Rethinking Randomness

An Interview with Jeff Buzen, Part I

by Peter J. Denning

Editor's Introduction

For more than 40 years, Jeffrey Buzen has been a leader in performance prediction of computer systems and networks. His first major contribution was an algorithm, known now as Buzen's Algorithm, that calculated the throughput and response time of any practical network of servers in a few seconds. Prior algorithms were useless because they would have taken months or years for the same calculations. Buzen's breakthrough opened a new industry of companies providing performance evaluation services, and laid scientific foundations for designing systems that meet performance objectives. Along the way, he became troubled by the fact that the real systems he was evaluating seriously violated his model's assumptions, and yet the faulty models predicted throughput to within 5 percent of the true value and response time to within 25 percent. He began puzzling over this anomaly and invented a new framework for building computer performance models, which he called operational analysis. Operational analysis produced the same formulas, but with assumptions that hold in most systems. As he continued to understand this puzzle, he formulated a more complete theory of randomness, which he calls observational stochastics, and he wrote a book Rethinking Randomness laying out his new theory. We talked with Jeff Buzen about his work.

Peter J. Denning
Editor in Chief

Rethinking Randomness

An Interview with Jeff Buzen, Part I

by Peter J. Denning

Peter Denning: What is randomness?

Jeff Buzen: Loosely speaking, randomness refers to behavior or events that are variable and unpredictable. For example, if you go to a casino and play roulette, the wheel will generate a sequence of numbers that appear to be random. Although there is no reliable way to predict the value of each successive number, most people assume that each observed number shows up with certain chances that can be summarized by a probability distribution. If you know the probability distribution, you can adjust your betting strategy.

Mathematicians who studied games of chance discovered they could make accurate predictions about average gains and losses over the long term. They assumed that the outcome of each bet a player makes is a sample drawn from an associated probability distribution. It's very easy to imagine what these distributions look like for games that involve dice, cards or spinning wheels. Most casinos have done these calculations and set the odds and payoffs to give a slight advantage to the house.

Uncertainty also arises when analyzing the throughput and response time of computer systems and networks. In these cases, important quantities such as individual message lengths, transaction processing times, and user think times are variable and unpredictable. Mathematicians who modeled these systems followed the same approach they used in games of chance and regarded such quantities as being generated by underlying random processes.

The more I thought about this, the more uncomfortable I got. The image of a user spinning a roulette wheel to determine the length of the next message did not resonate with my own experience—or that of anyone else I knew. For me, randomness is about appearances rather than causes. Instead of thinking about the way a random sequence can be generated, I begin with a simple question: What does randomness look like? More specifically, what types of relationships and regularities should I expect to see when I examine sequences that are presumed to be random?

Any statistician can tell you that an unlimited number of different relationships can be found in a sequence that is truly random in the traditional mathematical sense. In fact, I may only need a few of these relationships to analyze a computer system's performance. If the particular relationships I need make sense to me on an intuitive level and are directly verifiable, I can carry out my analysis with confidence. That's the bottom line. I never need to assume that message lengths or other important values are determined by spinning a roulette wheel or drawing a sample from a probability distribution. I can get by with assumptions that are expressed entirely in terms of relationships among directly observable quantities.

While thinking about these issues, I discovered something even more interesting: Certain results are valid for all types of observable behavior—random, deterministic, or anywhere in between. I refer to such results as laws. I'll show you later a surprisingly simple law that links a system's average response time to the number of active users, the system's overall throughput, and the time users spend thinking between individual requests.

It may strike you as odd that real systems can obey useful laws while still displaying behavior that appears to be driven by random forces. The reason is simple: Uncertainty and variability in individual message lengths and other quantities, which is something we can all see, play no role in the derivations of these laws. Step-by-step details may change with each new set of observations, but the laws remain valid in all cases.

PD: How does this idea of randomness show up in your own work, which is predicting the performance of computing systems and networks?

JB: Suppose I want to know if I can add my traffic to a channel in a computer network without creating an overload. I work with a guideline given by other engineers who have determined that I can safely do this when channel utilization is less than 90 percent. What exactly does utilization mean, and how is it measured?

As a practitioner, the answer is obvious. Simply attach a monitor to the channel that accumulates the total amount of time the channel is actually busy. Suppose this total is 48 minutes during a full hour of monitoring. Then utilization is simply the ratio of channel busy time to the total time: 48/60, which means channel utilization is 80 percent and it's OK for me to use the channel. What could be simpler?

When developing models of channel behavior, traditional mathematicians approach the concept of utilization differently. To understand why, think about the 90% guideline for maximum channel utilization that I've just mentioned. On an intuitive level, we would expect to

find a relationship between channel utilization and the length of time users have to wait for the channel to become available: the higher the utilization, the longer the likely wait.

The general form of this relationship is sketched below in Figure 1. In this particular case, the 90 percent guideline is based on the idea that delays become unacceptably long beyond this point. (The 90 percent guideline is not a general rule; real guidelines vary, depending on many factors).

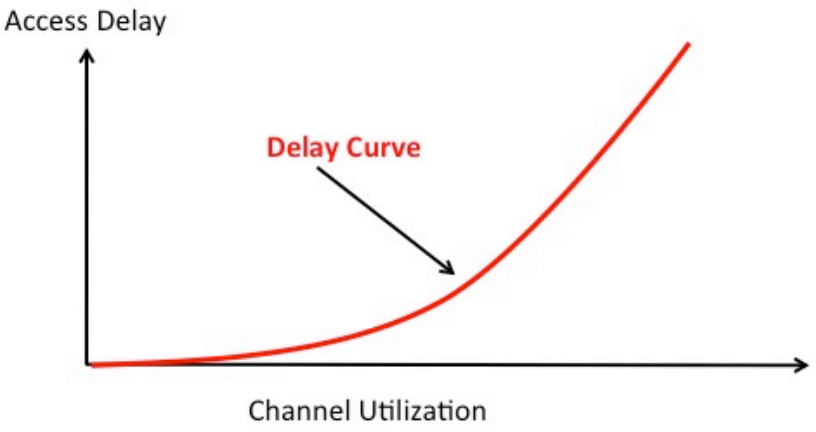

Figure 1 – Time Waiting for Channel to Become Available

Queueing theory is the branch of mathematics that deals with curves of this type. It's based on the idea that there's no way to know for sure if the channel will be busy or idle at any given instant. This leads traditional analysts to assume that some random process switches the channel between in-use and idle at random points in time. In technical terms, traditional analysts represent the state of the channel as a stochastic process.

From this starting point, analysts then derive an expression for the long-term probability that the channel is busy. This is how traditional mathematicians think about utilization: not as a measurable quantity, but as a long term probability. There is of course a link between the measurement and the probability, but that link is based on complex mathematical arguments that don't guarantee the two quantities will always be exactly the same.

I can derive the same curve shown in Figure 1 without making probabilistic assumptions of any type. My analysis is based entirely on variables I can measure directly and assumptions about observable relationships that are easy to verify. This makes everything much simpler—just as in the example I showed earlier. I'll get into some of the details later on.

PD: Gregory Chaitin also wanted to avoid defining randomness in terms of probabilities. He defined randomness of a sequence of numbers (what you call a trajectory) to mean that the shortest algorithm for computing the sequence is as long as the sequence itself. Is this a useful definition of randomness? Would this definition be of any use to computer performance analysts?

JB: In effect, Chaitin is arguing that a long sequence generated by a short algorithm must have a predictable structure and thus cannot be completely random.

Chaitin's work has implications for random number generators (technically, pseudo-random number generators). These utilities are used routinely to generate sequences of “random numbers” for Monte Carlo simulations. Even though the generated sequences are not completely random, the good ones pass standard statistical tests for randomness. The simulations that employ them typically generate valid results. In other words, they are “random enough” to provide satisfactory answers to the questions being asked.

Many of today’s computers use special hardware mechanisms to generate random numbers. Instead of algorithms, these mechanisms are based on inherently random physical processes such as thermal noise or unstable digital circuits. It’s like having a little roulette wheel built right into the motherboard.

PD: Why do we need to rethink randomness? What are we not able to do with our current notions of randomness?

JB: The mathematics behind conventional probabilistic models is perfectly sound. However, the relationship between probabilistic models and the real-world systems I deal with is a bit mysterious. The conclusions I derive depend on the existence of random processes that I cannot see or measure. Moreover, the assumptions are not valid in any system I have ever worked with.

Compounding the mystery is the fact that these models often work astoundingly well in practice—even though the real systems they are applied to violate all the key assumptions of the model itself!

I first noticed this in the early 1970s in conjunction with the queueing network models I developed to predict throughput, response time, and utilization levels of large mainframe computer systems. Analysts who employed my models found them quite accurate for predicting the impact of increases in processor speed, memory size and disk performance. As a theoretician, I was delighted to learn of these successes. However, as a practitioner, who appreciated the complexity of computer systems and the many factors that affect their performance, I found it curious that the models worked well in cases where they seemed to have no right to.

Rethinking Randomness provides an answer to this puzzle. It presents an alternative framework for thinking about and deriving formulas that predict throughput, response time and other performance quantities. In the new framework, all derivations are based on directly verifiable assumptions that are likely to be satisfied by many real world systems. These alternative assumptions lead to results that have the same mathematical forms as traditional results from queueing theory. This explains why traditional formulas are able to work in cases where traditional assumptions are unlikely to be valid.

PD: What are the key points of your approach to randomness?

JB: My approach is based on three simple ideas. The first is that the evolving dynamics of a system can be represented as a sequence of states together with the times of their transitions. I call one of these time-stamped state sequences a trajectory. Trajectories replace stochastic processes as the primary objects I deal with when developing a model.

The second idea is that all performance measures and parameters in a model must represent values I can evaluate by direct observation of a trajectory. Thus, the variables I use in models are all trajectory-based, rather than being properties of probability distributions.

The third idea is that all assumptions used in a model must be stated in terms of relationships among trajectory-based quantities such as the rates at which various transitions take place.

These relationships express properties of randomness that are observable and testable, and can be defined without ever mentioning the concept of an underlying probability distribution.

That's all you need. From this we can build a pragmatic, new theory of randomness that applies to computer systems and networks, and to many other fields where stochastic models are employed.

PD: What is an example of a trajectory?

JB: Suppose I represent an e-commerce server somewhere out in the Internet as a black box with inputs and outputs. Each input is a request from a user that requires the services in the box for some period of time. I'll refer to each such request as a job. Each output is a response that is being send back to that user.

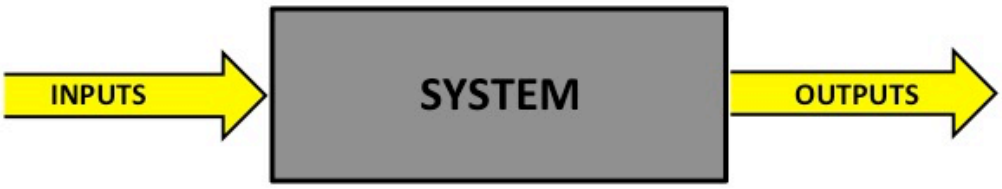

Figure 2 – Black Box View of a System

The state of the system, $n(t)$, is the number of jobs inside at time t. At random times the state increases by one (when a job arrives) or decreases by one (when a job completes). The graph of Figure 3 shows a possible trajectory.

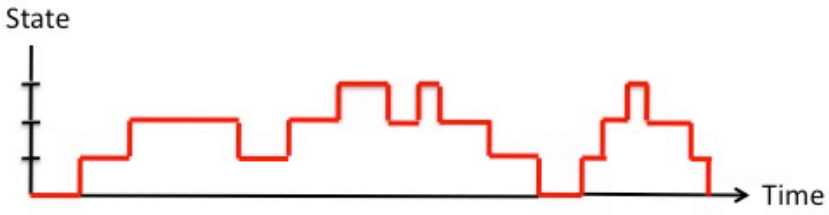

Figure 3 – Typical Trajectory

As a performance analyst I am interested in determining quantities such as the throughput and response time of these systems. From my perspective, these metrics —and all other symbolic variables I use in my analysis—should represent quantities whose values can be obtained directly from a trajectory such as the one in Figure 3. These symbolic variables include utilization (U), throughput (X), average service time (S), average number of jobs in the system (Q), and average response time (R). There are a few other quantities that I also need in special circumstances.

PD: How do you get performance measures from a trajectory, such as the channel utilization you mentioned earlier?

JB: There really are two questions here. First, how do you generate a trajectory by monitoring a real world computer system? Second, given a trajectory, how do you evaluate the symbolic variables that your model requires?

Most modern computers are equipped with built in hardware clocks that keep track of time with a very high degree of precision. Since a trajectory is simply a sequence of time stamped transitions, it's easy to imagine how a system with such a clock could be instrumented to record this information. Simply write some monitoring software that is called each time a job arrives or departs.

All that the monitoring software needs to do is obtain the time of day from the system clock and write that value out to a sequential file—along with the type of event that has just occurred (an arrival or departure). It would of course be advisable to add a header of some sort to identify the record type and the system being monitored.

Besides these event records, we also need periodic synchronization records. Every few minutes, the monitor generates a synchronization record containing the date, time, and number of jobs currently in the system. These records provide convenient starting points (initial conditions) for trajectories. They also provide checkpoints to verify that the trajectory is being generated properly.

It's a trivial matter to generate a complete trajectory from the file to which both types of records are written. Once a trajectory has been generated, it's also a routine matter to extract a basic set of measurements. These include: T = the total length of the observation interval; A = the number of jobs that arrived during the interval; C = the number of jobs that were completed during the interval; B = the amount of time the system is busy processing jobs (the amount of time there is at least one job in the system).

Given these basic measurements, the following variables that will be used in future analyses can be computed:

S = average service time per completed job = B/C

X = throughput (completed jobs per unit time) = C/T

U = Utilization (proportion of time the system is busy) = B/T

It follows immediately from these definitions that U is equal to the product of S and X. Because this simple relationship is valid for all trajectories (random, deterministic, or a combination of the two), I refer to it as a law: in this case, the utilization law. There are other laws whose proofs require a substantially more subtle form of analysis.

PD: Is mean response time the same as mean service time? How would you obtain mean response time from a trajectory?

JB: Response time is defined as the time between a job's arrival at a system and its departure. It includes both service time (time the job is actually being processed) and time the job spends waiting in queues. Average response time *R* is always greater than or equal to average service time *S*. The difference between the two is the extra delay caused by queueing. As an analyst I am almost always asked about response time. There's a way to obtain average response time from a trajectory, but it's a little trickier than other values we've looked at.

Begin by considering a trajectory such as the one in Figure 3. Note that the height of the trajectory increases and decreases as jobs enter and leave the system. The average height is clearly equal to the average number of jobs in the system.

To calculate the average height, first compute *W*, the total area under the trajectory. This value can be computed by simply summing the areas of the rectangles that lie under each horizontal segment in the trajectory. You don't need calculus for these computations.

Next divide *W* by *T*, the length of the interval. The result is *Q*, the average height of the trajectory—and thus the average number of jobs in the system during the entire interval. In other words, $Q = W/T$.

Average response time *R* can also be obtained directly from *W*, but the argument is more subtle. Note first that each job passing through the system adds 1 to the value of n(t) during the time it is actually present. Imagine that each job has an associated timer that counts "time in system" for every second the job is actually present. Since the value of n(t) at any instant represents the number of jobs that are currently present, n(t) also represents the number of timers that are counting "time in system" at that instant. Thus, the total rate at which "job-seconds" are being accumulated by all jobs present at time t will be equal to n(t) job-seconds per second.

This implies that the quantity $n(t)$ has two different interpretations. We've already discussed the original interpretation: The value of $n(t)$ is the number of jobs present at time t. However, the value of $n(t)$ is also equal to the rate at which job-seconds are being accumulated at time t by all jobs currently present. Our analysis hinges on the fact that $n(t)$ represents both these quantities.

Suppose we modify the graph in Figure 3 by changing the label of the Y-axis from "jobs present" to "job-seconds per second." Note that nothing else changes. The X-axis remains the same, the trajectory itself remains the same, and the computed value of *W* remains the same. **However,**

because the units on the Y-axis are different, *W* has a different interpretation: W now represents the total number of job-seconds accumulated by all jobs that were ever present during the interval. Dividing *W* by *C*, the number of jobs that were actually completed, yields the average "time in system" per completed job. This quantity is of course equal to *R*. Thus *R* = *W*/ *C*.

Combining *R*= *W*/*C* with *Q*= *W*/T yields *R* =Q/ *C* / *T*). Note next that *C*/*T*=*X*, the throughput rate of the system. Thus, *R* = *Q*/*X*.

A formal proof of this relationship—as a limit theorem for a general class of stochastic processes—was published in 1961 by John D.C. Little. It's often called Little's formula. I call it "Little's law" because, like the utilization law, it is valid for every possible trajectory that can be generated by observing a real world system over a finite interval of time. I care only about trajectories of this type, and basic algebra is all I need to derive this result.

I first presented these laws and several others in my 1976 ACM Sigmetrics paper, "Fundamental Laws of Computer System Performance." You and I summarized them and added a few others in our 1978 ACM Computing Surveys paper, "The Operational Analysis of Queueing Network Models."

PD: You have specialized in performance issues of computer systems and networks for 40 years. Performance evaluation was important a long time ago when resources were scarce and it paid off to identify and remove bottlenecks from our systems. But today computing resources are not scarce. Is performance evaluation a fading industry?

JB: There are many facets to what might be called the performance analysis industry. Forty years ago, the Internet was in its infancy and most large corporations had their own in-house mainframe systems (think of them as in-house servers that were exceptionally powerful for their era). These systems represented expensive corporate investments that could not be purchased as off the shelf commodities. Corporate customers had to wait for months on end for new or upgraded systems to be manufactured and installed. This placed a high premium on being able to determine in advance when a system would need to be upgraded and what that upgrade needed to be. Dozens of small software companies formed to develop tools for measuring and monitoring system performance, packages for modeling performance and answering "what if" questions, and data bases for storing and reconciling performance data.

Today's environment is quite different. In-house mainframes can still be found in environments that demand very high levels of security, but the majority of today's apps run on servers maintained by third parties and located out in the Internet. Performance remains an important concern in both environments, but performance analysis is now complicated by networking delays that degrade performance, content delivery systems that improve performance by caching data at multiple points around the edge of the Internet, and apps that run partly on servers and partly on clients including desktops, laptops and small hand held devices.

Just the other day I was working on a problem that illustrates the types of issues that currently arise. A customer of a cloud based app was having problems at one their sites. The app stored very large files in the cloud and enabled users working at a dozen or so sites around the world to access and update these files.

Each site had its own local server that could be accessed by employees who worked there. Because of the type of work that was being done, these local site servers were powerful and highly secure systems in their own right.

There were performance problems at one of these sites: sluggish response times, unusually high I/O rates to local disks, and higher than normal CPU utilization. But this problem only affected one site. Why was this happening, and what could be done to fix the problem?

In this case, the problem was traced all the way back to assumptions made during the design of the app. The designers assumed that users would be making small incremental changes to large documents. These changes would then be uploaded to the cloud so they could be synched with users at other sites. But they were also cached locally to provide good performance to local users still working on these documents.

The problem was that users at this particular site were making massive changes. In principle, this is perfectly fine—the system was capable of uploading these changes to the cloud without difficulty. However, since these massive changes were also being cached locally, they wiped out contents that other local users had cached. These other users then needed to retrieve their documents from local disks as they continued working. This caused a dramatic increase in local I/O activity and extra delay for users of these documents. CPU utilization also increased because of the overhead associated with the extra disk I/O. The end result was severely degraded performance and a very unhappy corporate customer.

Once the problem was diagnosed, it was fixed by a minor software update. My point is that the ability to diagnose problems of this type is as important today as it was 40 years ago—even though system architecture and economics have changed dramatically. What hasn't changed is the need for analysts who can understand the significance of performance measurement data

and cut through the clutter to identify the most crucial factors and interactions that determine the performance of complex systems.

This is not really a mathematical skill. It involves instead what I would refer to as proficiency in "systems thinking." I fear that this skill may not be getting enough attention in the computer science curriculum, but I believe that individuals who excel in this area will always have important work to do and will be compensated accordingly.

PD: When you undertook your Ph.D. work at Harvard, you examined a common configuration of a computing system, which you called the central server model. You accepted the traditional stochastic assumptions, which gave rise to a mathematical solution by Gordon and Newell called the product form solution. The GN model was considered useless in practice because its computational complexity grew exponentially with the size of the system (customers + servers). You discovered a fast algorithm for evaluating the product form solution. How did you make this discovery?

JB: As you've just noted, Gordon and Newell derived a general solution that was applicable to a very broad class of queueing network models. My central server model fell into this class, but I did not realize this fact at first—so I set out on my own to solve the large set of linear equations that my model was based upon.

I was able to derive closed form solutions for a few small models, but nothing large enough to be useful in a practical sense. Then I stumbled upon Gordon and Newell's article while searching through journals in the stacks of my Department's library. I saw immediately that their results could be used to solve instances of my model that were arbitrarily large and complex. But then I realized that these solutions involved the summation of an unmanageably large number of individual terms, each of which contained a set of factors raised to various powers.

It was clear that it would not be computationally feasible to evaluate this summation directly, so I looked for a more efficient way. I started by systematically applying all the various techniques I had ever learned in class. None of these standard techniques proved useful. I then began looking over the sea of algebraic expressions I had scrawled on the pages of my notebook, hoping to discover a mathematical relationship of some type that would prove useful.

After weeks of searching, something finally caught my attention. I began—without a clear idea of where I was heading—by separating the sum I was trying to evaluate into two groups. In the first group, the exponent of a particular factor was always greater than zero. In the second group, the exponent of that particular factor was always equal to zero.

I then removed the common factor that appeared (with exponent greater than zero) in every term in the first group—again with no special plan in mind. I was astonished to see that the sum of the remaining terms was exactly the same as the corresponding sum for the same network with one customer removed. This wasn't something I was looking for, but I realized immediately that it was a very promising discovery.

I then examined the second group and quickly realized that it corresponded to the sum for a network with one server removed (the server whose exponent was always zero). The implications of these findings were stunning: the sum I was trying to compute could be obtained with almost no effort by combining the corresponding sums for two smaller networks: the first with one customer removed and the second with one server removed. Multiplying the first sum by a constant and then adding the second generated the result I needed.

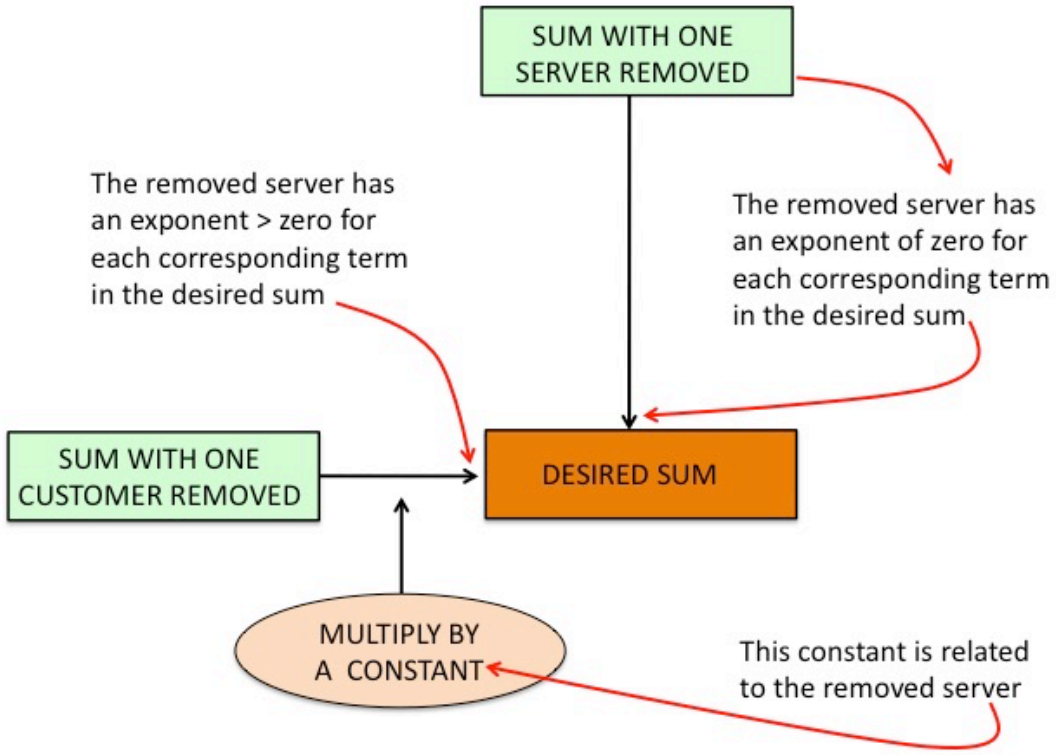

Figure 4 – Key Step in Algorithm

It was obvious that this relationship could be applied recursively to create a powerful and highly efficient algorithm. I then tried a similar procedure with networks that had queue-dependent servers. Even though the mathematics was more complex, I was still able to extend the algorithm to this case rather quickly. Within an hour or so, I also discovered that a number of other important network properties (such as the marginal distribution of queue lengths at each server) could be expressed as simple functions of quantities that are by-products of my new algorithm.

I suspect that many other "out of the blue" advances in the pure and applied sciences arise through a similar process. The ingredients are: (1) a long period of tireless and not necessarily well focused exploration; (2) the recognition that a seemingly minor discovery that happens to turn up has far reaching implications; and (3) the technical ability to exploit these implications to the fullest.

PD: Your algorithm enabled you to rapidly calculate the state probabilities, throughput, and queue length distributions in the time required to fill a rectangular matrix with numbers. Many people considered that a breakthrough because queueing network models could now be applied to real systems. What impact on the computing industry did it have?

JB: Four years after completing my Ph.D. dissertation, I founded a software company, BGS Systems, along with two of my fellow graduate students at Harvard— Robert Goldberg and Harold Schwenk. Conventional wisdom at the time was that hardware companies started in garages and software companies started in basements. Since I had the largest basement, we started our company there.

One of our primary goals was to transform the algorithms and models I developed for my Ph.D. dissertation into viable software products. Modeling technology had evolved substantially in the interim as a result of a worldwide surge of research activity that followed the publication of my models and algorithms. We incorporated many of these new results into our BEST/1 family of modeling products.

By the mid 1980s, most major corporations and large government agencies were using queueing network models to plan the capacity and manage the performance of their large mainframes. For the largest banks and financial institutions, the use of queueing network models for these purposes probably exceeded 80 percent.

BGS Systems was the dominant provider of these modeling products, but other companies using comparable technology also participated in this market. I think it's fair to say that the majority of purchases of new or upgraded mainframe systems that were made during that era were supported by analyses based at least in part on results generated by queueing network models.

PD: Did industrial engineers use your algorithm because the manufacturing facilities they modeled were also queueing networks?

JB: Yes, that's right. I know of industrial engineers who programmed the algorithm on their portable HP calculators and performed detailed performance analyses of manufacturing facilities on-the-spot.

In fact, queuing network models were originally developed to analyze workflows in industrial factories that manufactured physical products of various types. In these factories, complete products are assembled from individual components produced at different stations. Products move from station to station to have the required components attached.

The analysis of flows in such factories provided the impetus for major advances in the theory of queueing networks. Researchers working in this field became aware of my new algorithms shortly after I published my results. It's been reported in the literature that my algorithms had an important influence on that field, but I have no first hand knowledge of the details. If you're interested, I've included a reference to a paper by Rajan Suri and coauthors with more details.

Suggested Readings

J.P. Buzen, *Rethinking Randomness: A New Foundation for Stochastic Modeling*, CreateSpace, 2015.

J.P. Buzen, Fundamental Laws of Computer System Performance, *SIGMETRICS '76: Proc. 1976 ACM SIGMETRICS Conf. on Computer Performance Modeling, Measurement and Evaluation,* April 1976, 200-210.

P.J. Denning and J.P. Buzen, Operational Analysis of Queueing Network Models, *ACM Computing Surveys* 10, 3 (Sept. 1978), 225-261.

R. Suri, G.W. Dielh, S. de Treville and M.J. Tomsicek, From CAN-Q to MPX: Evolution of Queueing Software for Manufacturing, *INFORMS Interfaces* 25, 5 (Oct. 1995), 128-150.

About the Author

Peter J. Denning (pjd@nps.edu) is Distinguished Professor of Computer Science and Director of the Cebrowski Institute for information innovation at the Naval Postgraduate School in Monterey, California, is Editor of ACM *Ubiquity*, and is a past president of ACM. The author's views expressed here are not necessarily those of his employer or the U.S. federal government.

DOI: 10.1145/2986329

