that knowing the input's shape will give meaning to the expected output.

Some researchers have done away almost entirely with creating oracles by developing techniques specifically used on non-testable programs [**?**, **?**, **?**, **?**, **?**]. Metamorphic testing 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).

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 [**?**, **?**, **?**, **?**]. The idea is that if an existing test suite can find artifical bugs in a system, it not only can find real bugs but also inform the tester what parts of the code must be correct in order for the system not to fail. This approach is called mutation testing. When a test suite finds artifical bugs, it determines which variables in a system are most effective in revealing faults, and then present them to the tester to manually define. This makes the construction of an oracle quite a bit easier.

The problem with metamorphic and mutation testing is that, while they skirt automatic creation of an oracle, they require just as much effort to develop as just creating an oracle by hand. Discovering metamorphic relationships requires a high level of implementation knowledge to write. Mutation testing requires a substantial amount of setup work, and still turns to the tester for manually defined expected values. Even though these techniques are considered cutting-edge testing methods, they do not provide any automation and therefore are not usable enough.

The need for automation is why random testing has developed such popularity. Random testing is not new, but it also is not very useful. Neither *purely* random testing nor purely random data generation is strong enough to adequately test all branches of the code, nor does it provide any useful information about the test case itself or the data it contains. Consequently, constrained randomness, or using randomness in conjunction with other techniques, has gained the benefits of randomness' automation with the structure of other established testing methodologies.

The biggest complaint of pure random testing is that of the distribution type typically implemented: uniform distribution, which is why Lu recommends sampling along sample-space partitions non-uniformly [**?**], preferring uneven numbers of "Random Node" selections and "Random Edge" selections. Another generalized technique is that of continous feedback. Arcuri describes Adaptive Random Testing, which monitors the success of the result of a random test case, and then appropriately chooses the next input to be unlike the previous input to attempt to trigger a different result [**?**]. All input choices are random, but directed towards certain trends. Arcuri identified a primary problem with this technique in that it often finds repeat bugs and does not extend to as many situations that pure random test cases otherwise could have found. However, several recent groups similarly use a continuous feedback loops to identify the next test cases but with greater control over random stepping so as to address Arcuri's concerns [**?**, **?**, **?**]. Finally, one group used random grammar derivations limited to a finite size to generate programs to test compilers [**?**]. Again this harnesses the automation of random while limiting its recklessness with additional constraints that guide the random selection.

Applying constraints on random data generation is particularly topical to areas of Machine Learning and Database-Driven applications. A team from Columbia University constructed a framework for generating test data for machine-learning models that produce ranking decisions on likely failability of a network of devices. Not only did their framework take in to account the number of examples and number of attributes of a dataset, it considered appearance of repeat values, missing entries, and partitioned categories of examples [?, ?]. However their tool only implemented uniform distribution and could only produce positive integers. This is a keystone feature that indicates the specialization of their framework, and makes extensibility to other applications of software testing nonexistent without significant refactoring. In regards to database-driven applications, there is a considerable lack of developing in testing. The typical approach of data generation is really quite limited in its power. That data really only passes synactic checks, like matching database schema descriptions, and general "not null, unique, primary, and foreign key constraints" [?], which indicates that the links between data tables makes sense. The data is database-compliant, it does not have any mistakes, but it contains no semantic meaning. It does not exhibit any trends or patterns. The shortcomings of these applications influence the desired features of my project.

Chapter 3

Proprosed Argument

I have designed an approach to these problems and have architected a straightforward 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 framework, 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 a simplistic description 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 local file convenient for the testers use. Figure **??** describes the user interaction with the architecture of GenSequence. Red indicates user-written files. Blue indicates the output of GenSequence, the test data that can then be used for testing. This tool is written in the Python language, which is useful for prototyping the MVP (Minimum Viable Product). I call this entire pipeline GenSequence.

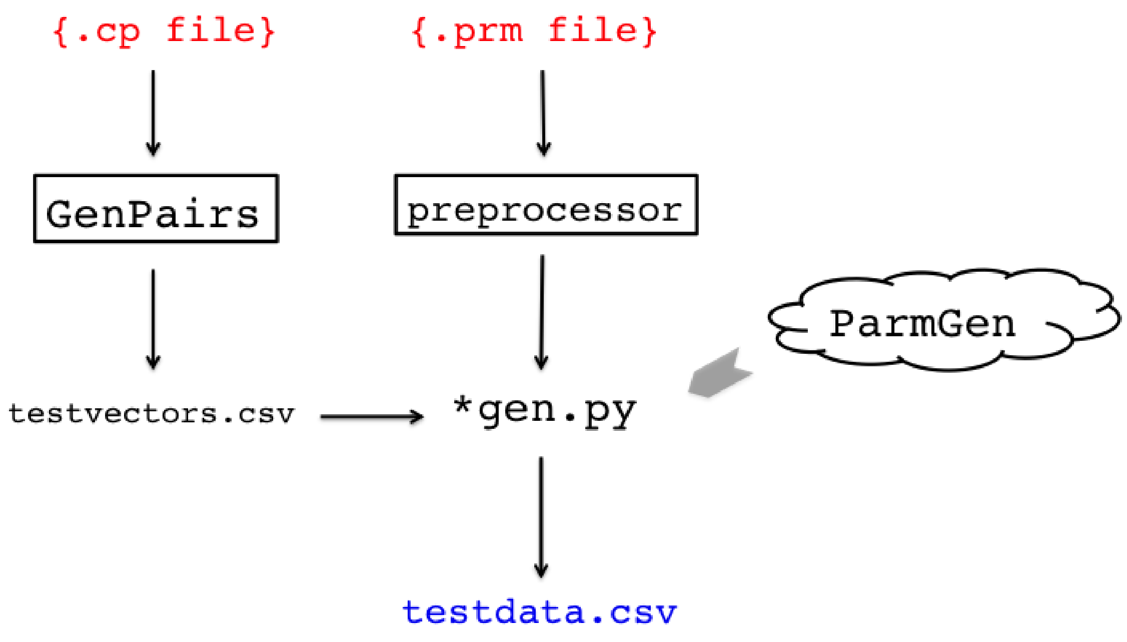


Figure 3.1: Basic Diagram of User's workflow with GenSequence

Chapter 4

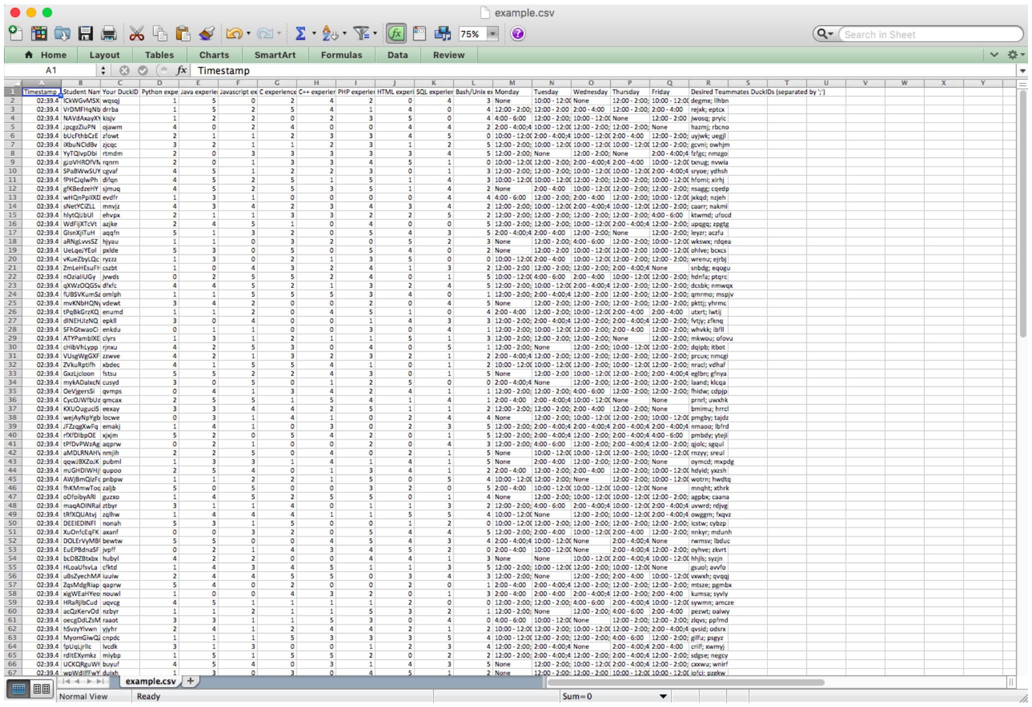
Background on Design Decisions

4.1 Parameters

The “input” to a program is a set of parameters, named variables that represent specific groups of data points. How those parameters are generated, and what literal form they take, effects 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 a certain parameter that really effects the outcome of the program.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
1	Timestamp	Student Nam	Your DuckID	Python expe	Java experie	Javascript ex	C experience	C++ experier	PHP experier	HTML experi	SQL experier	Bash/Unix ex	Monday	Tuesday	Wednesday	Thursday	Friday	Desired Teammates DuckIDs (separated by ';')
2	02:39.4	ICkWGvMSX	wqsqj	1	5	0	2	4	2	0	4	3	None	10:00 - 12:0	None	12:00 - 2:00;	10:00 - 12:0	degmx; llhbn
3	02:39.4	VrDMFHqNb	drrba	1	5	2	5	4	1	4	0	4	12:00 - 2:00;	12:00 - 2:00	2:00 - 4:00	12:00 - 2:00;	2:00 - 4:00	rejxk; eptcx
4	02:39.4	NAVdAxayXY	kisjv	1	2	2	0	2	3	5	0	4	4:00 - 6:00	12:00 - 2:00;	10:00 - 12:0	None	12:00 - 2:00	jwosq; pryic
5	02:39.4	JpcgzZiuPN	ojawm	4	0	2	4	0	0	4	4	2	2:00 - 4:00;4	10:00 - 12:0	12:00 - 2:00;	12:00 - 2:00;	None	hazmj; rbcno
6	02:39.4	bUcFthbCzE	zfowt	2	1	1	2	3	3	4	5	0	10:00 - 12:0	2:00 - 4:00;4	10:00 - 12:0	2:00 - 4:00	12:00 - 2:00;	uyjwk; uegjl
7	02:39.4	iXbuNCldBv	zjcqc	3	2	1	1	2	3	1	2	5	12:00 - 2:00;	10:00 - 12:0	10:00 - 12:0	10:00 - 12:0	12:00 - 2:00;	gcvni; owhjm
8	02:39.4	YyTQlvpDbi	rtmdm	2	0	3	3	3	3	3	4	5	12:00 - 2:00;	None	12:00 - 2:00;	None	2:00 - 4:00;4	fzfgc; nmzgo
9	02:39.4	gJoVHROfVN	rqnrn	2	0	1	3	3	4	5	1	0	10:00 - 12:0	12:00 - 2:00;	2:00 - 4:00;4	2:00 - 4:00	10:00 - 12:0	txnug; nvwia
10	02:39.4	SPaBWwSUY	cgvaf	4	5	1	2	2	3	0	1	3	12:00 - 2:00;	12:00 - 2:00;	10:00 - 12:0	10:00 - 12:0	2:00 - 4:00;4	sryoe; ydhsh
11	02:39.4	fPHCJqlwPh	difqn	4	5	2	5	1	5	1	4	3	10:00 - 12:0	10:00 - 12:0	12:00 - 2:00;	12:00 - 2:00;	10:00 - 12:0	hfomi; xirhj
12	02:39.4	gfKBedzeHY	sjmuq	4	5	2	5	3	5	1	4	2	None	2:00 - 4:00	10:00 - 12:0	12:00 - 2:00;	12:00 - 2:00;	nsagg; cqedp
13	02:39.4	wHQnPplIXD	evdfr	1	3	1	0	0	0	0	4	4	4:00 - 6:00	12:00 - 2:00;	2:00 - 4:00	12:00 - 2:00;	10:00 - 12:0	jxkqd; nzjeh
14	02:39.4	sNetYCiZLL	mnvjz	4	3	4	2	3	4	3	4	2	12:00 - 2:00;	10:00 - 12:0	2:00 - 4:00;4	10:00 - 12:0	12:00 - 2:00;	caarr; nakmi
15	02:39.4	hlytQJJbUl	ehvpx	2	1	1	3	3	2	2	5	2	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	4:00 - 6:00	ktwmd; ufocd
16	02:39.4	WdFijXTcVt	azjke	2	4	5	1	0	4	0	0	5	12:00 - 2:00;	12:00 - 2:00;	10:00 - 12:0	2:00 - 4:00;4	12:00 - 2:00;	upqgq; zpgtg
17	02:39.4	GlsnXjiTuH	aqqfn	5	1	3	2	0	5	4	3	5	2:00 - 4:00;4	2:00 - 4:00	12:00 - 2:00;	None	12:00 - 2:00;	leyzr; aczfu
18	02:39.4	aRNgLvvsSZ	hjyau	1	1	0	3	2	0	5	2	3	None	12:00 - 2:00;	4:00 - 6:00	12:00 - 2:00;	10:00 - 12:0	wkswx; rdqea
19	02:39.4	UeLqeJYEol	pxlde	5	3	0	5	0	5	4	0	2	None	12:00 - 2:00	10:00 - 12:0	12:00 - 2:00	10:00 - 12:0	ohlve; bcxcs
20	02:39.4	vKueZbyLQc	ryzzz	1	3	0	2	1	3	5	0	0	10:00 - 12:0	2:00 - 4:00	10:00 - 12:0	12:00 - 2:00;	12:00 - 2:00;	wrenu; ejrbj
21	02:39.4	ZmLeHEsuFH	cszbt	1	0	4	3	2	4	1	3	2	12:00 - 2:00	12:00 - 2:00;	12:00 - 2:00;	2:00 - 4:00;4	None	snbdg; eqogu
22	02:39.4	nOzialiUGy	jvwds	0	2	5	5	2	4	0	1	5	10:00 - 12:0	4:00 - 6:00	2:00 - 4:00	10:00 - 12:0	12:00 - 2:00;	hdnfa; ptqrc
23	02:39.4	qXWzOQGSv	dfxfc	4	4	5	2	1	3	2	4	5	12:00 - 2:00;	10:00 - 12:0	2:00 - 4:00;4	12:00 - 2:00;	12:00 - 2:00;	dcsbk; nmwqx
24	02:39.4	fUBSVKumSz	omlph	1	1	5	5	5	3	4	0	1	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00	12:00 - 2:00;	12:00 - 2:00;	qmrmo; mspjv
25	02:39.4	mvKNbHQNy	vdewt	3	4	2	0	0	2	0	4	5	None	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	pkttj; yhrmc
26	02:39.4	tPqBkGrzKQ	enumd	1	1	2	0	4	5	1	0	4	2:00 - 4:00	12:00 - 2:00;	10:00 - 12:0	2:00 - 4:00	2:00 - 4:00	utxrt; lwtij
27	02:39.4	dINEHJlzNQ	epkll	3	0	4	0	0	1	4	3	3	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	fvtjy; zfknq
28	02:39.4	SFhGtwaoCi	enkdu	0	1	1	0	0	3	0	4	1	12:00 - 2:00;	10:00 - 12:0	12:00 - 2:00;	2:00 - 4:00	12:00 - 2:00;	whvkk; ibfll
29	02:39.4	ATYPambIXE	clyrs	1	3	1	2	1	1	5	1	3	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	None	12:00 - 2:00;	mkwou; ofovu
30	02:39.4	cHibVhLypp	rjnxu	4	2	5	3	0	4	0	5	1	12:00 - 2:00;	None	12:00 - 2:00;	10:00 - 12:0	12:00 - 2:00;	dqipb; itbot
31	02:39.4	VUsgWgGXF	zzwve	4	2	1	3	2	3	2	1	2	2:00 - 4:00;4	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	prcux; nmcgi
32	02:39.4	ZVkuRptifh	xbdec	4	1	5	5	4	1	0	1	2	10:00 - 12:0	10:00 - 12:0	12:00 - 2:00;	10:00 - 12:0	12:00 - 2:00;	nracl; vdhaf
33	02:39.4	GxzLjcloon	fstsu	5	5	2	2	4	3	0	1	5	None	12:00 - 2:00	10:00 - 12:0	12:00 - 2:00;	2:00 - 4:00;4	eglbn; gfnya
34	02:39.4	mykADalxcN	cusyd	3	0	5	0	1	2	5	0	0	2:00 - 4:00;4	None	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	laand; klcqa
35	02:39.4	OeVjgersSi	qvmps	0	4	1	3	4	2	4	1	1	12:00 - 2:00;	12:00 - 2:00;	4:00 - 6:00	12:00 - 2:00;	12:00 - 2:00;	fhidw; cdpjp
36	02:39.4	CycOJWfbUz	qmcax	2	5	5	1	5	4	1	4	1	2:00 - 4:00	2:00 - 4:00;4	10:00 - 12:0	None	None	prnrl; uwxhk
37	02:39.4	KXUOguciS	eexay	3	3	4	4	2	5	1	1	2	12:00 - 2:00;	12:00 - 2:00;	2:00 - 4:00	12:00 - 2:00;	None	bmimu; hrrcl
38	02:39.4	wejAyNpYgb	locwe	0	3	1	4	1	0	2	4	4	None	12:00 - 2:00;	10:00 - 12:0	12:00 - 2:00;	10:00 - 12:0	pmgby; tajdz
39	02:39.4	JFZzqgXwFq	emakj	1	4	1	0	3	0	2	3	5	12:00 - 2:00;	2:00 - 4:00;4	2:00 - 4:00;4	2:00 - 4:00;4	2:00 - 4:00;4	nmaoo; lbfrd
40	02:39.4	rfXfDIbpOE	xjxjm	5	2	0	5	4	2	0	1	5	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	2:00 - 4:00;4	4:00 - 6:00	pmbdy; yteji
41	02:39.4	tPfDvPWzAg	aqprw	0	2	1	0	0	2	0	4	3	12:00 - 2:00;	4:00 - 6:00	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	qjolc; sgqul
42	02:39.4	aMDLRNAHV	nmjih	2	2	5	0	4	0	2	1	5	None	10:00 - 12:0	10:00 - 12:0	12:00 - 2:00;	10:00 - 12:0	rnzyy; sreul
43	02:39.4	qqwJBXZoJK	pubml	1	3	3	1	4	1	4	1	5	None	2:00 - 4:00	12:00 - 2:00;	12:00 - 2:00;	None	oymcd; mxpdg
44	02:39.4	mJGHDIWHJ	qupoo	2	5	4	0	1	3	4	1	2	2:00 - 4:00	12:00 - 2:00;	2:00 - 4:00	12:00 - 2:00;	10:00 - 12:0	hdyid; yxzsh
45	02:39.4	AWjBmQizFc	pnbpw	1	1	2	2	1	5	0	5	4	10:00 - 12:0	12:00 - 2:00;	None	12:00 - 2:00;	10:00 - 12:0	wotrn; hwdtq
46	02:39.4	fhKMmwToc	zaljb	5	0	5	0	0	3	2	0	5	2:00 - 4:00	10:00 - 12:0	10:00 - 12:0	10:00 - 12:0	None	mnqht; xthrk
47	02:39.4	oDfoibyARI	guzxo	1	4	5	2	5	5	0	1	4	None	12:00 - 2:00;	10:00 - 12:0	10:00 - 12:0	12:00 - 2:00;	agpbx; caana
48	02:39.4	maqAOINRal	ztbyr	3	1	1	4	0	1	1	3	2	12:00 - 2:00;	4:00 - 6:00	2:00 - 4:00;4	10:00 - 12:0	2:00 - 4:00;4	uvwrd; rdjvg
49	02:39.4	tRfXQUAtvj	zqlhw	1	4	4	4	1	1	5	5	4	10:00 - 12:0	None	12:00 - 2:00;	10:00 - 12:0	2:00 - 4:00;4	owggm; fxqvz
50	02:39.4	DEEIEDINFI	nonah	5	3	1	5	0	0	1	2	0	10:00 - 12:0	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	icstw; cybzp
51	02:39.4	XuOnfcEqFK	axanf	0	0	3	2	0	5	4	4	5	12:00 - 2:00;	2:00 - 4:00	10:00 - 12:0	2:00 - 4:00	12:00 - 2:00;	nnkyr; mdunh
52	02:39.4	DOLErVyMBI	bewtw	5	5	0	0	4	5	4	3	4	2:00 - 4:00;4	10:00 - 12:0	None	2:00 - 4:00;4	None	rwmsv; lbduc
53	02:39.4	EuEPBdnaSF	jvpff	0	2	1	4	3	4	5	2	0	2:00 - 4:00	10:00 - 12:0	None	2:00 - 4:00;4	12:00 - 2:00;	oyhve; zkvrt
54	02:39.4	bcDBZBtxbx	hubyl	4	2	2	0	0	4	4	1	3	None	None	10:00 - 12:0	2:00 - 4:00;4	10:00 - 12:0	hhjls; syzjn
55	02:39.4	HLoaUfsvLa	cfktd	1	4	3	4	5	1	1	3	5	12:00 - 2:00;	10:00 - 12:0	12:00 - 2:00;	10:00 - 12:0	None	gsuol; avvfo
56	02:39.4	uBsZyechMA	iuuiw	2	4	4	5	5	0	3	4	3	12:00 - 2:00;	None	12:00 - 2:00;	2:00 - 4:00	10:00 - 12:0	vxwxh; qvqqj
57	02:39.4	ZqsMdgRiap	qaprw	5	4	0	2	0	0	2	0	1	2:00 - 4:00	2:00 - 4:00;4	12:00 - 2:00;	12:00 - 2:00;	12:00 - 2:00;	mtsze; pgmbx
58	02:39.4	xigWEaHYeo	nouwl	1	0	0	4	3	2	0	1	3	2:00 - 4:00	2:00 - 4:00	2:00 - 4:00;4	12:00 - 2:00;	2:00 - 4:00	kumsa; syvly
59	02:39.4	HRaRjlbCud	uqvcg	4	5	1	1	1	1	2	0	0	12:00 - 2:00;	12:00 - 2:00;	4:00 - 6:00	2:00 - 4:00;4	10:00 - 12:0	sywmn; amcze
60	02:39.4	acQzKervOd	nzbyr	1	1	2	1	1	5	3	2	1	12:00 - 2:00;	None	12:00 - 2:00;	4:00 - 6:00	2:00 - 4:00	pezwt; oaiwy
61	02:39.4	oecgDdLZsM	raaot	3	3	1	1	5	3	0	4	0	4:00 - 6:00	10:00 - 12:0	None	12:00 - 2:00;	12:00 - 2:00;	zlqvs; ppfmd
62	02:39.4	hSvzyYlvwn	yjyhr	2	4	1	2	4	4	2	1	2	10:00 - 12:0	12:00 - 2:00;	10:00 - 12:0	12:00 - 2:00;	2:00 - 4:00;4	qvsid; odsrx
63	02:39.4	MyomGiwQz	cnpdc	1	1	1	5	3	3	3	5	4	10:00 - 12:0	12:00 - 2:00;	12:00 - 2:00;	4:00 - 6:00	12:00 - 2:00;	gilfu; psgyz
64	02:39.4	fpUqLjrllc	lvcdk	3	1	3	0	0	1	2	3	4	12:00 - 2:00;	2:00 - 4:00;4	None	2:00 - 4:00;4	2:00 - 4:00	crilf; xwmyj
65	02:39.4	rdltEXymkz	miybp	1	5	1	5	5	2	0	2	2	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	2:00 - 4:00;4	12:00 - 2:00;	sdgse; negcy
66	02:39.4	UCKQRguWH	buyuf	4	5	4	0	3	1	4	3	5	None	12:00 - 2:00;	10:00 - 12:0	2:00 - 4:00;4	12:00 - 2:00;	cxxwu; wnirf
67	02:39.4	wpWdllfFwY	duixh	1	3	0	3	0	4	5	1	2	None	12:00 - 2:00;	12:00 - 2:00	10:00 - 12:0	10:00 - 12:0	iofci; pzgkw

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

Figure **??** 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

GenPairs is a pairwise testing tool [?] that creates 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 amplify a problem. Creating the minimum number of test cases that satisfactorily describe all combinations of possible parameters gives 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 **??**)

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 **??**

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. This is so important because it makes GenSequence a more applicable tool for database-driven applications. If a tester would benefit from giving meaning to the data trend, they can benefit from using the GenSequence framework. This is the semantic sense that is often missing in testing database programs.

Most importantly, Pairwise testing satisfies the need to test "outlier" cases. The very function of pairwise testing ensures that a combination of classifications of parameters occurs at least once in the test suite. This is a test case that the tester might have otherwise forgotten or overlooked. It guarantees robust testing.

4.3 Context-Free Grammars

We can use the ideas of context-free grammar derivations to generate concrete test data. We call this feature Makogram, a tool implemented in the Python templating tool Mako [**?**]. Say a test case has the following production rule:

$\langle \textit{Case} \rangle ::= \langle \textit{Item} \rangle\ \langle \textit{Price} \rangle\ \langle \textit{Delivery Method} \rangle$

What constitutes an item? Item might have a production rule

$\langle Item \rangle$ ::= `'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,

$\langle Price \rangle$ ::= fun { random_choice(8,15) } ( )

Meaning that the data point for the Price parameter would be some number chosen at equal randomness from any number between 8 and 15.

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.

Moreover, the power of grammars makes creating the data very easy. GenSequence provides support for the *grammar* of the output csv files, which can be thought of as a template describing the "language" of a csv file. A derivation is a series of rule substitutions that eventually ends with a string. This "string" belongs to the language described by the grammar, and this string *is* the test data that is usable for testers because the production rules gave functional data points. Makogram was constructed not just to limit the number of substitutions but to reach exactly the desired number. So if a tester asks for 30 examples (30 rows of data), Makogram will hand back exactly 30 rows.

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 - however many the test case needs.[1]

I have yet to rigorously prove this selection scheme, 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 paradigm 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.

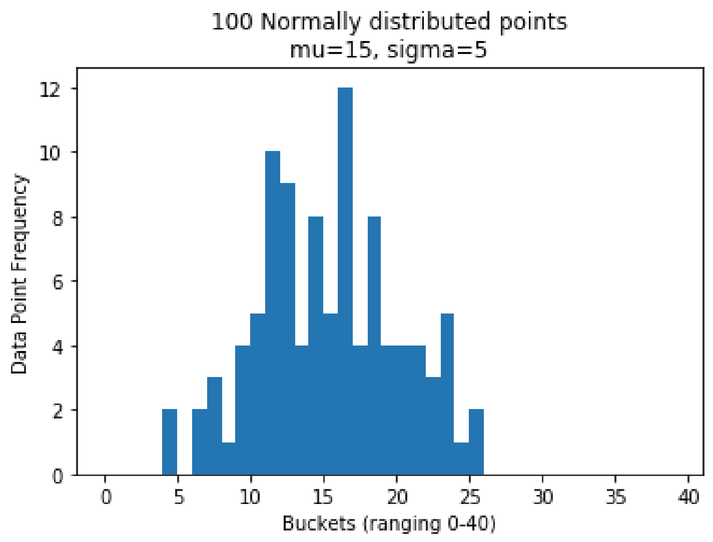


Figure 4.2: Histogram of point values generated by normal distribution - sample size 100

[1]Given the statements of the previous section, some clarification is in order. Context-free grammars would imply that the random function would be called upon every derivation of a data point. But since the number of derivations is limited, that would mean that the number of function calls would be minimized, which would not produce the desired distribution. GenSequence declares the grammar, and then uses what I call a Parm object to create a large sample size and reduce it down into a small pool. When a derivation of the grammar is instantiated, it calls upon the pool, which yields singular points.

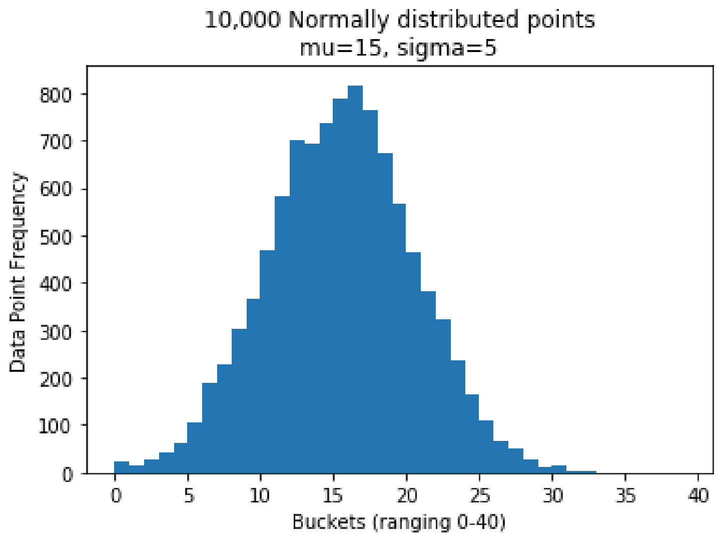


Figure 4.3: Histogram of point values generated by normal distribution - sample size 10,000

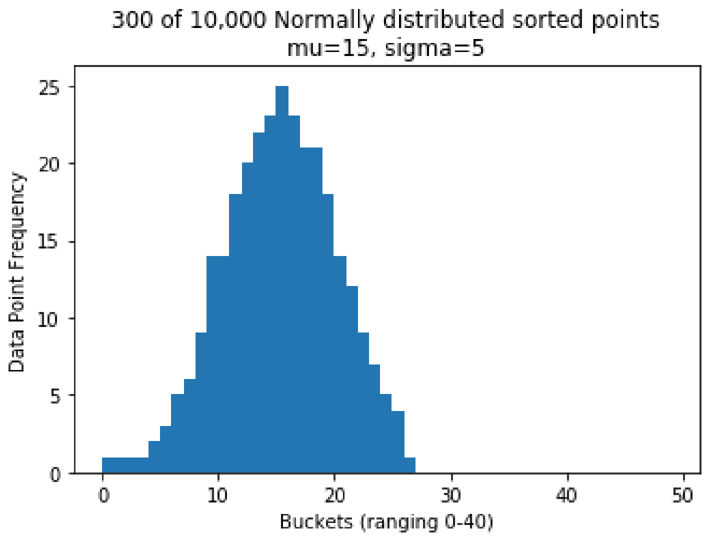


Figure 4.4: Histogram of point values generated by normal distribution and reduced to sample size of 300

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

A Cardioid[2] 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. It does, however, allow testers to generate tests that maintain realistic sense. It also allows the user to include their own test data generation constraints if statistical randomness does not quite 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

[2]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.

significantly increases ease of use. I have fully developed the language into basic syntax and grammar rules, which the tester can then write in a .prm file.

I have also constructed a simple preprocessor to parse this information. 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-Yacc) [**?**]. 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 interpretation. The preprocessor does not yet use Ply to read the entire .prm file, and mostly uses Python file-processing techniques, but the language is simple enough that I had little difficulty rendering the information into the data generator. For an MVP this tactic suffices.

Chapter 5

Methods

One key part of this project is deciding if it accomplished its goal and answered its research questions.

1. Is using random statistical distributions a valid approach to writing test data?
2. Do symbolic test vectors provide enough information about the input that it informs expectations of the output?

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 bugs found through other means of testing or from end user reports. Therefore, that piece of software simply may be bug–free. 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.

In response to Question 2, 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. I will 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). Ultimately it is a tester's decision as to how useful this project is, but these goals become a self-fulfilling prophecy. More features added to the project only increase the ease of use, since a machine does more of the heavy lifting than a human brain and body otherwise would have. Nevertheless, there is further discussion later.

Finally, it is important to determine how valid the random statistical distribution approach is. I will use a few case studies 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. Then I will observe my tools data representation in an earthquake analysis program.

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 is the test case number and 70 is the number of rows in the test case. Each row of the file represents an instance of a planetary body. GenSequence has encoded the way each parameter was generated in the filename. So `mass|right_slanted` means that 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:

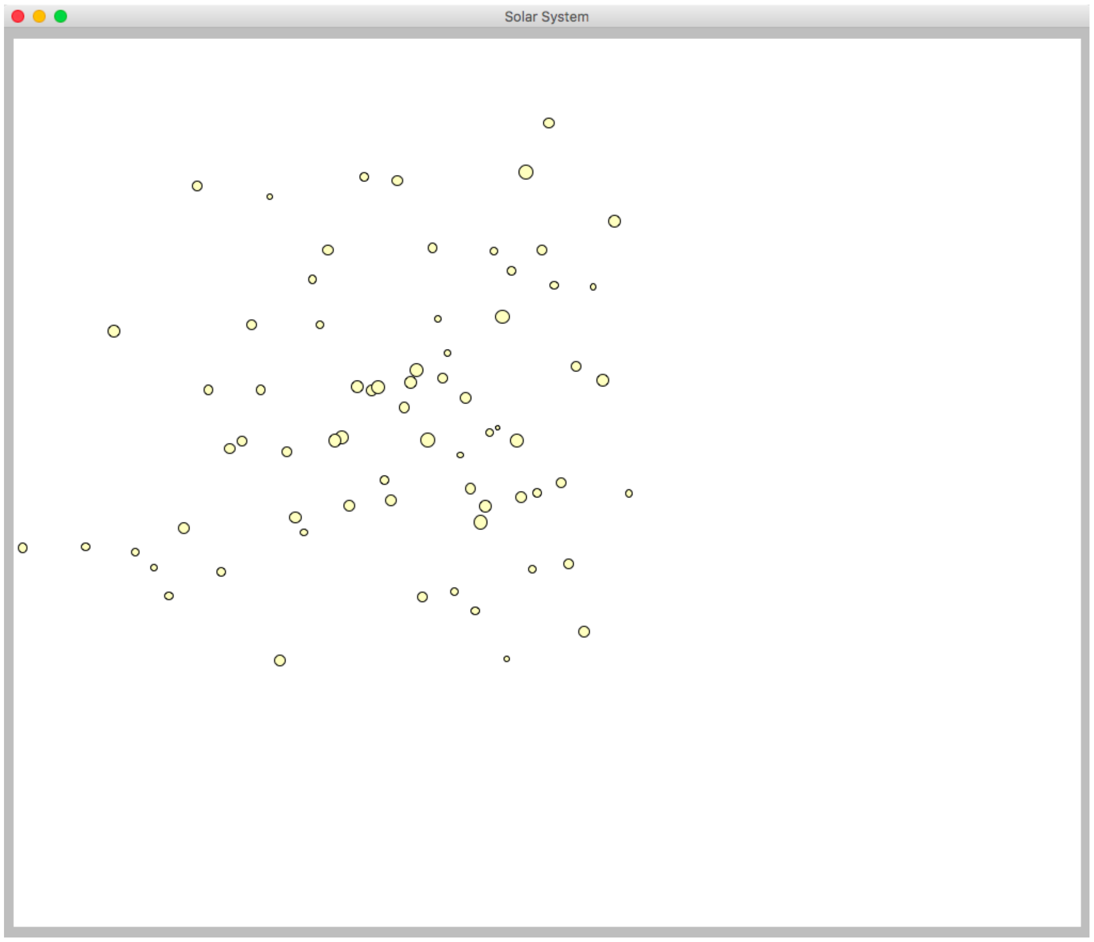


Figure 6.1: Starting Locations of 70 celestial bodies

Figure **??** 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, clearly slanted towards a smaller diameter, with most samples appearing small or medium size, but a notable number appearing fairly large (though from the naked eye one can agree it does require close inspection).

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

Spending a minute closely observing these paths registers some important observations about the natural behavior of gravity acting on planetary bodies. Some

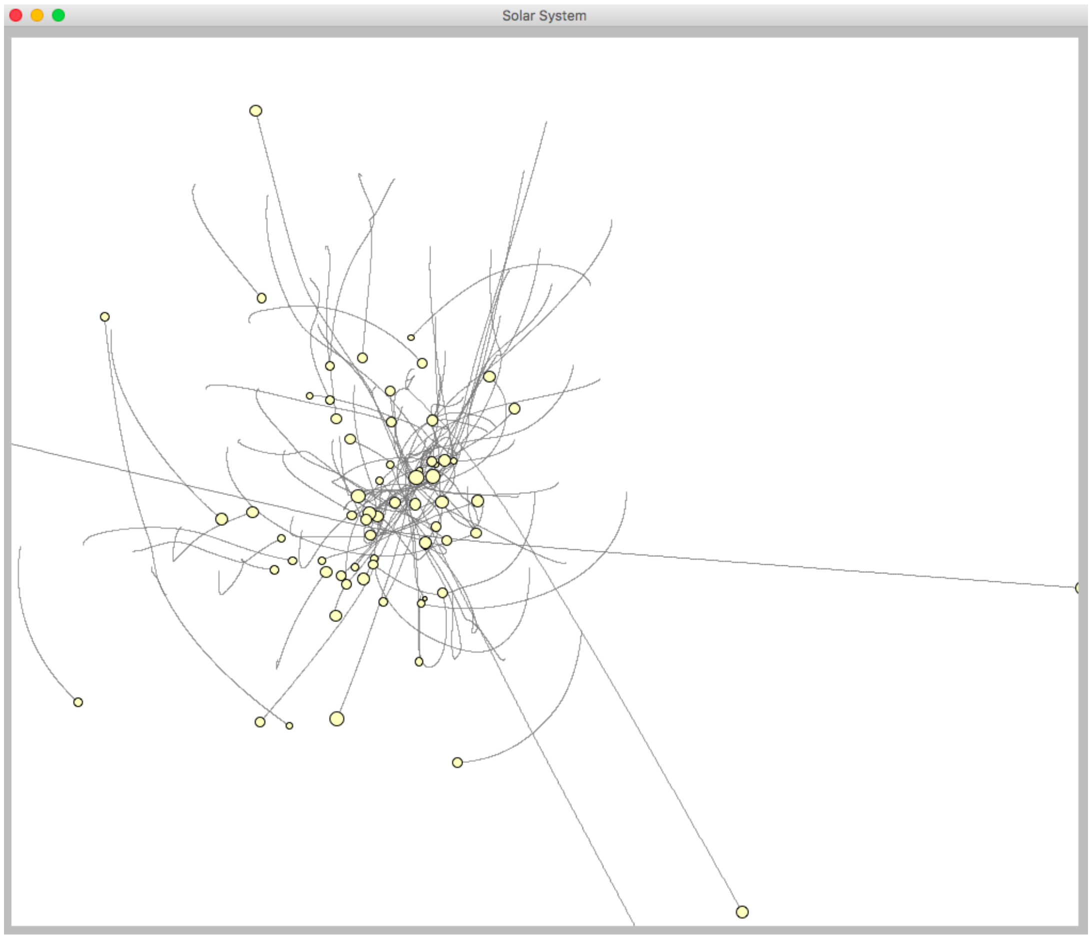


Figure 6.2: Ending Locations of 70 celestial bodies after 665 time steps

disobeyed their trajectory and were redirected by the cumulative 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.

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
```

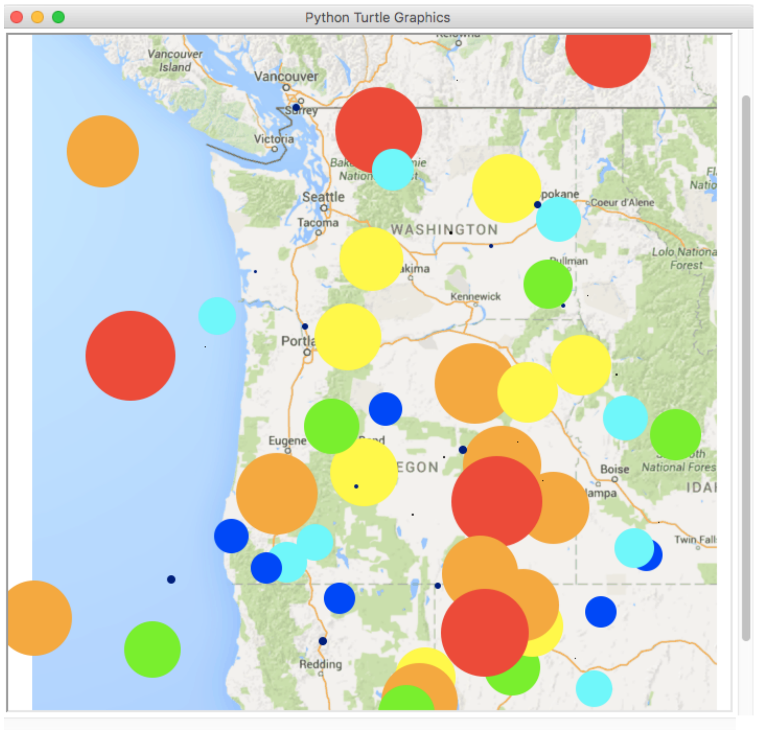
Observing the cardioid relationship between latitudes and longitudes is confirmation of GenSequence working. GenSequence was programmed with the following information:

```
#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)])
```

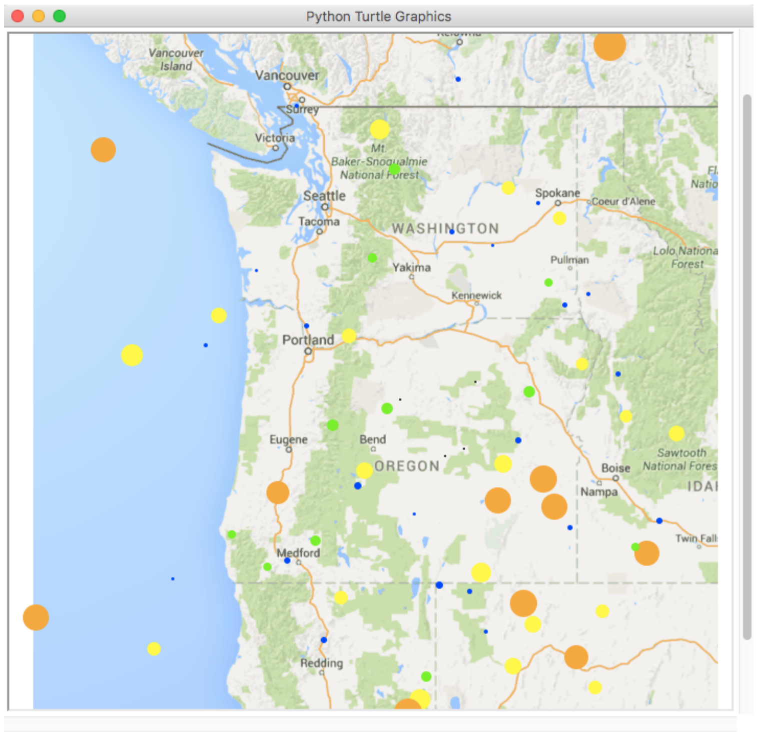
This is the literal implementation of the tool describing the most frequently occurring data pairs in this test case. Low intensity magnitudes should be near Earthss surface and very intense magnitudes should occur deep below the earthss surface. Moreover, it is not very frequent that low-intensity earthquakes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

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

[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.



(a) Magnitudes



(b) Depths

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

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, which means that a high-intensity quake was fairly deep. The small blue event due west of the Oregon-California border has a matching blue dot in the same place, indicating that a low-intensity quake was not that deep below the surface. One outlier is a sizable orange earthquake covering the Umatilla forest that has a very tiny blue depth dot.

If we consider 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, we should expect right-leaning longitudes and left-leaning latitudes. 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 **??**).

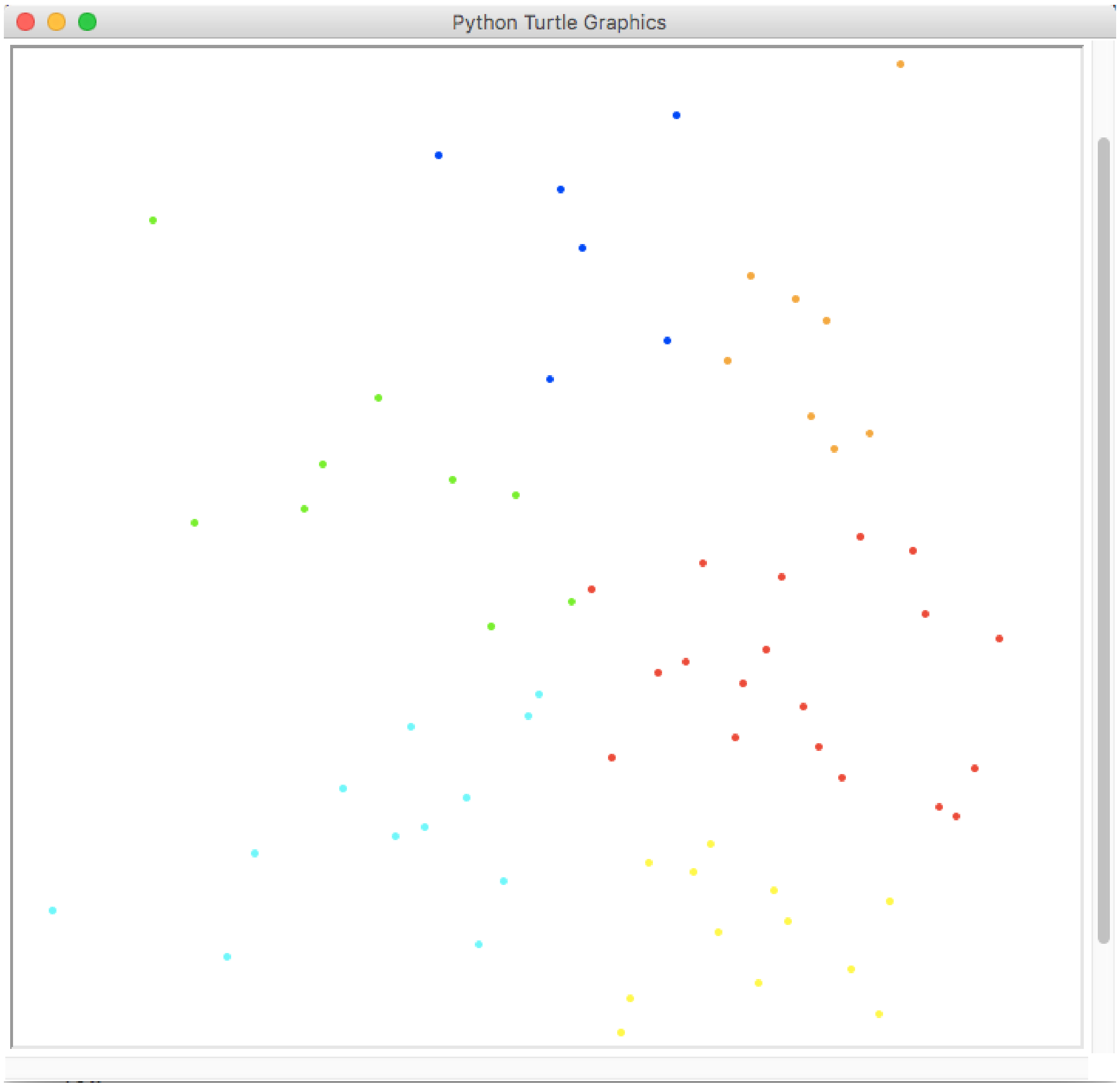


Figure 6.4: K-Means Clustering of Earthquake events from case 13

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.