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.

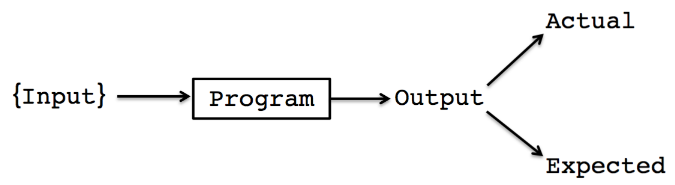


Figure 1.1: Basic Workflow of a software program

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.

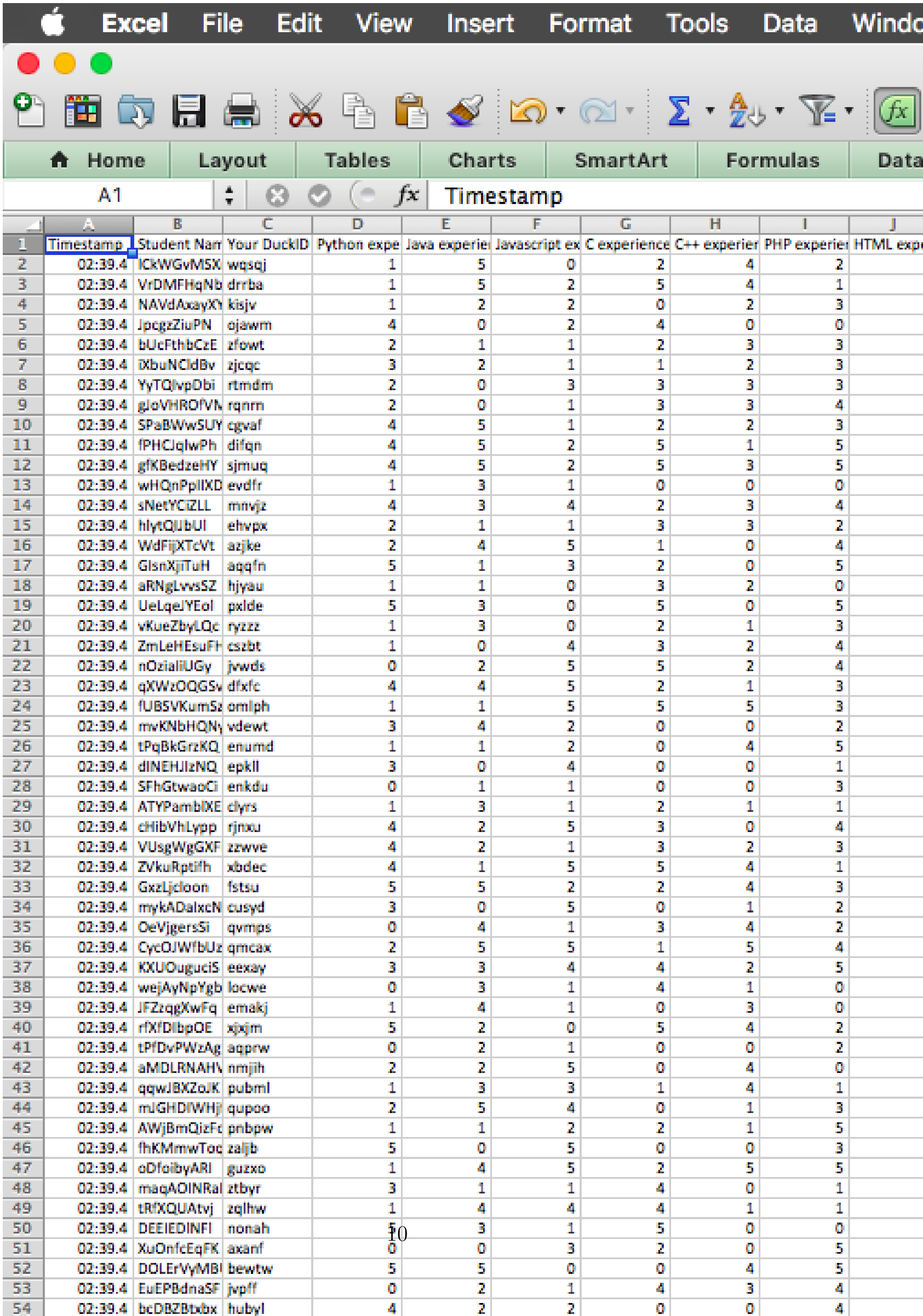
I was inspired to create this tool by testing a project I helped build in my software engineering class, which took in the records of each student in a class of 40 students, describing things like their skills in certain technologies and their available free times during the week. This project made a 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 2 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

more stuf



	A	B	C	D	E	F	G	H	I	J
1	Timestamp	Student Nam	Your DuckID	Python expe	Java experie	Javascript ex	C experience	C++ experier	PHP experier	HTML expe
2	02:39.4	ICkWGvMSX	wqsqj	1	5	0	2	4	2	
3	02:39.4	VrDMFHqNb	drrba	1	5	2	5	4	1	
4	02:39.4	NAVdAxayXY	kisjv	1	2	2	0	2	3	
5	02:39.4	JpcgzZiuPN	ojawm	4	0	2	4	0	0	
6	02:39.4	bUcFthbCzE	zfowt	2	1	1	2	3	3	
7	02:39.4	iXbuNCldBv	zjcqc	3	2	1	1	2	3	
8	02:39.4	YyTQlvpDbi	rtmdm	2	0	3	3	3	3	
9	02:39.4	gJoVHROfVN	rqnrn	2	0	1	3	3	4	
10	02:39.4	SPaBWwSUY	cgvaf	4	5	1	2	2	3	
11	02:39.4	fPHCJqlwPh	difqn	4	5	2	5	1	5	
12	02:39.4	gfKBedzeHY	sjmuq	4	5	2	5	3	5	
13	02:39.4	wHQnPpllXD	evdfr	1	3	1	0	0	0	
14	02:39.4	sNetYCiZLL	mnvjz	4	3	4	2	3	4	
15	02:39.4	hlytQJJbUI	ehvpx	2	1	1	3	3	2	
16	02:39.4	WdFijXTcVt	azjke	2	4	5	1	0	4	
17	02:39.4	GlsnXjiTuH	aqqfn	5	1	3	2	0	5	
18	02:39.4	aRNgLvvsSZ	hjyau	1	1	0	3	2	0	
19	02:39.4	UeLqeJYEol	pxlde	5	3	0	5	0	5	
20	02:39.4	vKueZbyLQc	ryzzz	1	3	0	2	1	3	
21	02:39.4	ZmLeHEsuFH	cszbt	1	0	4	3	2	4	
22	02:39.4	nOzialiUGy	jvwds	0	2	5	5	2	4	
23	02:39.4	qXWzOQGSv	dfxfc	4	4	5	2	1	3	
24	02:39.4	fUBSVKumSz	omlph	1	1	5	5	5	3	
25	02:39.4	mvKNbHQNy	vdewt	3	4	2	0	0	2	
26	02:39.4	tPqBkGrzKQ	enumd	1	1	2	0	4	5	
27	02:39.4	dINEHJlzNQ	epkll	3	0	4	0	0	1	
28	02:39.4	SFhGtwaoCi	enkdu	0	1	1	0	0	3	
29	02:39.4	ATYPamblXE	clyrs	1	3	1	2	1	1	
30	02:39.4	cHibVhLypp	rjnxu	4	2	5	3	0	4	
31	02:39.4	VUsgWgGXF	zzwve	4	2	1	3	2	3	
32	02:39.4	ZVkuRptifh	xbdec	4	1	5	5	4	1	
33	02:39.4	GxzLjcloon	fstsu	5	5	2	2	4	3	
34	02:39.4	mykADalxcN	cusyd	3	0	5	0	1	2	
35	02:39.4	OeVjgersSi	qvmps	0	4	1	3	4	2	
36	02:39.4	CycOJWfbUz	qmcax	2	5	5	1	5	4	
37	02:39.4	KXUOuguciS	eexay	3	3	4	4	2	5	
38	02:39.4	wejAyNpYgb	locwe	0	3	1	4	1	0	
39	02:39.4	JFZzqgXwFq	emakj	1	4	1	0	3	0	
40	02:39.4	rfXfDlbpOE	xjxjm	5	2	0	5	4	2	
41	02:39.4	tPfDvPWzAg	aqprw	0	2	1	0	0	2	
42	02:39.4	aMDLRNAHV	nmjih	2	2	5	0	4	0	
43	02:39.4	qqwJBXZoJK	pubml	1	3	3	1	4	1	
44	02:39.4	mJGHDIWHj	qupoo	2	5	4	0	1	3	
45	02:39.4	AWjBmQizFc	pnbpw	1	1	2	2	1	5	
46	02:39.4	fhKMmwToc	zaljb	5	0	5	0	0	3	
47	02:39.4	oDfoibyARI	guzxo	1	4	5	2	5	5	
48	02:39.4	maqAOINRal	ztbyr	3	1	1	4	0	1	
49	02:39.4	tRfXQUAtvj	zqlhw	1	4	4	4	1	1	
50	02:39.4	DEEIEDINFI	nonah	5	3	1	5	0	0	
51	02:39.4	XuOnfcEqFK	axanf	0	0	3	2	0	5	
52	02:39.4	DOLErVyMBI	bewtw	5	5	0	0	4	5	
53	02:39.4	EuEPBdnaSF	jvpff	0	2	1	4	3	4	
54	02:39.4	bcDBZBtxbx	hubyl	4	2	2	0	0	4	

Chapter 5

This is the Title of the First Chapter

This is a sample document for the Auburn LaTeX style-files known as `aums` (for Master's papers) and `auphd` (for Ph.D.'s). The appendix contains some of the history of this project, including contact information for the authors. Site administrators should upgrade to LaTeX2e; however, the style files should work with the older LaTeX.

The style files should be available on mallard. The current release is available by anonymous ftp to ftp.dms.auburn.edu in the directory aums (on-campus computers may also retrieve these from `http://www.dms.auburn.edu/manuals`). Most users will need either Lamport's book [?] or Hahn's book [?].If you do not need the List of Abbreviations, comment the nomencl package and associated nomenclature commands.

Theorem 5.1 *This is an example theorem.*

5.1 This is an example of a section heading

This is some text which follows the section heading. You can find the data in Table 5.1.

5.1.1 This is a subsection heading

Text after the subsection. And we have a figure, Figure 6.1.

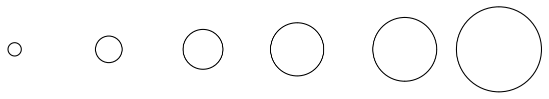


Figure 5.1: Hollow circles 1

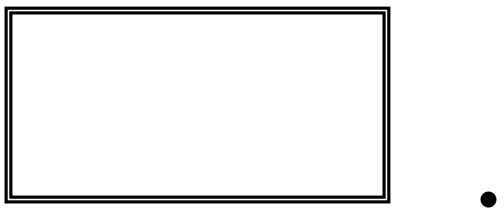


Figure 5.2: Some TikZ picture.

	Multicolumn Heading 1		
Heading 1	Heading 2	Heading 3	Heading 4
1	19, 20 (19.5)	NA	NA
3	∞* (∞)	18, 15 (16.5)	9, 9 (9)
5	23, 18 (20.5)	16 (16)	7, 7, 8 (7.33)
*Some random comment for the whole table.			

Table 5.1: Some Table of data

Chapter 6

New Chapter

Theorem 6.1 *Another theorem.*

Figure 6.1: Hollow circles

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.