Deterministic v Stochastic Behaviour of Software

Peter Bernard Ladkin, Harold Thimbleby, & Martyn Thomas

Version 2 of 20240624

Every so often computer scientists encounter a view that

> *Software execution is deterministic; therefore it is not stochastic; therefore stochastic processes (such as typically Bernoulli processes or Poisson processes, more generally renewal processes) cannot be used to model software execution. This entails that statistical reasoning cannot be used to model software failures in execution.*

Such a view requires some critical thinking. First, one has to (try to) say what it means for "software execution" to be deterministic. Given that, one must distinguish the property of being deterministic from the property of being predictable. If software execution is not predictable then some of that execution (some execution paths) is uncertain. The science of uncertainty, such as statistical reasoning using stochastic processes, is applicable to processes with uncertain execution paths and therefore uncertain outcomes. Indeed many senior computer scientists have spent their working lives on statistical reasoning about software execution, and have won awards for doing so. So it remains a puzzle why some people think this cannot be done.

What it may mean for software execution to be deterministic

Languages in which software is written, from machine language through assembler on to very-high-level languages, are often given a formal semantics. This semantics can be of the Floyd-Hoare type, in which the execution of an atomic program statement in a given partial state of memory, which we can take as consisting of the current values of all program variables, results in a new, definable, state of memory. For example, consider executing the statement x ⟵ (x+1).

> "x ⟵ (x+1)" means "variable x receives a new value which is +1 greater than its old value". It is written "x = x+1" in Fortran, "x := x+1" in various other languages, and "x' = x+1" using the "prime" notation for old and new values, as in the specification languages Z and TLA.

The semantics of this action written in Floyd-Hoare-style would say: if the value of variable x is A, and "x ⟵ (x+1)" is executed, then the new value of x is A+1, and – a crucial *addendum* – the values of all other program variables remain unchanged. Given this, if we know what the memory state was before the execution of "x ⟵ (x+1)", then we know what it is afterwards. This is "deterministic execution": the after state is determined exactly, given the before state. Notice that the addendum is crucial: if other variables may change their values as x ⟵ (x+1) is executed, and it is not said how, then we do not know what the state of memory is afterwards and we cannot call the state determined. Floyd-Hoare notation makes a distinction between program variables, say written lower case, and values that these variables can take on, say written upper case. Then Floyd-Hoare notation for this action and its consequence, a so-called Hoare triple, would be $\{x = A\}x \leftarrow (x+1)\{x = A+1\}$. Here, the expressions inside the curly parentheses are mathematical expressions of partial memory values. A general Hoare triple has the form {memory state before}action{memory state after}. Note that this notation contains no annotation for whether the addendum holds or not.

We can regard input to the program as given by a special memory variable called "inputvalue" and the Hoare triple describing input as $\{\}\text{inputvalue} \leftarrow A\{\text{inputvalue} = A\}$. So we don't have to consider inputs as different from any other program action, except in so far that the input value is not determined at program start.

If all the actions in a program can be specified by means of Hoare triples with addendum, then given a memory state at the start of program execution, and a given sequence of inputs, the memory state at the termination of execution is uniquely determined. This is plausibly what is meant by saying that software execution is deterministic. There is a unique sequence of states – the execution path – from the initial state to the end state – given the input values along the way.

All this assumes that the program terminates. It might not – it might "run for ever" (that is, until explicitly terminated by an ab-end or by an input, or by switching the computer off). We can call a continuously-running program deterministic if the memory state at any stage of execution is uniquely determined by the memory state at start and the inputs to the program by that stage.

If a program has such a semantics as explained in this section, we call it "*nominally deterministic*".

We have described what is called the "operational semantics" of a program written in an "operational language". An operational programming language specifies a series of operations on memory values and an operational semantics says what those operations achieve, one by one, and enables you to determine what a program will achieve by decomposing it into successive operations.

There are other ways of construing computation, such as with the programming language Lisp, which (as "pure Lisp") considers a program to be a massive expression of a mathematical calculus called the λ calculus, which has "reduction rules" by means of which one can substitute subexpressions for nominally "simpler" subexpressions and continue until you can't do it any more – a process called "evaluation". Another paradigm is given by the language Prolog, whose programs are logical expressions of a certain form, and whose executions consist in performing logical inference on those expressions. We don't consider either of those paradigms here. We are concerned here with programs whose meaning is given by a semantics of actions, not by an evaluation algorithm.

But is program execution deterministic, for "real programs"?

Common programming languages such as C are often defined by a document which everyone agrees to be the definition, a "standard". There is a "C standard". It defines a syntax for program statements and additionally says what actions are defined by which program statements, sometimes in Hoare-triple style.

But in the C standard, notoriously, not all legitimate program statements are given a semantics. Some are said to be "compiler dependent" – that is, given that you know what compiler is used to translate your C program into machine code, you can determine what action such a statement performs. And if you use a different compiler, it may well perform a different action. Users of languages such as C in a situation in which program reliability is very important will often use a subset of the full language – a subset in which the action of each statement is determined by the standard. Alternatively, another document can extend the standard to statements with compiler dependence for such high-reliability applications. By using a combination of these techniques, it is possible to achieve nominal determinism.

There are a number of "software development environments" (SDE) or "toolsets" on the market which enable programs for high-reliability applications to exhibit such nominal determinism. These SDEs/toolsets often come with formal analytical capabilities which enable a programmer to prove certain properties of her program, given that this nominal determinism is assured. These can be very useful, indeed essential, in providing incontrovertible reasoning that a program execution satisfies a given functional specification. This is increasingly the basis of program reliability reasoning in applications in which exceptionally high reliability is required.

Such a program, written with such an SDE, accompanied by proofs using the toolset that the program satisfies its functional requirements, may guarantee nominal determinism but does not necessarily guarantee that the program will terminate with an output satisfying the functional requirements, as follows. Software which is nominally deterministic, when run on a processor in the real world, does not necessarily produce the same result for the same inputs each time. The processor might have faults. Or, if the software is distributed over multiple processors, some other component might affect the result. Or there could be temperature- or humidity-related intermittent HW failures. There are, on any real processor, limits on correct execution according to the Hoare semantics. Most notably, there are limitations on number representations and number arithmetic which have to be met in order to ensure nominal determinism.

Besides the use of an SDE or toolset which guarantee nominal determinism, and abiding by hardware limitations as above, there are further conditions which are necessary to ensure nominal determinism when software is executing in a typical environment – see below.

Software that is nominally deterministic can be unpredictable

A pseudo-random number generator (PRNG) is a piece of software which, upon a prompt, produces a number whose value is not predictable from the value of the prompt by any available algorithm except itself. The purpose of pseudo-random number generators is that they appear to generate *random numbers* (a precise mathematical concept), but their behaviour is reproducible: they will produce the same number as output for the same prompt each time. The purpose of pseudo-random number generators is to produce software whose outputs are not predictable but which can be tested, which amongst other things entails the behaviour must be reproducible.

When we say the behaviour is "not predictable" we must be more specific. If the PRNG is nominally deterministic, then in principle one can write out and evaluate the Hoare triples and input the prompt value and determine what value is output. But one wants this process to take a lot longer than the PRNG takes to execute, and indeed it may take more resources (logical and computational) to compute the semantics than are available. The PRNG is thus *resource-constrainedly* unpredictable. Indeed, PRNGs exist whose nominal predictability is unattainable by any reasonably-available set of computational resources. This is indeed their point.

A PRNG is a piece of software which is nominally deterministic. However, its output is practically unpredictable.

Considering some mathematics gives further examples of unpredictability of values of mathematical functions. The phenomenon of *chaotic functions* became known through the thesis of Henri Poincaré in 1879 (including the famous "three-body problem"). Chaotic function behaviour can be exhibited by infinite series. One relatively well-studied function is $x \leftarrow \lambda.(1-x^2) - 1$. That is, given a value x_0, the value x_1 is given by $\lambda.(1-x_0^2) - 1$; $x_2 = \lambda.(1-x_1^2) - 1$; and so on. This is called a difference equation. One can wonder how the values of x_0, x_1, x_2, ….. can be characterised. This turns out to depend on the value of λ. One possibility is that the values become periodic; that is, that $x_{j+k} = x_j$ for some fixed k and for any $j >$ some fixed r. If k is the minimum number such that $x_{j+k} = x_j$ for a given series, we say that the sequence is *periodic with period k.* For low values of λ this is so, but for values of λ between about 3 and 3.5, the series converges on one of two values, depending on x_0 (to say a series *converges* to a value v is to say that all terms in the series beyond a certain point are close approximations to v). With values of λ beyond 4, there are many, many values which a series might approach, and with even higher values of λ the series takes larger and larger numbers of possible values. This is illustrated in Figure 1, in which the possible values of the terms of the series are given for various values of λ.

The significance of this piece of mathematics is as follows. Suppose a program contains a series of statements

```
x ← 0.534;
for j = 0 to <very large integer> do  x ← λ.(1-x²) – 1;
if (x < 0.5) y ← 1 else y ← 0
```

then it becomes practically impossible to predict whether the resulting value of y is 0 or is 1.

Notice that this is all deterministic mathematics. Deteminism does not exclude unpredictability.

Not only that, but it can be shown that the function is chaotic, which entails that an arbitrarily small deviation in value for x_0, or indeed any x_n for $n > 0$, results in a value for some (indeed many) x_k for $k >> n$ that is arbitrarily far away from the value it would have had had the deviation not occurred. Computer arithmetic is necessarily mathematically approximate; irrational numbers, for example, can only be approximated. That entails that any attempt to run a program such as the above cannot be shown to output the mathematically correct answer: the mathematically true value of x can be arbitrarily far away (within the limiting interval) from its computed value, no matter how accurate the computer arithmetic. (There may be of course particular sequences whose computer-arithmetic turns out to be exact. But these are very likely not dense in the space of all sequences which start from a precisely representable x_0.) This uncertainty is fundamental to the construction; it cannot be avoided. The only way intellectually to handle it is via the mathematics of uncertainty.

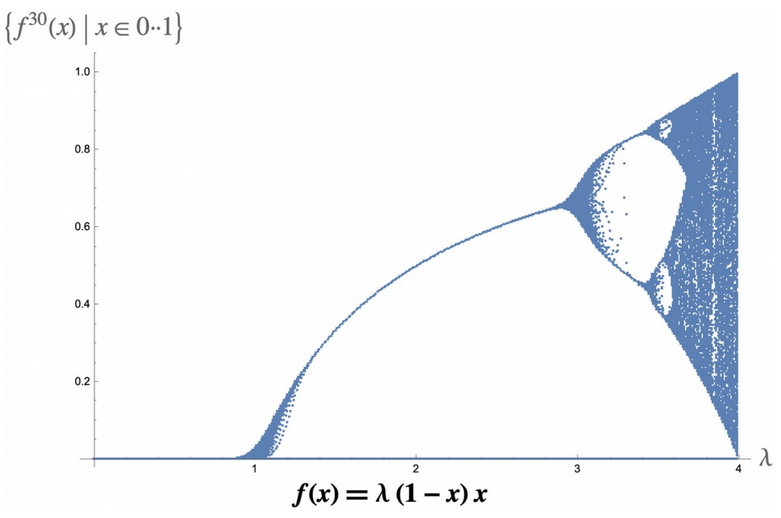

Figure 1: How possible values of x_{30} in an infinite series $x \leftarrow \lambda.(1-x^2) - 1$, starting from an x_0 in the interval (0,1), are dispersed for various values of λ *(please note: there is some quantisation inaccuracy due to jpeg compression)*

This type of chaotic functionality is discussed in more detail in Chapter 6, Universality, of Peter Smith, Explaining Chaos, Cambridge University Press, 1998. Rather than the function above, however, Smith considers the *logistical function* of the mathematical biologist Robert May, which is $x \leftarrow \lambda.x.(1-x)$.

Dealing with unpredictability in the presence of determinism

There are ways to deal with deterministic outcomes which are uncertain or unpredictable. A different kind of unpredictability arises when one simply does not know a particular fact. Consider the following game:

Guess the number of orange Smarties in a sweety jar full of Smarties.

There is a fact: for some number n, the number of orange Smarties in the jar is n. But participants have no effective way of determining what n is. For those prepared to play this game, notice that they are guessing the value of something that is *already determined* but unknown to them. Of this game, one can ask the following two meaningful questions.

Question 1: what is the probability of a guess being correct?

Question 2: what is the probability that a guess is within 5 of the true value?

The notion of probability here enters through the participants. A participant develops a personal probability for each possible outcome. If the game is a betting game, then participants are prepared to accept "odds". The odds they are prepared to accept is a measure of what may be called their personal probability for the outcome. This observation was turned into a rigorous semantics for so-called subjective probability by Frank Ramsey about a century ago (Truth and Probability, 1926. See Stanford Encyclopedia of Philosophy, entry Frank Ramsey (many authors), https://plato.stanford.edu/entries/ramsey/). Subjective probability satisfies the Kolmogorov axioms for a probability measure, so it is appropriately termed "probability".

Notice that the state of the world on which the game and its parameters are based is determined in advance. The probability, given in the Ramsey semantics by the value of the odds in bets a participant is prepared to place/not to place, arises from the uncertainty of the participants as to what the actual state of the world is.

We can conclude that the unpredictability of the outcomes of running mathematical programs is subject to similar techniques to handle uncertainty as the Ramsey semantics for uncertainty. This introduces the vocabulary of statistical assessment (albeit using the Bayesian model) into deterministic (determined) but unpredictable situations.

Other ways in which stochastic processes arise in deterministic transformations

Suppose you have a program, and the program terminates on all inputs from the set I. The program succeeds on certain inputs (gives you a correct output, according to the purpose/specification of the program) and it fails on certain other inputs from I. This can be given by a mathematical function with domain I and range {0,1} of which the value is 1 if the program succeeds on $i \in I$ and 0 if the program fails on $i \in I$. Suppose inputs are taken from a probability distribution on I; that is, that values $i \in I$ given to the program occur as inputs with a frequency given by that distribution. Then one can reasonably ask for the proportion of 1's, respectively 0's (= 1 – proportion-of-1's) in the output. These are probabilistic and statistical calculations, performed on the behaviour of nominally deterministic software programs. A detailed consideration of how Bernoulli processes capture the relevant questions (and answers) in such a study is given in Peter Bernard Ladkin, Software, the Urn Model, and Failure, 2015/2017 https://rvs-bi.de/publications/books/RVS-Bk-17-01/Ch01-SoftwareUrnModelFailure.pdf

Thus can stochastic processes describe the behaviour of deterministic programs.

A Canonical Model of Probability: Dice Throwing as Deterministic and as Stochastic

The origins of probability are said to have started with the study of games of chance, such as throwing a die or flipping a coin. Let us consider throwing a die. Suppose you have a die, and it is unbiased. Literally physically unbiased (it is likely that you can't get such a thing in the real world, but one can approximate it well enough for the following reasoning). The Laplacian interpretation says that the die has properties which we can call propensities[1]. They are real properties of the die, such as its colour and weight are. Here are six propensities:

1. The propensity to land with 1 uppermost when tossed.
2. The propensity to land with 2 uppermost when tossed.
....
6. The propensity to land with 6 uppermost when tossed.

These propensities have values. The value of each of them is taken classically to be the same, namely 1/6.[2]

Are there such objective properties of the die as these propensities? Well, "when tossed" is not particularly well defined. Consider the following.

Suppose a robot hand tosses the die in a closed room with no drafts. At the point of release, record the orientation of the die, the velocity of its C of G, and the angular velocities about its C of G, and the height of its C of G above the landing plane. As well as the local swirls of the atmosphere surrounding the die. Messrs Newton, Navier and Stokes have shown us how to calculate the uppermost face when it lands. It is a finite trace in time of a deterministic function.

Say there are people standing at a window into the room. At the point at which the die is tossed, you shield the window (render it non-transparent so they cannot see what is happening in the room) and ask them to lay bets on the outcome. The sensible people will video the release and send the data off from their smartphone to some computer, which will give them the correct answer. It is not a game of chance in this case.

It does become a game of chance if you specify a range for all those physical parameters of the toss, and you don't allow a participant to record the toss she sees in any way. Given those input ranges, there will correspond outputs which consist of the values 1 to 6 along with the measures of the sets in which a given value is achieved. If the ranges are big enough, you'll get the pairs <1, 1/6>, <2, 1/6>, ... <6, 1/6>. If the ranges are small, you might get varying values: if it is let go - that is, horizontal component of velocity is 0 - with 1 uppermost and angular velocity of 1 rotation a second in the vertical plane of the toss and air currents are negligible and the landing plane is (one second's worth of gravitational motion) under the toss, then the chance that 1 will be uppermost is 1 or close to it and the chance that 6 is uppermost is 0 or close to it.

We may conclude that the result of a die toss is a deterministic function (to all intents and purposes)

1 This is not to suggest that Laplace used this term. The term has been used most recently by Karl Popper and followers. Christian Huygens, author of the first published book on probability, Ratociniis in aleae ludo (1657) called it “proclivity”, a term also used by Jacques Bernoulli, the author of Ars conjectandi (1713). This was at a time when the concepts surrounding probability were being developed, so it is by no means certain that these authors meant the same thing. See Ian Hacking, The Emergence of Probability, Cambridge University Press 1975.

2 They thus form what Jerzy Neyman called a Fundamental Probability Set of equiprobable, mutually exclusive alternatives.

described by Messrs Newton, Navier and Stokes. To describe that function as a physical property of the die seems a metaphysical stretch too far, like ascribing the result of a road accident as a propensity of the cars involved.

However, it does make sense to prescribe ranges of inputs and ask what are the ranges of outputs for inputs in those ranges, and what are their measures. As above, and was also done for the chaotic function. The answer for the unbiased die is fairly simple, unlike the answer for the chaotic function.

Although the entire process of die-throwing as described is deterministic, there is a stochastic process to be identified here: tossing a die with unspecified parameter values in specified ranges. The possible outputs are bounded and have measures. If you are not to lose money while betting on outcomes of this process, you lay your bets according to those measures, and you lay lots of them, because indeed in the shorter term you might lose (or win) quite a lot. Messrs. Bernoulli and Laplace quantified all of that for us.

Software in the "Real World"

We have seen that asking the question whether a process is deterministic or stochastic, and then to use the answer to decide whether you can use statistical inference or not, is misleading. We can use the language of statistics, if we wish, to talk about the values or ranges of values of a deterministic function, as above. And sometimes, as in the case of fundamental unpredictability, we will need to do so.

We have so far considered intellectually simple functions. But much software has more complex functionality than that. Besides the use of an SDE or toolset which guarantee nominal determinism, and abiding by hardware limitations as above, we noted above that there are further conditions which are necessary to ensure nominal determinism. Some of these could be internal to the software, and some concern the environment within which the software runs. A partial list is:

- that the hardware is deterministic, with no intermittent faults;
- that the system as a whole is stateless (which rules out any persistent memory such as databases);
- that the software contains no race conditions (that's what killed patients in the disaster with the Therac-25 radiation machines);
- that all the connected systems are deterministic and stateless (notice this rules out clocks!);
- that every user session is synchronous with respect to any external interrupts (which rules out connections to active communications).

In the case of, for example, IT systems, we may consider that a software-based system might always produce the same results with the same inputs if all settings and environmental conditions are the same, but in practice we can rarely be completely certain whether or not any property of the system or its environment has changed. In all but the shortest term the practical stance is to consider that some elements have changed and that we probably won't know which and how.

Then there is the fact that IT systems have human users. Even if all components are constant, different users might use the system in different ways with different results. The Post Office Horizon system is an example. The Post Office apparently considered for many years that a system user (a Postmaster or her employee) who got the wrong result was obviously at fault. Such an attitude contributed to what has been described as the largest miscarriage of justice in British legal history.

Even if complex software is deterministic in principle, there is often so much uncertainty concerning the environment and the precise mode of use of the software system, and even the versions of various software subsystems in use, that considering the software under conditions of uncertainty is the manifest way to approach reasoning effectively about it. That entails using the mathematics of uncertainty.