USING STATISTICAL DISTRIBUTIONS TO GENERATE RANDOM TEST DATA

Jamie Zimmerman
Thesis Defense
May 25, 2018

Abstract

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.

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 -> 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 – misbehavior in a piece of software, 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

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

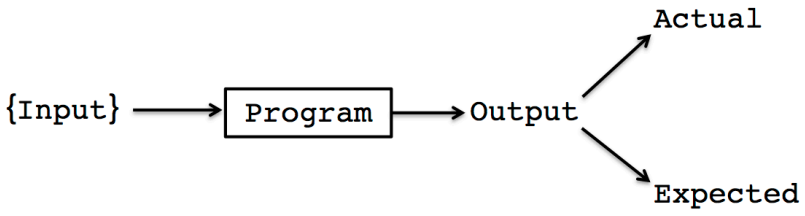


Figure 1: basic workflow of a software program

spend *too* much of its lifetime under test. Therefore, cost and convenience of testing is an important consideration in determining the rigor of testing.

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.

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.

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. But 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. This 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 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[i], 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"[ii], 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.

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"[iii]

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[iv] [v] [vi]. 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[vii]. 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[viii]. An alternative solution identified chains of dependent variables using probabilistic substitution graphs to find the most important variables[ix]. 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.[x]

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.[xi]

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[xii].

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"[xiii]. This is essentially generating data that is database compliant but has little semantic sense.

Proposed 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. I thank Michal for his construction of that tool that exists as the primary first step in this pipeline of testing automation. 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 tester's 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.

Background – Design Decisions

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.

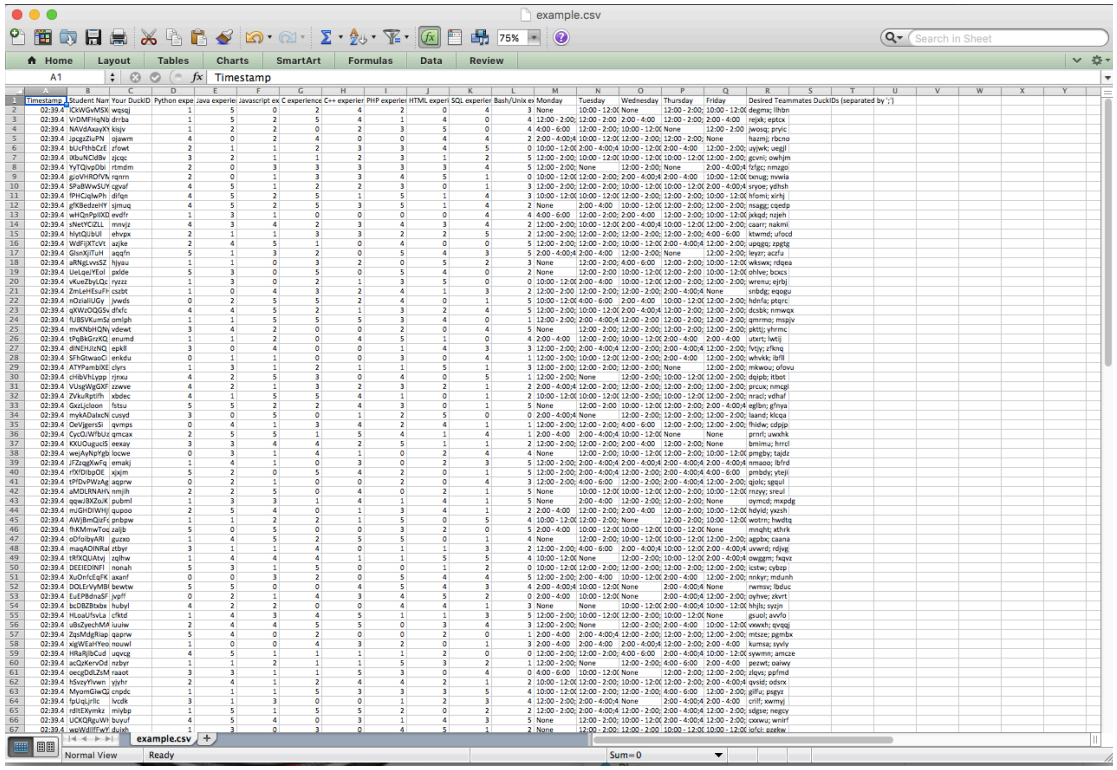


Figure 2 shows the CSV file of the literal input into the Team Building program. The software takes in the filename, opens the file, and parses the data. Each row is a student, and each column is a parameter, such as free time availability on Monday or self-rated HTML experience level.

2. Pairwise Testing

Using a pairwise testing tool developed by the author's advisor[xiv], 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:

Item Purchased	Price	Delivery Method
Large Item	Expensive	Ultrafast

Table 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 end up with

Motorboat	18700.00	Jet Airplane

Table 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 company's 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.

2. Context-Free Grammars

The aforementioned skipped step uses context-free grammars implemented in the Python templating tool Mako to generate concrete test data. This tool, Makogram, has been developed by the author's advisor[xv]. Say a test case has the following production rule:

Case --> Item + Price + DeliveryMethod

What constitutes an item? Item might have a production rule

Item --> "Ship" or *"Book"* or *"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)])

Meaning that the data point for the Price parameter would be some number chosen at equal randomness from any number between 1 and x.

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.

3. 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[xvi]. 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 will approach the expected outcome as the sample size grows to infinity. So in general, the greater the sample size the more accurate the results.

I expect my use case to arrive on the order of hundreds of data points, so I believe a default sampling magnitude 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 every many the test case needs.

This selection scheme is yet to be proven rigorously, but I believe 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

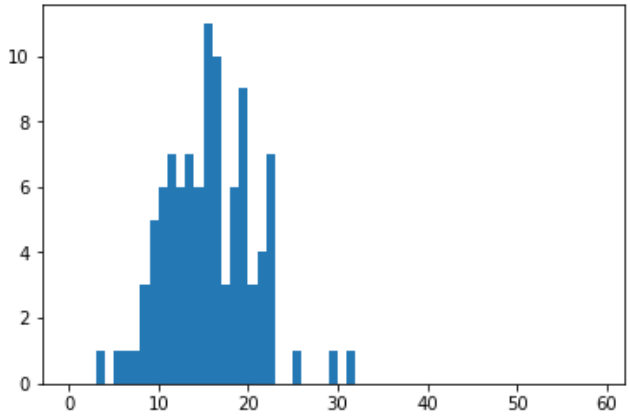


Figure 3 shows the histogram of points generated by normal distribution with a sample size of 100

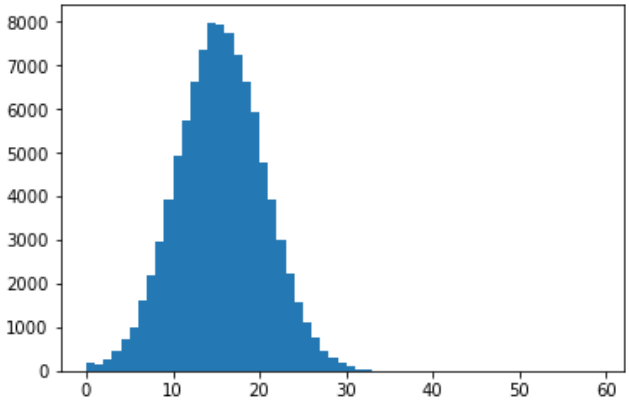


Figure 4 shows the histogram of points generated by normal distribution with a sample size of 10,000

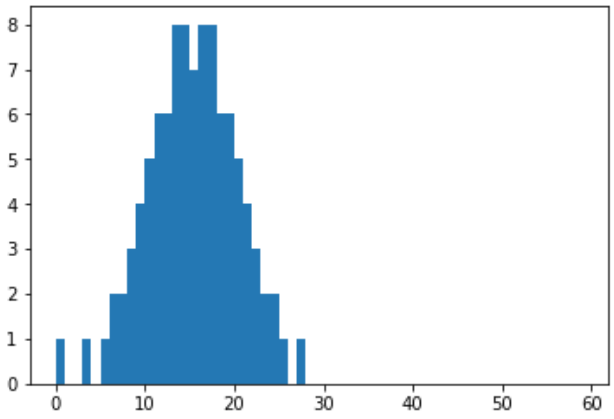


Figure 5 shows the histogram of points generated by normal distribution and reduced down to a sample size of 300.

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

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 tester's 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. 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 parameter's data was generated helps clarify the expected result. First I will test my testing tool against a planetary orbits simulation. 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 properties: 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).

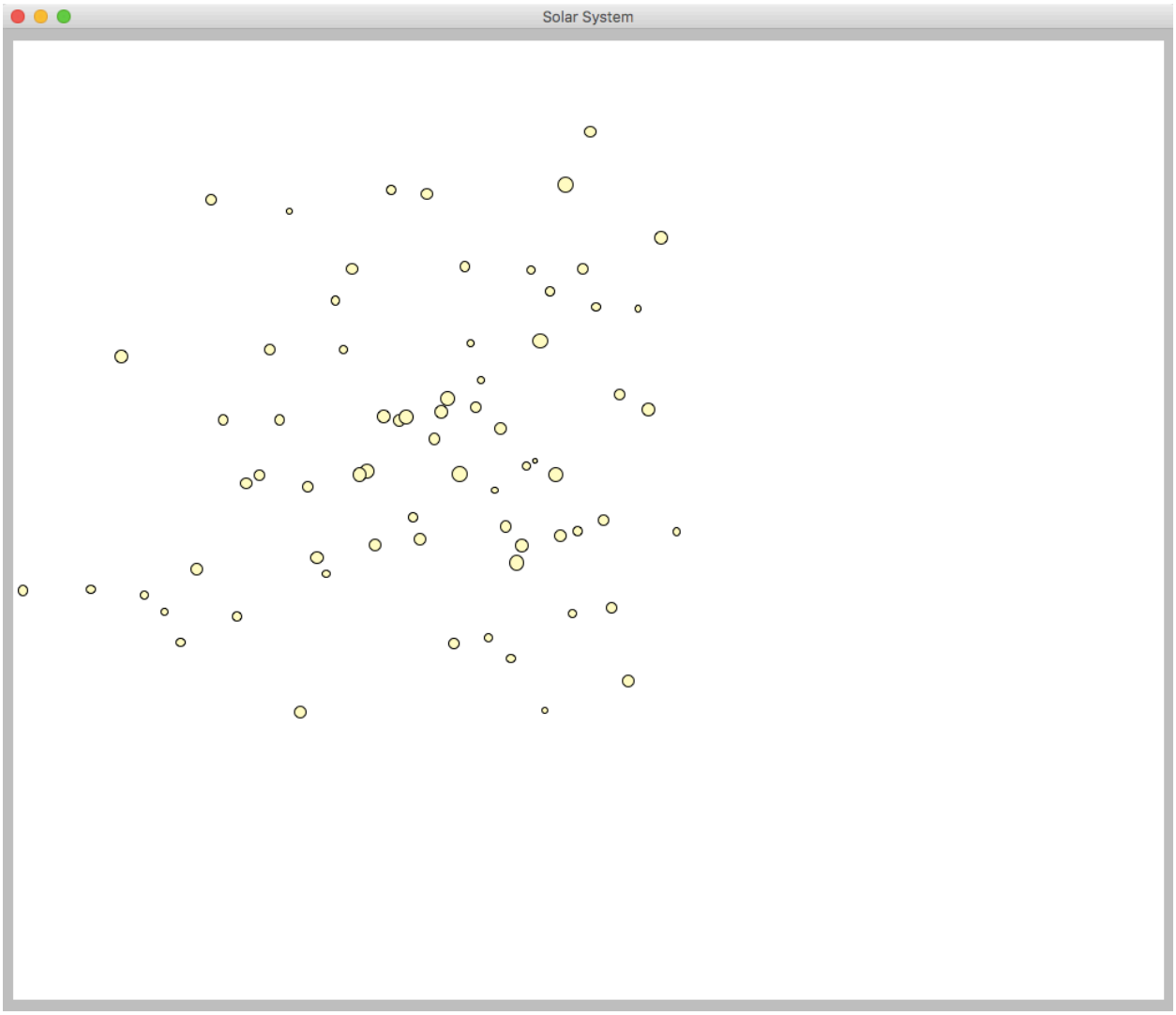
Results

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
```

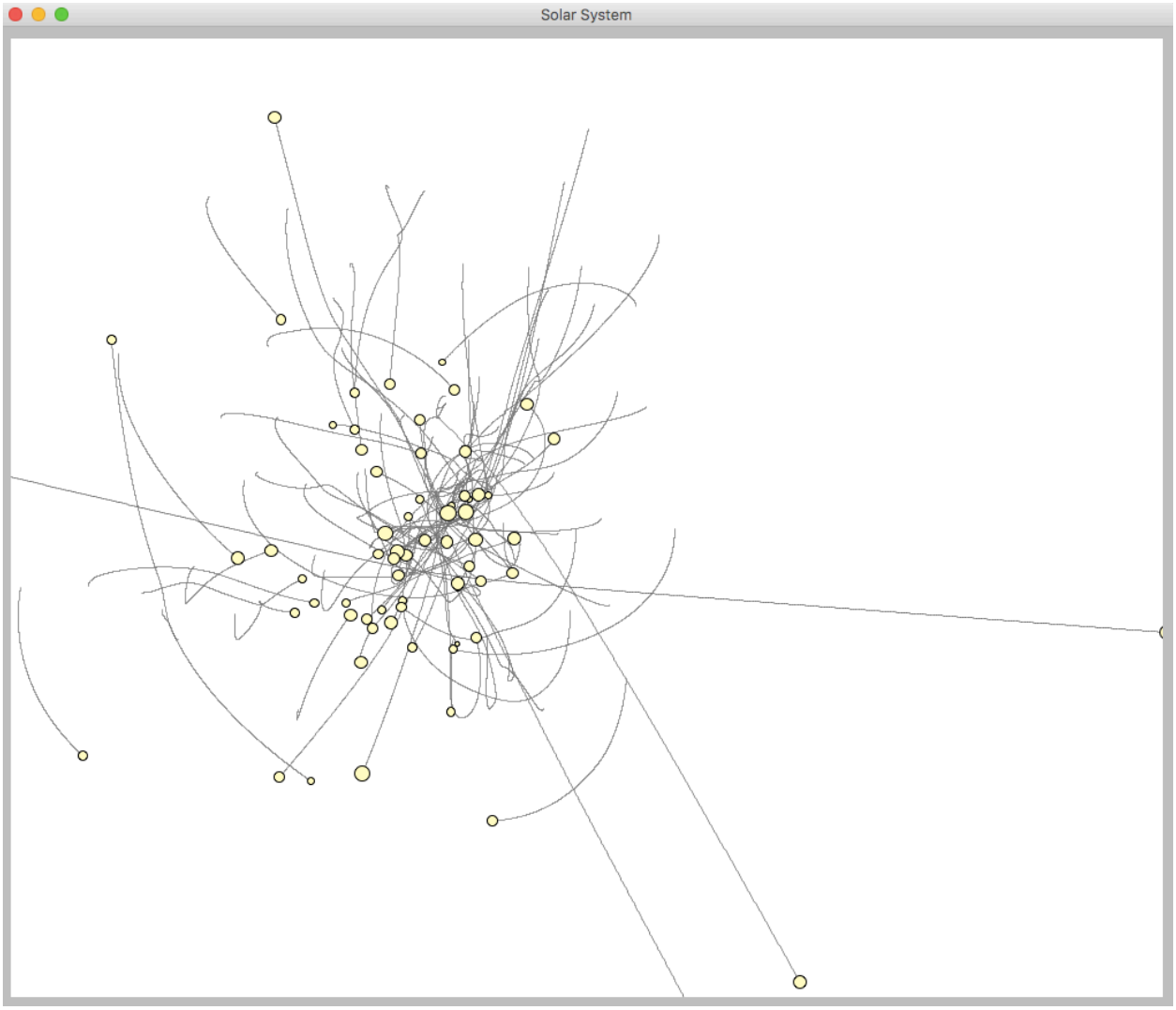
13 describes the test case number, 70 describes the number of rows in the test case. In this simulation, each row represents an instance of a planetary body.

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:



This picture 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.

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 this output:



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.

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"[xvii]. 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. One person's entire life can be dictated by the record of every transaction, every payment, every credit report, and every bill. The infrastructure surrounding that data must not expunge, lose, or fabricate any of it, and must maintain its integrity. Software that accesses, controls, and manipulates this huge amount of data has a lot of responsibility on it. 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 filled the need for trustworthy niche software. 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.

Annotated Bibliography

Barr, E.T, and M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey," in *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507-525, May 1 2015. DOI: 10.1109/TSE.2014.2372785
URL: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6963470&isnumber=7106034

This paper describes different testing techniques and methods of parsing an oracle.

Chen, Junjie, and Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. 2016. Supporting oracle construction via static analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ASE 2016). ACM, New York, NY, USA, 178-189. DOI: https://doi.org/10.1145/2970276.2970366

Static collection of fault-finding variables is a variant of the method that Staats, Gay, and Heimdahl proposed.

Chen, Tsong Yuen. 2015. Metamorphic testing: a simple method for alleviating the test oracle problem. In *Proceedings of the 10th International Workshop on Automation of Software Test* (AST '15). IEEE Press, Piscataway, NJ, USA, 53-54.

A discussion on metamorphic testing practices and applications.

Gotterbarn, Don. 2016. The creation of facts in the cloud: a fiction in the making. SIGCAS Comput. Soc. 45, 3 (January 2016), 60-67. DOI: http://dx.doi.org/10.1145/2874239.2874248

A review of the veracity and completeness of data stored on the cloud, and best practices for securing the privacy and integrity of public cloud platforms.

Guttag, John. *Introduction to Computation and Programming Using Python: With Application to Understanding Data*. Cambridge, MA: MIT Press, 2017, pp. 156-157.

Bernoulli proved that the Law of Large Numbers guarantees certain statistical predictions.

Haller, Klaus. “The test data challenge for database-driven applications.” In *Proceedings of the Third International Workshop on Testing Database Systems* (DBTest '10). ACM, New York, NY, USA, Article 6. DOI=http://dx.doi.org/10.1145/1838126.1838132

A review of the current state of research in generating data for database-driven applications. Most commercial tools that take advantage of random automation still require a lot of specification from human testers.

Jahangirova, Gunel. 2017. Oracle problem in software testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA 2017). ACM, New York, NY, USA, 444-447. DOI: https://doi.org/10.1145/3092703.3098235

A discussion on the benefits of oracle location: inside or outside the source code.

Lawrence Livermore National Laboratory. "VisIt". Simulation and Computer Codes. https://wci.llnl.gov/simulation/computer-codes/visit

VisIt is an open-source visualization software that can visualize big data and provide tools for scientific visualization.

Lindvall, Mikael, and Adam Porter, Gudjon Magnusson, and Christoph Schulze. 2017. Metamorphic model-based testing of autonomous systems. In *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, USA, 35-41. DOI: https://doi.org/10.1109/MET.2017..6

Murphy, Christian, Gail Kaiser, and Marta Arias. "Parameterizing random test data according to equivalence classes". *In Proceedings of the 2nd International Workshop on Random Testing* (ASE '07). IEEE Press, Atlanta, GA, USA, 38-41. DOI: http://doi.acm.org/10.1145/1292414.1292425

This group created a framework that applies constraints on randomly generated test data to feed into machine-learning models that rank likelihood of sensor failure in an industrial setting. Their data was limited to only numerical values and only made use of uniform distribution.

Patrick, Matthew, Matthew D. Castle, Richard O. J. H. Stutt, and, Christopher A. Gilligan. "Automatic Test Image Generation using Procedural Noise". *In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ASE 2016). ACM, New York, NY, USA, 654-659. DOI: https://doi.org/10.1145/2970276.2970333

Using Perlin noise was necessary as a constraint to generate realistic images representing the infectious spread of insects in Britain.

Segura, Sergio, and Amador Durán, Javier Troya, and Antonio Ruiz Cortés. 2017. A template-based approach to describing metamorphic relations. In *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, USA, 3-9. DOI: https://doi.org/10.1109/MET.2017..3

Since metamorphism is thriving, this group attempts to standardize the practice with a template so that practitioners who begin using the technique have a baseline approach that makes the transition easier.

Souza, Francisco Carlos M., Mike Papadakis, Yves Le Traon, and Márcio E. Delamaro. "Parameterizing random test data according to equivalence classes". *9th International Workshop on Search-Based Software Testing* (2016). ACM, New York, NY, USA, 38-41. DOI: http://dx.doi.org/10.1145/2897010.2897012

This group used a search-based backtracking technique to find potential data points that would successfully kill mutants, artificial bugs, in a program, the hypothesis being that a test suite that can find fake bugs can find real bugs just as well.

Staats, Matt, and Gregory Gay, and Mats P. E. Heimdahl. 2012. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, USA, 870-880.

By seeding faults in mutant versions of a program, an existing test suite can find the variables most likely to cause faults in a program. By narrowing the size of the expected oracle values, the tester has significantly less work.

Weyuker, Elaine J. On Testing Non-Testable Programs, *The Computer Journal*, Volume 25, Issue 4, 1 November 1982, Pages 465–470, https://doi.org/10.1093/comjnl/25.4.465

This paper defines what it means to be non-testable, and clearly describes the oracle problem and why it is so difficult to provide every expected value in an oracle.

Young, Michal. GenPairs. Github repository, 2017. https://github.com/TestCreator/GenPairs.

Michal built a pairwise testing tool that takes in descriptions of a parameter's possibilities and creates a multitude of test vectors, symbolic descriptions of the necessary test cases.

Zimmerman, Jamie, and Michal Young. GenSequence. Github repository, 2018. https://github.com/TestCreator/GenSequence

We built a tool that works in conjunction with GenPairs. It takes the symbolic test descriptions and generates concrete data points for each test case.

---

[i] "VisIt," Simulation and Computer Codes, Lawrence Livermore National Laboratory. https://wci.llnl.gov/simulation/computer-codes/visit

[ii] Don Gotterbarn, "The Creation of Facts in the Cloud – a fiction in the making", SIGCAS Computing Society, September 2015. http://dx.doi.org/10.1145/2874239.2874248

[iii] Weyuker, Elaine J. "On Testing Non-Testable Programs". *The Computer Journal* Volume 25, Issue 4 (1 November 1982): 465-470, accessed October 22, 2017 https://doi.org/10.1093/comjnl/25.4.465

[iv] Mikael Lindvall, Adam Porter, Gudjon Magnusson, and Christoph Schulze. "Metamorphic model-based testing of autonomous systems". *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, 35-41, (2017).

[v] Sergio Segura, Amador Durán, Javier Troya, and Antonio Ruiz Cortés. "A template-based approach to describing metamorphic relations". *Proceedings of the 2nd International Workshop on Metamorphic Testing* (MET '17). IEEE Press, Piscataway, NJ, 3-9, (2017).

[vi] Tsong Yueh Chen. "Metamorphic testing: a simple method for alleviating the test oracle problem". *Proceedings of the 10th International Workshop on Automation of Software Test* (AST '15). IEEE Press, Piscataway, NJ, 53-54 (2015).

[vii] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz and S. Yoo, "The Oracle Problem in Software Testing: A Survey". *IEEE Transactions on Software Engineering* Volume 41, No. 5 (May 2015): 507-525.

[viii] Matt Staats, Gregory Gay, and Mats P. E. Heimdahl. "Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing". *Proceedings of the 34th International Conference on Software Engineering* (ICSE '12). IEEE Press, Piscataway, NJ, 870-880, (2012).

[ix] Junjie Chen, Yanwei Bai, Dan Hao, Lingming Zhang, Lu Zhang, Bing Xie, and Hong Mei. "Supporting oracle construction via static analysis". *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (ASE 2016). ACM, New York, NY, 178-189, (2016).

[x] Christian Murphy et al. "Parameterizing Random Test Data According to Equivalence Classes," *In Proceedings of the 2nd International Workshop on Random Testing* (2007): 38-41. doi: http://doi.acm.org/10.1145/1292414.1292425

[xi] Matthew Patrick et al. "Automatic Test Image Generation using Procedural Noise," *In Proceedings on the 31st IEEE/ACM International Conference on Automated Software Engineering* (2016): 654-659.

[xii] Francisco Carlos M. Souza et al. "Strong Mutation-based Test Data Generation Using Hill Climbing," *In Proceedings of the 9th International Workshop on Search-Based Software Testing* (2016): 45-54.

[xiii] Klaus Haller. "The test data challenge for database-driven applications," In Proceedings of the 3rd International Workshop on Testing Database Systems (2010): Article 6.

[xiv] Michal Young, GenPairs, (2017), Github repository, https://github.com/TestCreator/GenPairs.

[xv] Jamie Zimmerman and Michal Young. GenSequence, (2018), Github repository, https://github.com/TestCreator/GenSequence

[xvi] John Guttag, *Introduction to Computation and Programming Using Python: With Application to Understanding Data* (Cambridge, MA: MIT Press, 2017), 156-157.

[xvii] Francisco Carlos M. Souza et al. "Strong Mutation-based Test Data Generation Using Hill Climbing," *In Proceedings of the 9th International Workshop on Search-Based Software Testing* (2016): 45-54.