Developing Empirical Skills in an Introductory Computer Science Course

David Reed
Department of Mathematics and Computer Science
Creighton University
DaveReed@creighton.edu

Abstract

This paper describes an introductory computer science course that emphasizes empirical skills as well as programming and computer science breadth. Designed to attract both non-majors and potential computer science majors, the course utilizes JavaScript in a Web-based environment, allowing students to learn the basics of programming quickly and also to take advantage of familiar and intuitive GUI interfaces. In completing online laboratory assignments, students study interdisciplinary applications and learn to form testable hypotheses, design and conduct experiments, and analyze the results. Through interdisciplinary examples and experimentation, students not only develop critical thinking skills but also learn to apply computing to other areas of study.

Introduction

Empirical skills are playing an increasingly important role in the computing profession and our society. For students pursuing a career in computing, the ability to design, execute, and analyze experiments is central to tasks such as evaluating the validity of hardware and software systems. While claims based on benchmarks, test suites, and usability studies are common in the computing field, the evidence for such claims is not always well grounded. Training in experimental methodology exposes aspiring computer scientists to potential flaws in conclusions based on empirical evidence and teaches how to compensate for them. Practicing experimental methodology within the computing context also provides the best learning situation for computer science majors since they can perform "sanity checks" on conclusions based on their prior knowledge and expertise. In the wake of the increasing importance of empirical investigation, computing professionals have called for more training in experimentation [6, 9, 11].

Beyond the computing profession, empirical skills benefit students in all disciplines. In daily life, we are constantly confronted with scientific, economic, and social claims that are based on empirical studies. Developing a basic understanding of empirical methods enables students to evaluate such claims and their relevance to everyday life. Since empirical methods involve both quantitative and analytical thinking, students develop skills in these areas and gain practice presenting and explaining results. Finally, exposure to experimental methods, especially in the context of computing, can be directly beneficial in a number of disciplines. Computers are used extensively as research tools, especially in the natural and social sciences where systems modeling and data analysis require an understanding of both computing technology and the scientific method.

In 1999, Craig Miller, Grant Braught and I began an initiative at Dickinson College to integrate experimentation into the computer science curriculum [7]. As is the case with programming and problem-solving skills, we claimed that it is unrealistic to expect students to be able to develop effective empirical skills in a single course or even a single year. Instead, we proposed a systematic, integrated approach where students are introduced to experimental concepts early and revisit those concepts throughout the curriculum. Early on, students would learn by performing experiments, analyzing the results, and perhaps most importantly, discussing their conclusions. By the end of the process, they would be capable of forming testable hypotheses, designing and conducting experiments, and presenting conclusions based on the results.

This paper describes an introductory computer science course developed as part of this empirical initiative (see [8] for further details). Designed to attract both non-majors and potential computer science majors, the course utilizes JavaScript in a Web-based environment. While programming is the central activity in the course, programs are often presented as tools for experimentation in interdisciplinary applications. As such, students master fundamental empirical concepts and obtain practical experience in applying experimental methods to real-world problems. For those students who continue in computer science, this course builds a foundation for the further development of empirical skills throughout the curriculum.

Course Format

Introductory computer science courses have generally focused on either programming depth (e.g., [1, 4, 5]) or computing breadth (e.g., [2, 10]). The choice of JavaScript in this course, with its flexible syntax and familiar Web-based interface, allows for a more balanced approach. Using a simple subset of the JavaScript language, the course is able to provide enough programming depth to develop problem-solving skills and an appreciation of the algorithmic core of computer science. And since the language is easier to learn than full-featured languages such as C++ and Java, roughly 35% of class time can still be devoted to a survey of computing topics. Throughout the course, interdisciplinary applications and experimentation serve to connect programming and the broader field of computing.

Variants of this course have been taught at Dickinson College since Fall 1998 and at Creighton University since Spring 2001. While these variants differ in some ways (e.g., the Dickinson course has weekly closed laboratories while the Creighton course does not), they have the same basic format. Programming concepts are introduced and implemented using a series of online programming tutorials. Programming skills are then applied to problem solving using online laboratory assignments, which also emphasize critical thinking and experimentation. Finally, the breadth of computer science is presented in the form of readings and class discussions on a variety of computing topics.

Programming Tutorials

In order to be responsive to the individual needs of beginning programmers, the programming component of the course emphasizes self-paced, interactive learning over traditional lectures. Students are introduced to new programming concepts through a series of online tutorials. Each tutorial contains explanatory text, examples, and exercises for applying new concepts and techniques. While the emphasis is on developing programming and problem-solving skills, many exercises have experimental components as well. For example, the tutorial on loops includes exercises where the students simulate dice rolls and verify statistical properties of the roll distribution (such as the likelihood of sevens versus twos). Another exercise involves simulating repeated drawings of a Pick 4 Lotto, and thus demonstrating just how unlikely is that a specific sequence of numbers will be drawn.

Online Laboratories

Students apply their programming skills to solving a wide variety of problems using online laboratory assignments. Interdisciplinary applications are frequently chosen to demonstrate the relevance of computing to other fields of study such as biology, physics, and economics. Modeled loosely on laboratories in the natural sciences, most lab assignments emphasize empirical concepts and involve experimentation, requiring students to form hypotheses about complex systems, design and conduct experiments, and analyze their results. Example lab assignments are described in the next section.

Breadth Topics

The breadth component of the course focuses on topics that help students to understand computer technology and its impact on society. Class periods are scheduled throughout the semester for researching and discussing topics such as the structure of the Internet, the history of computers, and ethical issues in computing. Some of these topics involve experimentation using online applications. For example, Grant Braught, has developed a collection of resources for exploring the internal workings of a computer [3]. Over the course of the semester, students experiment with data representation, circuit design, data flow and the ALU, and program translation using interactive applications in a Web browser.

Laboratory Examples

In addition to providing practice in the design and implementation of programs, lab assignments emphasize empirical concepts and the scientific method of experimentation. Since the empirical aspect of this course is most apparent in these laboratories, a description of representative lab assignments is given below.

Random Letter Sequences

Early lab assignments emphasize the use of existing programs as tools for supporting or refuting hypotheses about complex systems. In the first lab, students are asked to estimate the total number of 4-letter words in the English language. Initial guesses can range from a few hundred to many thousands. To obtain a reasonable estimate, a more scientific method is required.

Since this may be their first exposure to experimental methods, the instructor first leads the students through the process of estimating the number of 3-letter words. It is noted that there are $26^3 = 17{,}576$ different 3-letter sequences. Using a Web page that generates random letter sequences (see Figure 1), each student generates 100 random 3-letter sequences and counts the number of real words that appear. If 3 of those sequences turn out to be words, then that student will estimate the ratio of words to sequences to be 3/100, and thus the number of 3-letter words to be 527. Of course, given only 100 letter sequences each, the counts obtained by individual students can very greatly and thus produce disparate estimates. By averaging the counts obtained by all of the students, however, the resulting estimate is usually quite close to 550 (the number of 3-letter words in the UNIX dictionary).

Once the students understand the experimental method, they are then asked to repeat the process to estimate the number of 4-letter words. Finally, a related question is posed for their consideration: *If the choice of letters in the random sequences were limited to only the most commonly used letters, how would that affect the likelihood of obtaining real words?* Students must state a hypothesis and then use the page to conduct experiments to refute or support that hypothesis (see Figure 2).

This assignment demonstrates several key concepts that will be constantly revisited throughout the course. First is the idea that real-world, non-trivial problems can be modeled and solved using computer programs. While the approach to solving this problem can be understood independent of computers, its implementation would be tedious and unwieldy without the program for generating and reviewing random letter sequences. This realization can help to motivate students as they learn the (often frustrating) details of programming. From an empirical perspective, students are introduced to the idea that random events can have statistical predictability over the long run. This can be counter-intuitive to students, who often assert, "Since it's random, you can't predict anything." This and later assignments clearly demonstrate that the distribution of certain random events can be predicted and used in problem solving (using so-called Monte Carlo methods). Finally, this assignment provides a first look and appreciation for the "Law of Large Numbers". While the estimate obtained using only 100 random letter sequences is questionable, the estimate obtained when you combine the data from 20 to 30 students (totaling 2,000 to 3,000 sequences) can be quite accurate.

Monte Carlo π

A similar laboratory assignment involves the use of a Monte Carlo method for approximating the value of π. Using basic geometry, it can be shown that the ratio of the area of an inscribed circle to the area of a square is π/4. Knowing this, it is possible to approximate the value of π by generating random points in a square and keeping track of how many of those points lie within the inscribed circle. For example, suppose you generated 1000 random points in the square, 800 of which landed inside the inscribed circle. From this data, you could estimate that the area of the circle is 800/1000 or 80% of the area of the square. Since the actual ratio of the areas is known to be π/4, solving for π produces the approximation 3.2.

Using a Web page for generating random points (see Figure 3), students are able to conduct repeated experiments to estimate the ratio of the two areas. The visual nature of the page is appealing to many students and further demonstrates the ability of computer programs in modeling complex systems. This lab also begins to integrate experimentation with programming, as the students must write a simple program that takes their experimental data and produces an approximation for π.

This assignment reinforces many of the empirical concepts that were introduced in the first lab. Once again, it demonstrates that computer programs can be used to solve problems using data generated by random events. This assignment also emphasizes the distinction between consistency and accuracy. Further demonstrating the "Law of Large Numbers", students note that repeated experiments using a small number of random points (say 100) can produce estimates that differ significantly, whereas repeated experiments using a large number of points (say 10,000) will generally produce consistent results. A formal measure of consistency, the relative difference between the most extreme value and the average, is introduced for quantifying this concept.

Likewise, students note that the approximations of π are more accurate (compared to the actual value of π) as more and more points are generated.

Turtle Graphics & Random Walks

As the students develop more programming expertise, laboratory assignments further integrate programming with experimentation. After they have learned about function calls, a laboratory assignment provides a simple Turtle Graphics environment fro drawing figures. Using a combination of JavaScript code and function calls to control the turtle, the students are able to experiment and draw various shapes on the screen. For example, they must determine the sequence of steps necessary to draw a triangle (move forward and turn 120 degrees, three times) and a square (move forward and turn 90 degrees, four times), and then generalize these answers to arbitrary N-sided polygons (move forward and turn 360/N degrees, N times). Such tasks involve extensive trial-and-error to see if proposed solutions work and making proper adjustments when they do not.

The idea of a random walk is introduced in the context of Brownian motion, although applications from biology and computer graphics can be used as motivation as well. Using the provided Turtle Graphics environment, students are able to program a simulation of a random walk and verify the seemingly random distribution of walks on the screen (see Figure 4). While a theoretical result concerning the expected distance attained by a random walk of N steps is known, the final distance squared should equal N, this result is tedious to verify experimentally using the Turtle Graphics page. Since a large number of repetitions is required for accuracy (again, the Law of Large Numbers), a separate Web page is provided for conducting this experiment (see Figure 5).

As was the case with the random letter sequences assignment, a new question is then posed that requires the student to present a hypothesis and then conduct experiments to support or refute that hypothesis: *If the random walk were constrained so that turns can only be made at right angles (i.e., 90^o, 180^o, 270^o, or 360^o), how would that affect the expected distance of a random walk?*

Random Sentences

Throughout the course, the role of experimentation in the testing and debugging of programs is emphasized. After students learn about function definitions, they complete a lab assignment involving grammar rules. For example, the following grammar rules describe simple English sentences composed of a noun phrase followed by a verb phrase. Optional parts of speech are possible in both the noun phrase and verb phrase, so sentences of different lengths are possible.

sentence ← nounPhrase + verbPhrase
nounPhrase ← article + optional(adjective) + noun
verbPhrase ← verb + optional(nounPhrase)

As part of the lab assignment, students must write a program that generates sentences, with individual functions for randomly generating each of the parts of speech. Before writing such a program, however, they must first study the grammar rules and make predictions about the types of sentences that might be generated by those rules. For example, they must recognize that the shortest possible sentence using the above grammar rules contains three words, while the longest possible sentence contains seven words. Similarly, if optional parts of speech are expected to appear 50% of the time, then N randomly generated sentences would be expected to contain ¾*N adjectives. Predictions such as these can then be used to help test and debug their program as they write it.

Slot Machine

With careful planning, even traditional programming assignments can contain an empirical component. After learning about conditionals and dynamic images in a Web page, students complete a lab assignment in which they write an interactive program for simulating a slot machine (see Figure 6). In addition to designing and implementing the program, students also analyze the likelihood of winning at slots and verify their analysis through experimentation. For example, assuming there are three slots and each slot can display one of four random images, then there is a 1/16 chance of a spin producing three identical images. If the payoff on a win is less than 16 times the cost of playing, then the odds are against the player. Students perform this analysis and verify the long-term performance of the player given different payoff schemes.

2-Dimensional Random Walks

Late in the course, students are presented with more open-ended lab assignments. Instead of a specific sequence of exercises, students are given a problem to solve or system to model, and must design programs and experiments on their own. For example, one lab revisits the concept of a random walk, only now constrained to one dimension. The analogy is that of an inebriated person standing in the middle of a narrow alley. With each step, the person can stagger towards either exit. As in the earlier random walk lab, the students are given a theoretical result concerning 1-dimensional random walks: to reach a goal distance of N requires N^2 steps on average. In order to verify this result experimentally, students must design and implement a program for simulating such walks and collect statistics on the number of steps (see Figure 7).

Following the pattern developed in earlier labs, a new question is then posed requiring the student to formulate a hypothesis and then design experiments to test that hypothesis: *If it is a dead-end alley with only one exit, how does this affect the expected number of steps required to exit, assuming steps that bounce up against the wall still count as steps?* Students must form a hypothesis and present a plausible justification for that hypothesis (e.g., it will require more steps than in an unconstrained walk since steps up against the wall count but are ineffective). They must then modify their random walk program to simulate such constrained walks and conduct experiments to either support or refute their hypothesis (see Figure 8).

Outcomes

Since this course was introduced at Dickinson College in the fall of 1998, student reaction has been very positive. Student evaluations suggest that the balance between breadth and depth has provided a more rewarding and engaging experience for non-majors and potential majors alike. Enrollments in the course have increased steadily, forcing the addition of extra sections in each successive year that the course has been offered. In the spring of 2001, this course was adopted at Creighton University, replacing the breadth-based and computer literacy courses previously offered.

While formal testing is required to make definitive claims, anecdotal evidence strongly suggests that students are more capable experimenters and critics of empirical results than they were before taking the class. In lab assignments, students clearly demonstrate the ability to form hypotheses about the behavior of complex systems, design experiments to test hypotheses, and integrate programming as a tool for conducting experiments. Empirical concepts such as the distinction between consistency and accuracy and the Law of Large Numbers are included on tests to ensure that students have a deeper understanding of experimental methods.

For those students who continue in the computer science curriculum, the exposure to empirical concepts prepares them for a deeper understanding of computing concepts. For example, experimentation can help to identify the tradeoffs between data structures, to characterize the efficiency of algorithms, and to understand scheduling schemes within an operating system. The repeated coverage of experimental methods throughout the curriculum reinforces fundamental concepts and further demonstrates the applicability of computing to interdisciplinary applications.

Materials for this course, including programming tutorials and lab assignments, can be found online at **http://www.creighton.edu/~davereed/cs0** .

References

1. Astrachan, O., and D. Reed (1995). "AAA and CS1: The Applied Apprenticeship Approach to CS1." *SIGCSE Bulletin* **27**(1): 1-5.
2. Bagert, D., W. Marcy and B. Calloni (1995). "A Successful Five-year Experiment with a Breadth-first Introductory Course." *SIGCSE Bulletin* **27**(1): 116-120.
3. Braught, G. (2001). "Computer Organization in the Breadth-first Course." To appear in the *Journal of Computing in Small Colleges*.
4. Herrmann, N. and J. Popyack (1994). "An Integrated, Software-based Approach to Teaching Introductory Computer Programming." *SIGCSE Bulletin* **26**(1): 92-96.
5. House, D. and D. Levine (1994). "The Art and Science of Computer Graphics: A Very Depth-first Approach to the Non-majors Course." *SIGCSE Bulletin* **26**(1): 334-338.
6. National Research Council Committee on Information Technology Literacy (1999). Being Fluent with Information Technology, National Academy Press, Washington, D.C.
7. Reed, D., C. Miller and G. Braught (2000). "Empirical Investigation throughout the CS Curriculum." *SIGCSE Bulletin* **32**(1): 202-206.
8. Reed, D. (2001). "Rethinking CS0 with JavaScript." SIGCSE Bulletin **33**(1): 100-104.
9. Tichy, W.F. (1998). "Should computer scientists experiment more?" *Computer* **31**(5): 32-40.
10. Vandenberg, S. and M. Wollowski (2000). "Introducing Computer Science Using a Breadth-First Approach and Functional Programming." *SIGCSE Bulletin* **32**(1): 202-206.
11. Zelkowitz, M.V., and D.R. Wallace (1998). "Experimental models for validating technology." *Computer* **31**(5): 23-31.

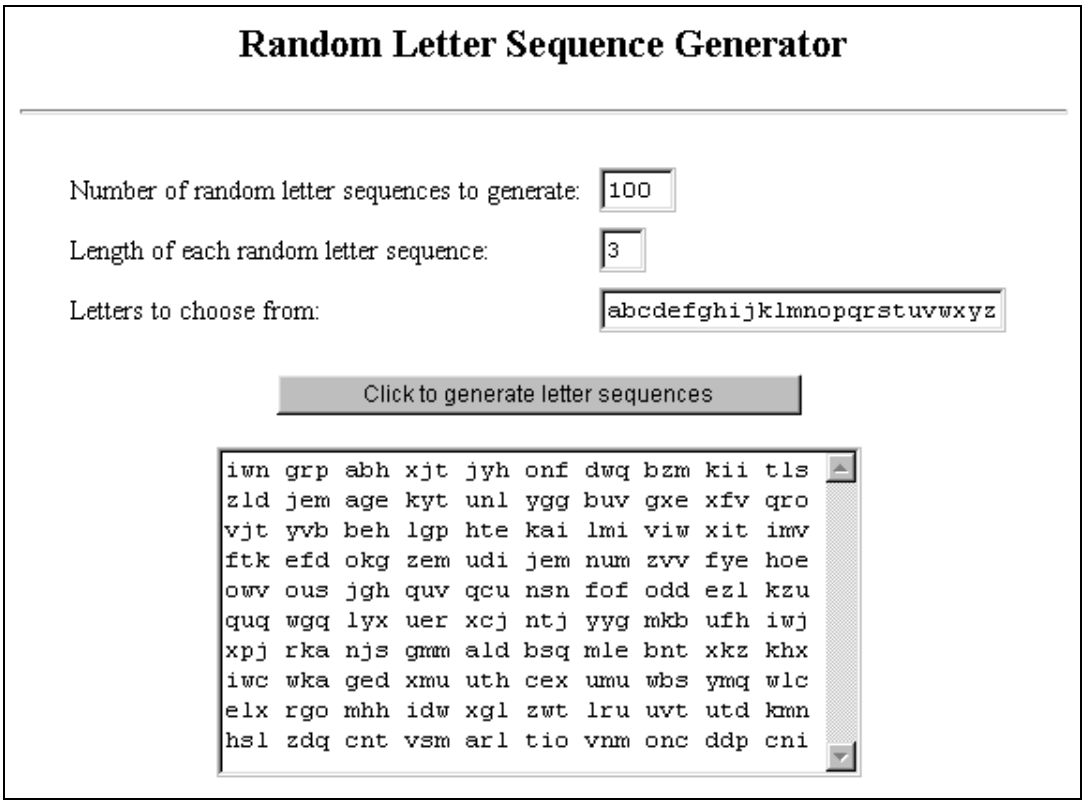

Figure1: Random 3-letter sequences choosing from all letters.

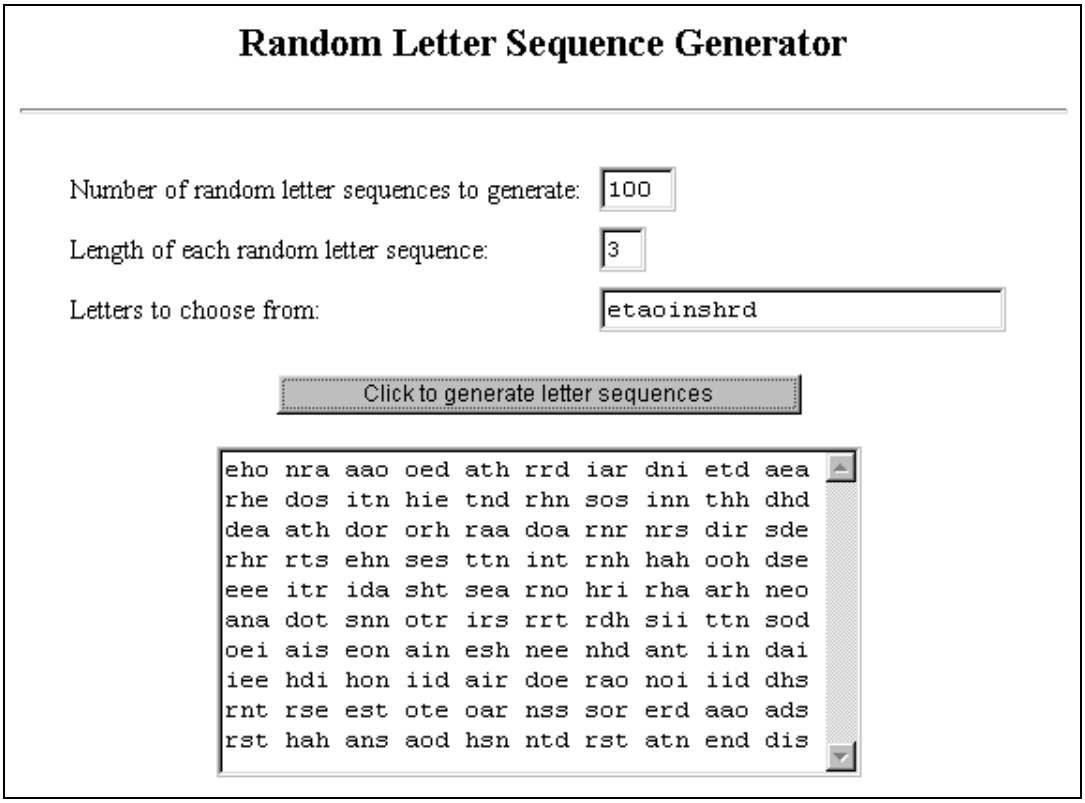

Figure2: Random 3-letter sequences choosing from common letters only.

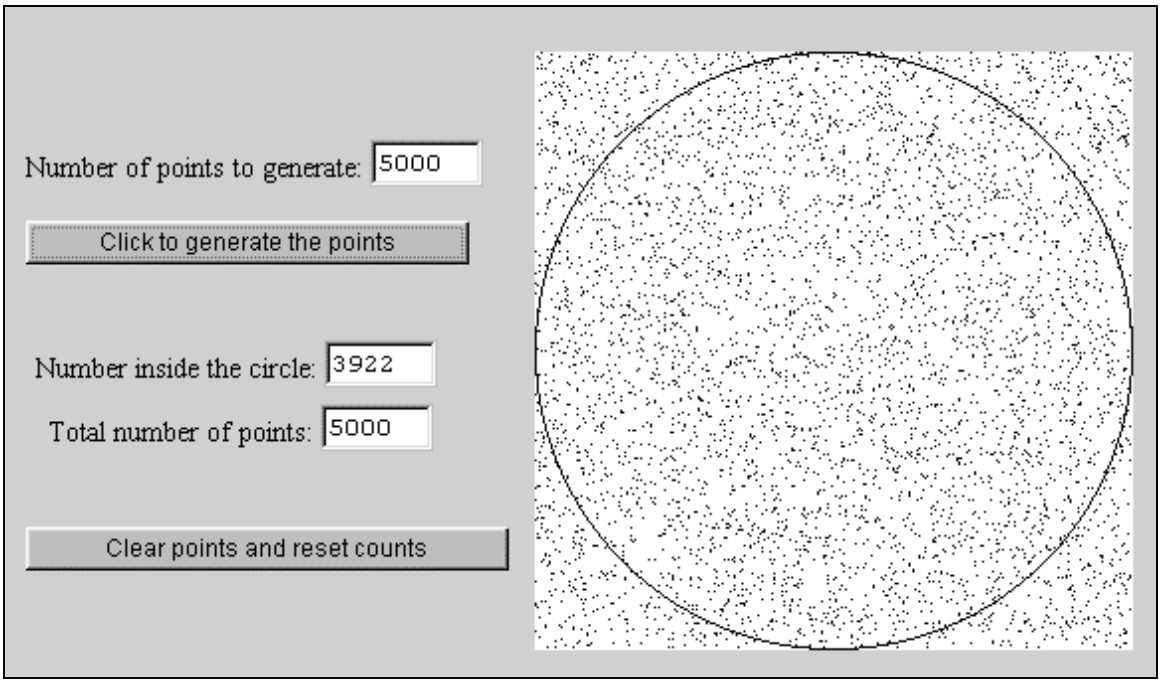

Figure 3. Monte Carlo method for approximating PI.

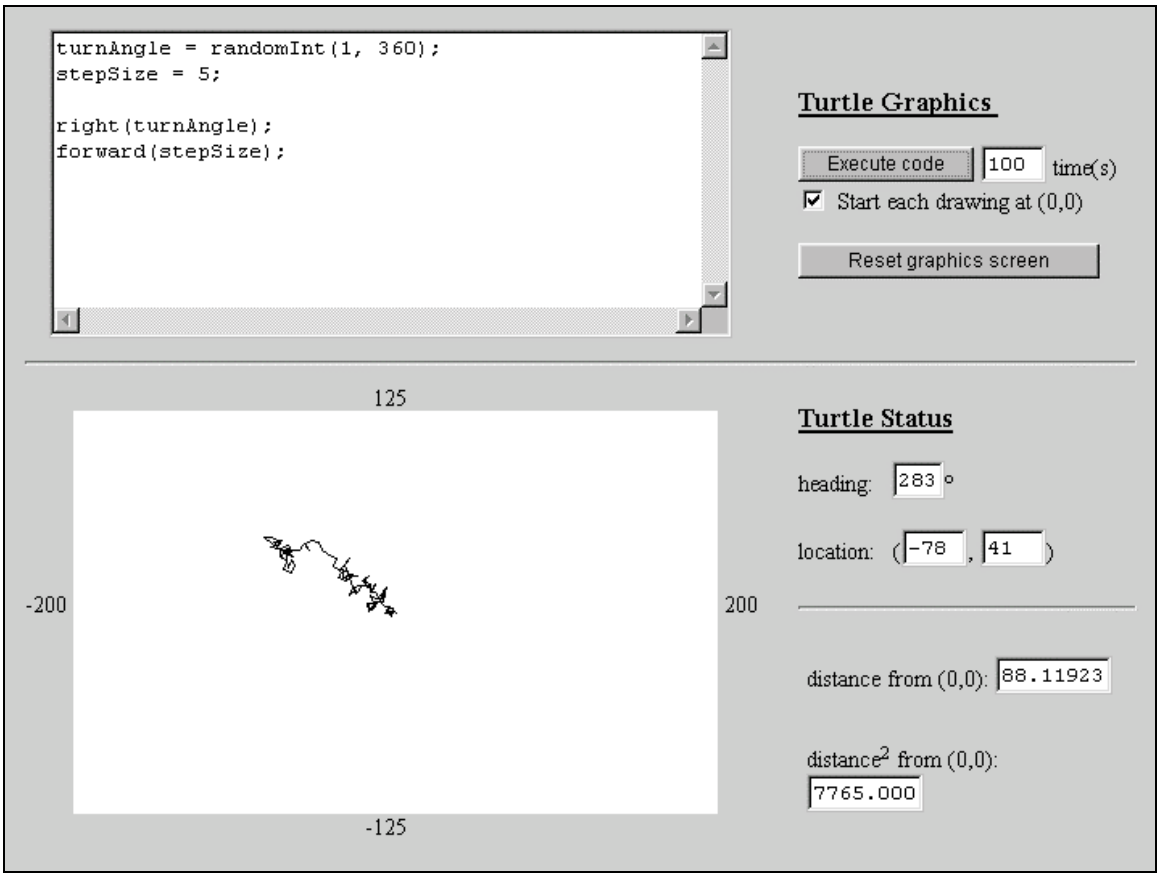

Figure 4: Random walk simulation using Turtle Graphics.

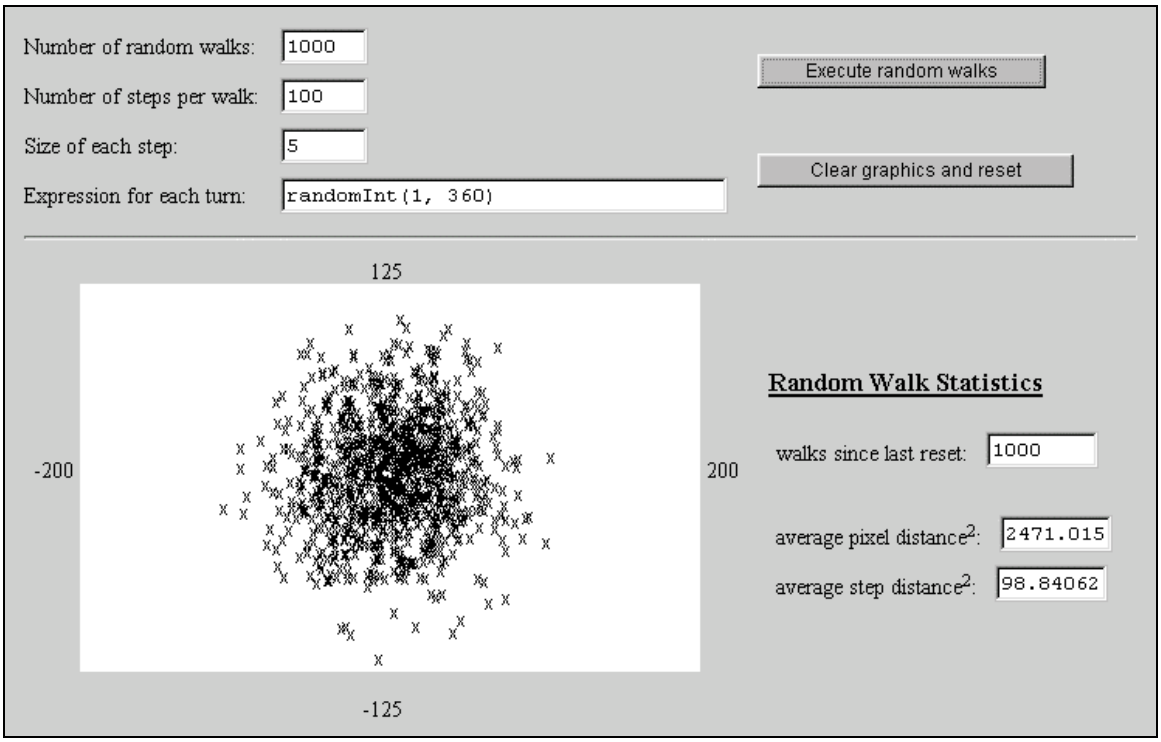

Figure 5: Repeated random walk simulations.

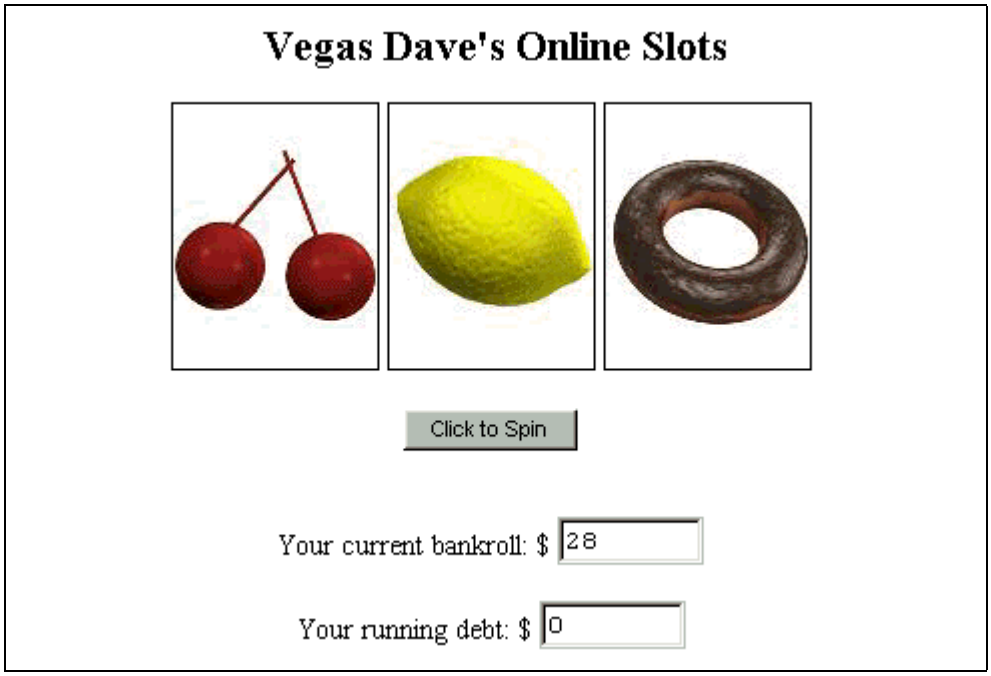

Figure 6. Slot machine simulation.

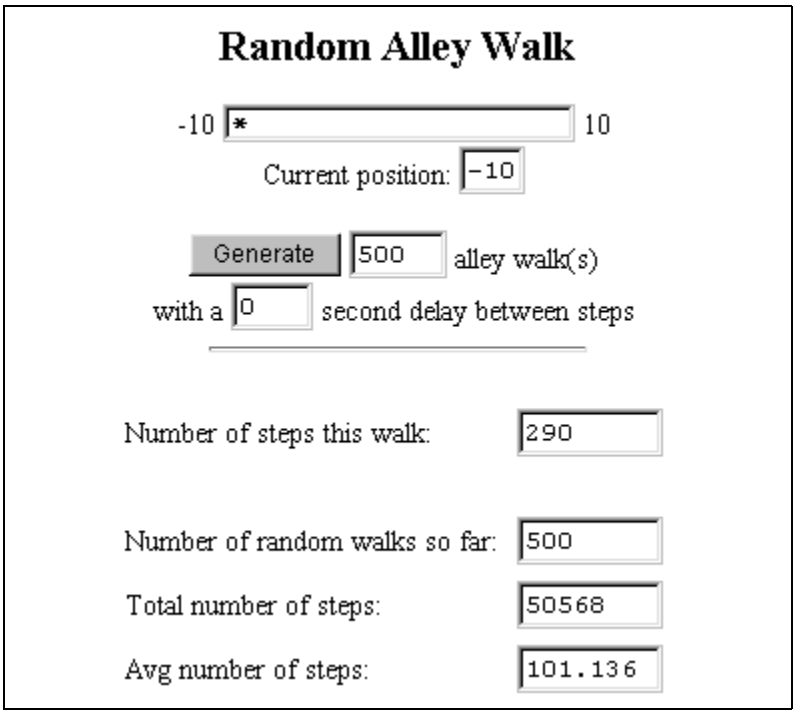

Figure7. Repeated 1-dimensional random walks.

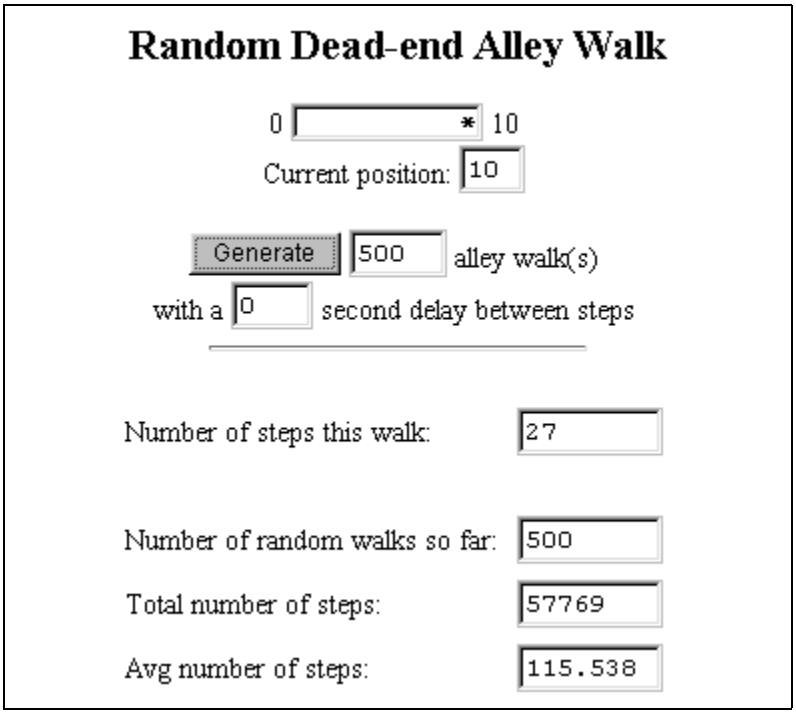

Figure 8. Repeated 1-dimensional constrained walks.