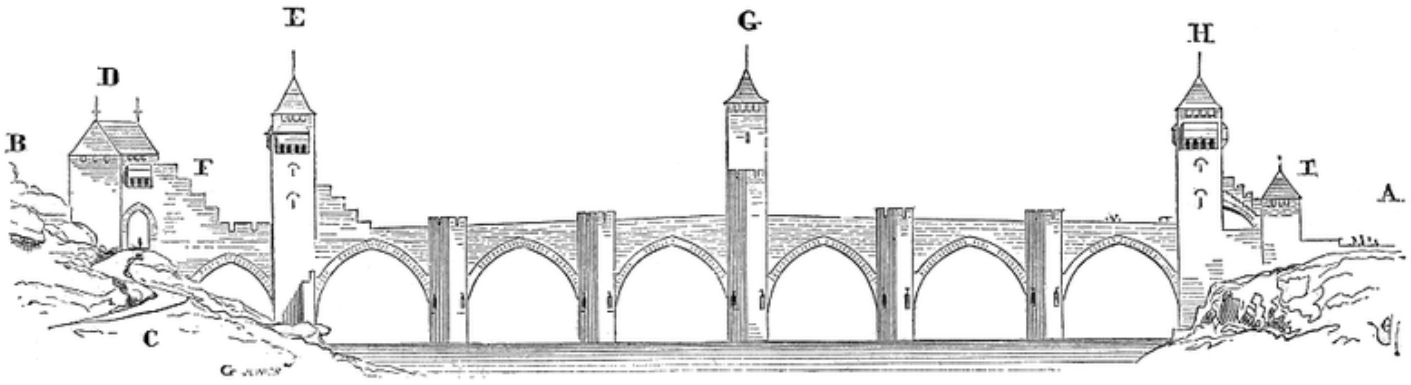
Janos

A Pedagogical Stochastic Simulator

Antoine B. Rauzy

PUBLISHED BY THE ALTARICA ASSOCIATION
ALTARICA-ASSOCIATION.ORG

First printing, February 2022



Contents

Lily

On la trouvait plutôt jolie, Lily
Elle arrivait des Somalies, Lily
Dans un bateau plein d'émigrés
Qui venaient tous de leur plein gré
Vider les poubelles à Paris

Elle croyait qu'on était égaux, Lily
Au pays d'Voltaire et d'Hugo, Lily
Mais, pour Debussy, en revanche
Il faut deux noires pour une blanche
Ça fait un sacré distinguo

Elle aimait tant la liberté, Lily
Elle rêvait de fraternité, Lily
Un hôtelier, rue Secrétan
Lui a précisé, en arrivant
Qu'on ne recevait que des Blancs

Elle a déchargé des cageots, Lily
Elle s'est tapée les sales boulots, Lily
Elle crie pour vendre des choux-fleurs
Dans la rue, ses frères de couleur
L'accompagnent au marteau-piqueur

Et quand on l'appelait Blanche-Neige, Lily
Elle se laissait plus prendre au piège, Lily
Elle trouvait ça très amusant
Même s'il fallait serrer les dents
Ils auraient été trop contents

Elle aima un beau blond frisé, Lily
Qui était tout prêt à l'épouser, Lily
Mais, la belle-famille lui dit
"Nous n'sommes pas racistes pour deux sous
Mais on veut pas de ça chez nous"

Elle a essayé l'Amérique, Lily
Ce grand pays démocratique, Lily
Elle aurait pas cru sans le voir
Que la couleur du désespoir
Là-bas, aussi ce fût le noir

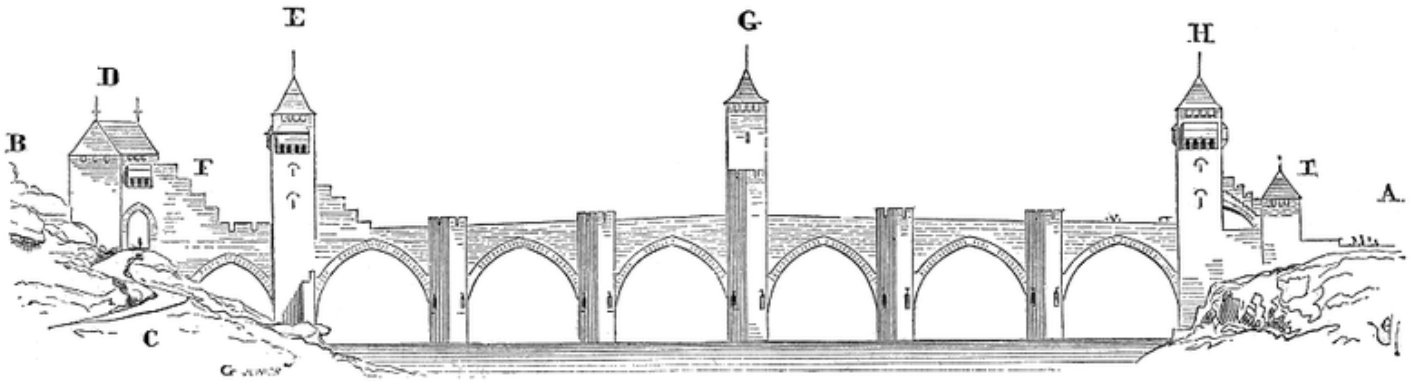
Mais, dans un meeting à Memphis, Lily
Elle a vu Angela Davis, Lily
Qui lui dit "viens, ma petite sœur"
"En s'unissant, on a moins peur"
"Des loups qui guettent le trappeur"

Et c'est pour conjurer sa peur, Lily
Qu'elle lève aussi un poing rageur, Lily
Au milieu de tous ces gugus
Qui foutent le feu aux autobus
Interdits aux gens de couleur

Mais, dans ton combat quotidien, Lily
Tu connaîtras un type bien, Lily
Et l'enfant qui naîtra, un jour
Aura la couleur de l'amour
Contre laquelle on ne peut rien

On la trouvait plutôt jolie, Lily
Elle arrivait des Somalies, Lily
Dans un bateau plein d'émigrés
Qui venaient tous de leur plein gré
Vider les poubelles à Paris

Pierre Perret, 1977



Preface

Rational

When designing my course on model-based systems engineering, I faced a challenge: I was absolutely convinced that a rigorous introduction to the topics was necessary, which should rely on a formal (textual) language, with a clearly defined syntax and a well established mathematical semantics. The problem was that such language does not exist, or more exactly that existing ones, such as Modelica (Fritzson 2015) or AltaRica (Batteux, Prosvirnova, and Rauzy 2019) are both too complex to serve as the support for an introductory course and not fully adjusted to the needs.

Consequently, I decided thus to design a new one, based on the S2ML+X paradigm (Batteux, Prosvirnova, and Rauzy 2018; Rauzy and Haskins 2019), i.e. relying on the one hand on a simple mathematical framework (the X), and on the other hand on the set of object- and prototype-oriented constructs gathered into S2ML. The problem was thus to find the appropriate mathematical framework.

With that respect, systems of data-flow equations are probably the simplest one can imagine. It consists of sets of equations of the form:

$$v = e$$

where v is a variable and e is an expression involving constants, other variables and operations such as the addition, the subtraction... This is about all, with the only additional constraint that the system must be data-flow, i.e. that a variable cannot depend eventually on itself.

Such systems of data-flow equations are of interest in the systems engineering framework because variables can be used to represent the state of components as well as the flow of matter, information or energy circulating throughout the network of components.

It started working on this idea and found out soon that it would be very useful and much more fun to add some randomness both in the expressions defining states and flows. No sooner said than done! I had with other modeling languages I was working on all what I needed.

Systems of stochastic, data-flow equations were also perfect to introduce Monte-Carlo simulation, an essential tool in engineering. Since its introduction in the late 1940s by Stanislaw Ulam and John von Neumann, the Monte-Carlo method is actually pervasive in sciences and engineering.

The main idea behind this method is that the result of a certain calculation is computed based on repeated random sampling and statistical analysis. This method applies not only to calculations on stochastic models, but also to calculations on deterministic ones for which there is no analytical solutions or analytical solutions are too difficult to obtain.

`Janos` is thus a pedagogical stochastic simulator. Models are systems of data-flow equations written in the S2ML+DFE domain specific modeling language. It comes as a command interpreter, making it possible to perform various studies.

I made the choice to develop `Janos` in Python, for two main reasons. First, developing in Python is incredibly faster than in more traditional languages such as C++ or Java. Second, Python is an interpreted language and there are Python environments available on all main operating systems, Windows, MacOs and Linux. This choice has some drawbacks however. First, it forces students to install a Python environment on their machines. Well, this is only half a drawback as I anyway strongly encourage students to learn Python, as a fantastic productivity tool for their studies, and later for the engineering and scientific work (Rauzy 2020). The second, more serious drawback is the poor performance of interpreted programs compared to the one of compiled programs. `Janos` is probably by orders of magnitude less efficient than available commercial tools (implementing stochastic simulators). But it is worth to pay this price as `Janos` has only pedagogical purposes. Namely, the objective is to familiarize students with Monte-Carlo simulation and modeling languages.

John von Neumann

`Janos` is named so in honor of John von Neumann (1903 - 1957). John von Neumann (Hungarian: Neumann Jànos Lajos) was a Hungarian-American mathematician, physicist and computer scientist. Von Neumann is generally regarded as the foremost mathematician of his time and said to be “the last representative of the great mathematicians”; a genius who was comfortable integrating both pure and applied sciences.

He made major contributions to a number of fields, including mathematics (foundations of mathematics, functional analysis, ergodic theory, representation theory, operator algebras, geometry, topology, and numerical analysis), physics (quantum mechanics, hydrodynamics, and quantum statistical mechanics), economics (game theory), computing (Von Neumann architecture, linear programming, self-replicating machines, stochastic computing), and statistics.

He was a pioneer of the application of operator theory to quantum mechanics in the development of functional analysis, and a key figure in the development of game theory and the concepts of cellular automata, the universal constructor and the digital computer.

Installation of Janos

The current version of `Janos` is version 1.2.1.

The first thing you have to do to install `Janos` is to install a Python environment. Python was conceived in the late 1980s by the dutch developer Guido van Rossum (Rossum 1995). It now developed by many people, acting under the direction of a steering committee. There are two major versions of Python: Python 2 and Python 3. Despite of the efforts of the developers of the language, these two versions are not fully compatible. `Janos` uses Python 3.

I recommend the Anaconda environment for Python 3.7 or later.

`https://www.anaconda.com/download/`

On Windows, install Anaconda only for you (not for all users). This will avoid problems when installing packages.

```
37  # 2) Main
38  # -------
39
40  engine = JanosEngine()
41  engine.OpenSession()
42  engine.SetExecutionStatus(S2MLEngine.ACTIVE)
43  engine.SetCurrentWorkingDirectory("examples/EstimationOfPi")
44  engine.LoadScript("NaiveEstimation.janos")
45  engine.CloseSession()
46
```

Figure 1: Python lines to modify to launch a Janos calculation

In addition to Anaconda, it is often convenient to have a good text editor to edit data files and programs. If you have a PC under Windows, you should download and install the Notepad++ text editor.

https://notepad-plus-plus.org/download/

Janos comes with Notepad++ syntax highlighting styles.

Once Python installed, to install Janos you just need to decompress the archive "Janos1.2.1.zip" into a local directory. Source files are the Python file "Janos.py" as well as the directory "src" and its content.

To run Janos you have to open the file Python file "Janos.py" into your Python environment, set up the working directory (line 43) and the name of Janos script file (line 44) as shown on Figure 1, and run it.

The archive contains also:

- A folder doc containing this manual, some presentations and the Notepad++ configuration files.
- A folder examples containing the correction of all exercises proposed in this manual.

Organization of the book

This book is organized into four chapters and three appendices.

Chapter 1 introduces the conceptual framework of Janos, namely systems of stochastic equations and Monte-Carlo simulation.

Chapter 2 presents Janos, i.e. its modeling language, S2ML+DFE, and its commands.

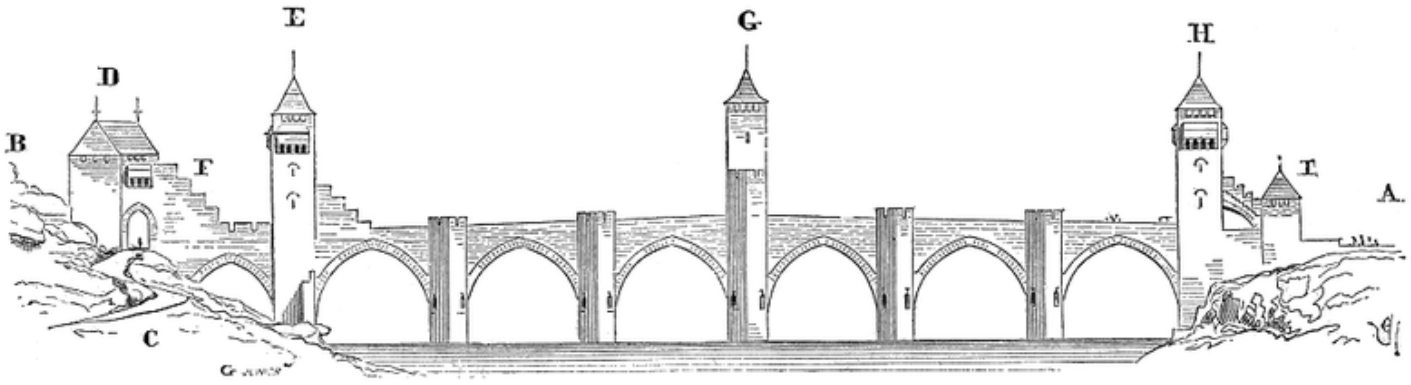
Chapter 3 proposes a series of exercises involving the design of deterministic models.

Chapter 4 proposes a series of exercises involving the design of stochastic models.

Appendix A recalls basics about probabilities and statistics. This appendix is actually a copy of the appendix of the MBRE Book (Rauzy 2022). I added it here for the sake of convenience.

Appendix B gives the EBNF grammar of the language S2ML+DFE.

Finally, Appendix C gives the EBNF grammar of Janos commands.



1. Systems of Stochastic Equations

Key Concepts
- Stochastic simulation
- Systems of stochastic equations
- Uncertainty management
- Sensitivity analyses

Since its introduction in the late 1940s by Stanislaw Ulam and John von Neumann, the Monte-Carlo method is pervasive in sciences and engineering, see e.g. (Metropolis 1987) for historical notes. The main idea behind this method is that the result of a certain calculation is computed based on repeated random sampling and doing statistical analysis. This method applies not only to calculations on stochastic models, but also to calculations on deterministic ones for which there is no analytical solutions or analytical solutions are too difficult to obtain.

This chapter presents systems of stochastic equations. Systems of stochastic equations are probably on the simplest mathematical framework on which Monte-Carlo simulation can be applied. We shall see that a surprisingly large number of engineering problems can nevertheless be solved using this framework.

1.1 Systems of Stochastic Equations

1.1.1 Syntax

Stochastic expressions are built over a universe $\mathscr{C}$ of constants (including typically Boolean constants, integers, real numbers and symbolic constants), a finite or denumerable set $\mathscr{V}$ of variables, and a finite or denumerable set $\mathscr{O}$ of operators (such as Boolean connectives, arithmetic operators, trigonometric operators...). Among these operators, some does not return always the same value. Rather, they pick-up a value at (pseudo-)random in a set of constants, according to certain rules. These operators are called random deviates.

Each variable V of $\mathscr{V}$ comes with its set of possible values, called its domain and denoted $\mathrm{dom}(V)$.

Each operator o of $\mathscr{O}$ comes with its required number of arguments, called its arity and denoted $ar(o)$, $ar(o) \geq 0$. It requires also that its arguments are of certain type (Boolean values cannot be added up). We shall consider, for the sake of simplicity, an operator that can take a varying number of arguments as a collection of distinct operators, one per possible arity.

Definition 1.1.1 – Stochastic expressions. The set of *stochastic expressions* is the smallest set such that:

- Constants of $\mathscr{C}$ are stochastic expressions.
- Variables of $\mathscr{V}$ are stochastic expressions.
- If o is an operator of $\mathscr{O}$ and $e_1,\ldots,e_{ar(o)}$ are stochastic expressions, then $o(e_1,\ldots,e_{ar(o)})$ is a stochastic expressions.

The above definition characterizes an abstract syntax for stochastic expressions. In practice however, arithmetic and Boolean operators are typically written using their usual syntax, e.g. $x+3*y$, f `and` g `or not` f `and` h. The concrete syntax of expressions in languages of the S2ML+X family is given in Appendix B.

As usual, we denote by $\mathrm{var}(e)$ the set of variables that occur in the stochastic expression e.

Definition 1.1.2 – Systems of stochastic equations. A system of stochastic equations is a finite set of assignments in the form:

$$\begin{aligned} V_1 &:= e_1 \\ &\vdots \\ V_n &:= e_n \end{aligned}$$

Where the V_i's are variables of $\mathscr{V}$ and the e_i's are stochastic expressions.

It is required moreover, that each variable is uniquely defined, i.e. there is exactly one assignment $V := e$ for each variable $V \in \bigcup_{i=1}^{n} \mathrm{var}(e_i)$.

We say that a variable V defined by the assignment $V := e$ depends on the variable W if either $W \in \mathrm{var}(e)$ or there is a variable $U \in \mathrm{var}(e)$ such that U depends on W.

A system of stochastic equations is *data-flow* if none of its variables depends on itself. In the sequel, we shall consider only data-flow systems of stochastic equations.

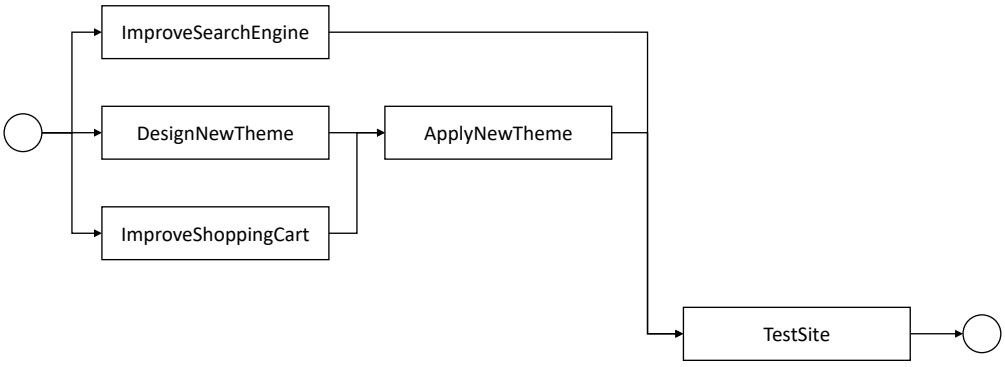


Figure 1.1: The tasks of a website improvement project together with their precedence constraints

■ **Example 1.1 – Website project.** Consider a project that consists in improving the website of an on-line shop. Assume that the project consists of 5 tasks:

- Improving the search engine, which takes about 20 days;
- Design a new theme and apply this new theme to the site, which take respectively about 15

and 3 days;
- Improve customer's shopping cart, which takes about 12 days.
- Extensively test the new version of the site, which takes about 5 days.

There are indeed precedence constraints among these tasks:

- The application of the new theme cannot be done until both the new theme has been designed and the improvements to the customer's shopping cart have been realized.
- The new version of the website can be tested until the new theme has been applied and the search engine has been improved.

Figure 1.1 shows the tasks of the project and their precedence constraints.

A quick calculation shows that the project should be completed in about 25 days.

Assume that the expected benefit made by the company in charge of realizing the project for the on-line shop is 5000 euros. Assume moreover that the penalties this company has to pay is 1000 euros per day of delay the new version of the website it delivers beyond 25 days.

The problem is then to assess the risk that the company takes by accepting the project.

Figure 1.2 shows the system of equations we can write to describe it formally. (for the sake of clarity, we write this system directly at the S2ML+DFE syntax, which will be described Chapter 2.1).

It is easy to verify that this system is actually data-flow.

Now, the reader noticed that we wrote that improving the search engine takes *about* 20 days, which means that there is some uncertainty in the duration of this tasks as well as on the others. It is often assumed that duration tasks in projects obeys some triangular (or similar distributions). Triangular distributions are characterized by three values: a lower bound, an upper bound and a mode, which is located in between the two bounds and that represents the most probable value (see Appendix A.2.1 for more details).

Using triangular deviates, which implements triangular distributions, we could for instance rewrite the definition of variables defining the duration of the tasks as follows.

```
1 ImproveSearchEngine.duration := triangularDeviate(15, 25, 20);
2 DesignNewTheme.duration := triangularDeviate(12, 20, 15);
3 ApplyNewTheme.duration := triangularDeviate(2, 4, 3);
4 ImproveShoppingCart.duration := triangularDeviate(10, 20, 12);
5 TestSite.duration := triangularDeviate(4, 7, 5);
```

■

As the reader shall see through the exercises and problems at the end of this chapter, many properties of systems can be studied by means of such systems of equations.

1.1.2 Semantics

The semantics of stochastic expressions is defined recursively. Each operator o of $\mathscr{O}$ which is not a random deviate is interpreted as a partial function $[\![o]\!]$ from $\mathscr{C}^{ar(o)}$ into $\mathscr{C}$. Random deviates are also interpreted as partial functions, but they take implicitly an additional parameter, which typically a real number in the range $[0,1]$. The additional parameter is used as the (unique) source of randomness. A random deviate r is thus interpreted as a partial function $[\![o]\!]$ from $\mathscr{C}^{ar(r)} \times [0,1]$ into $\mathscr{C}$.

To get these implicit parameters, we assume given an infinite sequence of numbers in $[0,1]$. Let zs be such a sequence, then $\text{next}(zs)$ denotes the operation consisting in removing the first number of the sequence and returning it. The calculation of the value of a stochastic expression may "consume" in this way the first numbers of the sequence.

The semantics of stochastic expressions is defined as follows.

```
// Individual task description
ImproveSearchEngine.completion :=
  ImproveSearchEngine.start + ImproveSearchEngine.duration;
ImproveSearchEngine.duration := 20;
DesignNewTheme.completion :=
  DesignNewTheme.start + DesignNewTheme.duration;
DesignNewTheme.duration := 15;
ApplyNewTheme.completion :=
  DesignNewTheme.start + DesignNewTheme.duration;
ApplyNewTheme.duration := 3;
ImproveShoppingCart.completion :=
  ImproveShoppingCart.start + ImproveShoppingCart.duration;
ImproveShoppingCart.duration := 12;
TestSite.completion := TestSite.start + TestSite.duration;
TestSite.duration := 5;

// Project description
ImproveSearchEngine.start := 0;
DesignNewTheme.start := 0;
ImproveShoppingCart.start := 0;
ApplyNewTheme.start :=
  max(DesignNewTheme.completion, ImproveShoppingCart.completion);
TestSite.start :=
  max(ImproveSearchEngine.completion, ApplyNewTheme.completion);
projectDuration := ceil(TestSite.completion);

// Expected profit
expectedProfit :=
  if projectDuration<=25
  then 5000
  else 5000 - 1000*(projectDuration - 25);
```

Figure 1.2: System of equations describing the website project

Definition 1.1.3 – Semantics of stochastic expressions. Let σ be an assignment of the variables of $\mathscr{V}$ and zs be an infinite sequence of real numbers in $[0,1]$. Then,

- $[\![c]\!]_{\sigma,zs} = c$, if c is a constant of $\mathscr{C}$.
- $[\![V]\!]_{\sigma,zs} = \sigma(V)$, if V is a variable of $\mathscr{V}$.
- $[\![(o\ e_1\ \ldots\ e_{ar(o)})]\!]_{\sigma,zs} = [\![o]\!]\left([\![e_1]\!]_{\sigma,zs},\ldots,[\![e_{ar(o)}]\!]_{\sigma,zs}\right)$ if o is an operator of $\mathscr{O}$ but not a random deviate.
- $[\![(rd\ e_1\ \ldots\ e_{ar(rd)})]\!]_{\sigma,zs} = [\![rd]\!]\left([\![e_1]\!]_{\sigma,zs},\ldots,[\![e_{ar(rd)}]\!]_{\sigma,zs},\mathrm{next}(zs)\right)$ if rd is a random deviate of $\mathscr{O}$.

As all systems of stochastic equations we consider are data-flow, there is always a way to order their equations so that the equation $V := e$ comes after the equations defining variables occurring in e. The semantics of a system of stochastic equations consists is building step by step, i.e. equation by equation, a variable valuation, once the equations ordered. Systems of stochastic equations are thus considered as suitably ordered lists of equations.

We denote by $[]$ the empty list of equations and by $V := e;\Lambda$ the list made of the equation $V := e$ followed by the list of equations Λ.

The construction of the variable valuation works as follows.

Definition 1.1.4 – Semantics of systems of stochastic equations. Let σ be an assignment of the variables of $\mathscr{V}$ and zs be an infinite sequence of real numbers in $[0,1]$. Then,

- $[\![[]]\!]_{\sigma,zs} = \sigma$.
- $[\![V := e;\Lambda]\!]_{\sigma,zs} = [\![\Lambda]\!]_{\tau,zs}$, where τ is the variable valuation such that:

$$\tau(W) = \begin{cases} [\![e]\!]_{\sigma,zs} & \text{if } W = V \\ \sigma(W) & \text{otherwise} \end{cases}$$

■ **Example 1.2 – Website project (bis).** Consider again the system of stochastic equations of example 1.1. We can reorder its equations as follows to be able to calculate them in order. Here follows a candidate order.

1	`ImproveSearchEngine.duration`
2	`DesignNewTheme.duration`
3	`ApplyNewTheme.duration`
4	`ImproveShoppingCart.duration`
5	`TestSite.duration`
6	`ImproveSearchEngine.start`
7	`ImproveSearchEngine.completion`
8	`DesignNewTheme.start`
9	`DesignNewTheme.completion`

10	`ImproveShoppingCart.start`
11	`ImproveShoppingCart.completion`
12	`ApplyNewTheme.start`
13	`ApplyNewTheme.completion`
14	`TestSite.start`
15	`TestSite.completion`
16	`projectDuration`
17	`expectedProfit`

The calculation of the value of `ImproveSearchEngine.duration` is performed first. It involves the random deviate `triangularDeviate`, which will consumes a number from the infinite sequence. Then, the value of `DesignNewTheme.duration` is calculated, which consumes also a number from the infinite sequence. And so on until the calculation of `expectedProfit`.

■

At this point, two important remarks can be made:

- No matter the initial variable valuation, the final variable valuation will be the same.
- The calculation of the final variable valuation is fully deterministic and can thus be reproduced at will.

This is well and good, but does not tell us how to calculate probabilistic indicators on variables of systems of stochastic equations, e.g. their expected value or their distribution. In the above example, it would be still possible to analyze expressions and to get some manageable analytical solutions. In the general case, where complex expressions and large number of variables may be involved, it is impossible, at least in practice, to obtain analytical solutions.

Fortunately, Monte-Carlo simulation provides a practical way to estimate probabilistic indicators.

1.2 Monte-Carlo Simulation

1.2.1 Rational and Principle

In many system operation situations, we face an uncertain environment and nevertheless we have to make decisions. *A priori*, this seems impossible: how to make a decision if we do not know what will happen and what will be the consequences of our decisions?

In many cases however, we have at hand some information about the system under study and its environment, in form of statistical data resulting from operation feedback. We can use these data to design stochastic models, i.e. models in which the system evolves at random. Systems of stochastic equations defined in the previous section are examples of such models.

The randomness in stochastic models is controlled in two ways: first, executions of these models describe are not purely random, but governed by the statistical data; second, they are reproducible. In a word, they are pseudo-random executions.

```
MonteCarloSimulation(model, numberOfExecutions, sequenceOfNumbers):
  Initialize(model)
  for i in [1, numberOfExecutions]:
    Execute(model, sequenceOfNumbers)
    UpdateIndicators(model)
  MakeStatisticsOnIndicators(model)
```

Figure 1.3: Pseudo-code for the Monte-Carlo simulation

It is thus possible to draw a large number of such pseudo-random executions, and to make statistics on them. This is the basic principle of *Monte-Carlo simulation*.

This method is widely used in many different areas: from epidemiological studies to financial markets going through insurances, nuclear safety... We shall review thus review its fundamental constituents and look at some applications.

Assume given a model, written in some modeling language. The experiment performed on the model consists in some calculations, that we shall call *execution* of the model in the sequel. We assume thus given a function `Execute` in charge of performing the execution. What this function depends indeed on the modeling language and the experiment at stake. But they key idea is that at some steps of the execution of the model, `Execute` consumes values from an infinite sequence of numbers and make decisions according to these values, just as we have done for systems of stochastic equations. Two successive executions using the same infinite sequences of numbers may be totally different because the numbers of the sequence consumed by the first execution can be different from those consumed by the second one. By running `Execute` a sufficiently large number of times, we can make statistics on values taken by the indicators of interest in the different executions.

The generic Monte-Carlo simulation algorithm is as sketched Figure 1.3.

Monte-Carlo simulation is a quite generic algorithm, which is relatively simple to implement and works satisfactorily in many situations. However, one must take care at the following issues.

First, one must control where the randomness comes from. If the model aims at analyzing a phenomenon on which there is an uncertainty, then one must collect, prior to any experiment, suitable statistical data on the phenomenon. Monte-Carlo simulation is not a magic wand, or to put more crudely, it is subject to the well known adage: garbage-in, garbage-out.

Second, it is of primary importance to check that the model meets the reality. This is in general not an easy task. By definition, programs implementing Monte-Carlo simulations are difficult to debug as they perform much too many calculations for the programmer to follow them "by hand". Even if we assume that the program is free of bugs, there are in general also much too many possible executions (or even groups of similar executions) of the model for the analyst to check them exhaustively. In the infancy of computerized methods, execution times were the major cause which prevented the deployment of the method. With the increase of computation power, the problem of the validation of models becomes the most important one (although execution times are still an issue).

Third, Monte-Carlo simulation relies on the generation of numbers at random or pseudo-random, according to some predefined distributions. It is thus of primary importance that the generation mechanism mimics correctly randomness. This problem has been controversial for many years, but is now considered as solved satisfactorily, see Appendix A.4.

Fourth, Monte-Carlo simulation provides statistics on indicators. It is thus important to design significant indicators and to interpret correctly these statistics. In particular, statistics do not give exact results, but results that are probably correct. The larger the sample, the more accurate the result, but also the more costly the experiment. It is thus important to find a good trade-off between

accuracy and computation cost.

1.2.2 Two Uses of Monte-Carlo Simulation

In the context of model-based systems engineering, Monte-Carlo simulation is used for two main purposes: the direct calculation of performance indicators and sensitivity analyses.

Calculation of performance indicators

The most obvious reason of using Monte-Carlo simulation consists in calculating performance indicators on a system subject to aleatory uncertainties. Example 1.1 illustrates this usage. Here follows another one.

■ **Example 1.3 – Simple queue.** Consider a service with a unique line that serves clients in the order they arrive. Assume the delay between two successive client arrivals is exponentially distributed with a mean time between arrival of 3 minutes. Assume moreover that the time to serve a client varies uniformly between 1 and 4 minutes. The question is whether this service line is efficient enough, i.e. if the waiting time of clients is not too high.

In this system, there are two sources of aleatory uncertainty: the delay between two successive arrivals of clients and the service time.

To answer the question, an analytical reasoning may be possible, but a Monte-Carlo can also do the job, probably at a much lesser cost.

The idea is to design a model to simulate successive arrivals of clients and their service. For each client i, $i = 1, 2 \ldots$ we have three dates:

- The date a_i at which the client arrives in the system.
- The date s_i at which the service of the client starts.
- The date d_i at which the service of the client is completed.

According to the specifications of the problem:

- a_i is equal to a_{i-1} plus a random delay, which is exponentially distributed with a rate $1/3$ minutes^{-1} (posing $a_0 = 0$)
- s_i is the maximum of a_i and d_{i-1} (assuming the service of the client i starts immediately after the service of the client $i-1$ is completed and posing $d_0 = 0$).
- d_i is equal to s_i plus a random delay, which is uniformly distributed between 1 and 3 minutes.

The waiting time w_i of the client i is thus $d_i - s_i$.

The S2ML+DFE code for this example is provided with the `Janos` distribution.

The following table reports the results of a Monte-Carlo simulation with 10,000 executions on 20 client arrivals (i.e. about 1 hour of operation). In this table w_i is the mean value of the waiting time of the ith client.

client	1	2	3	4	5	6	7	8	9	10
w_i	0.0	0.86	1.43	1.89	2.28	2.59	2.86	3.06	3.26	3.44
client	11	12	13	14	15	16	17	18	19	20
w_i	3.60	3.76	3.89	4.02	4.17	4.27	4.39	4.47	4.56	4.67

This experiment is sufficient to show the following picture: the waiting time of clients increases, quickly at the beginning, regurlarly after. It tends to slowly stabilizes around 5 minutes after a while.

Whether this situation is acceptable or not cannot be told be the model. ■

Sensitivity analyses

The other important use of Monte-Carlo simulation in the context of model-based systems engineering is the so-called sensitivity analyses. These analyses aim at testing the robustness of decisions one makes based on performance indicators calculated via models.

It is in general the case that models involve parameters that are known only up to a certain uncertainty. E.g. the arrival rate and the lower and upper bounds for the service time of Example 1.3 are most probably known only approximately.

It therefore worth to test whether the decision made would change if the values of these parameters would change slightly. It is always possible to test the impact of the variation of one parameter, *mutatis mutandis*. However, such an approach does make it possible to test the impact of simultaneous variations of the parameters. When the number of parameters is small, it is possible to enumerate all possible cases, e.g. an increase of both the arrival rate and the bounds of service time, an increase of the former and a decrease of the latter... When the number of parameters gets larger, enumerating all of the cases, or even the most significant ones, becomes impossible for the sake of combinatorial explosion of the number of cases.

A solution consists in defining associating a distribution of values with each parameter, rather that a unique value, and to perform a Monte-Carlo simulation. This idea is illustrated in the following example.

■ **Example 1.4 – Manchester.** Figure 1.4 shows a simplified map of the region of Manchester. Jane Doe, a talented systems engineer who lives in Rochdale, is offered a new job in Stockport. The job is interesting but she is a bit worrying about travel times, as she will have to drive from home to work every morning. She wants thus to determine how long will it take. Distances in miles are indicated on the map, but how the driving time is related to these distances depends on the traffic, which is known only up to an uncertainty.

Jane Doe designs first a model to determine the travel from Rochdale to Stockport at the average speed of say 30 miles per hour, assuming a regular traffic. Her model, provided with the distribution of `Janos`, hard codes the Disjkra's algorithm to find shortest paths in a graph, i.e.

- It defines a travel time between each city linked with a direct road, according to the distance and the speed.
- It defines the travel time to each city C as the minimum, over the cities P that precede C, of the travel time to P augmented with the travel time from P to C.

For instance, the travel time to Manchester is the minimum of:

- The travel time to Rochdale (hence 0) augmented with the travel time from Rochdale to Manchester.
- The travel time to Bury augmented with the travel time from Bury to Manchester.
- The travel time to Oldham augmented with the travel time from Oldham to Manchester.

Using this model, Jane Doe determines that the travel time at an average speed of 30 miles per hour from Rochdale to Stockport is 62 minutes.

Now, it is a bit arbitrary to set up the average speed to 30 miles per hour. It can be less and it can more, depending on the traffic, which may be different from one road to the other. She can of course re-do the calculation for an optimistic one of 35 miles per hour. and a pessimistic average speed of 20 miles per hour. Doing so, she would obtained 44 and 93 minutes respectively.

These three values are probably not informative enough. In particular, it is quite optimistic to assume that the traffic will be clear on all the inter-city roads. Jane Doe decides therefore to perform a sensitivity analysis, assuming that the speed on each inter-city road is uniformly distributed from 20 miles per hour to 35 miles per hour.

Running a Monte-Carlo simulation (over 10000 executions of the models), she obtains the following results.

- The average travel time is 72 minutes, with a standard-deviation of about 6 minutes.
- In one case out of four, the travel takes between 44 and 68 minutes. In one case out of four, it takes between 68 and 72 minutes. In one case out of four, it takes between 72 and 76 minutes. Finally in the last quarter of the cases, it takes from 76 to 89 minutes.

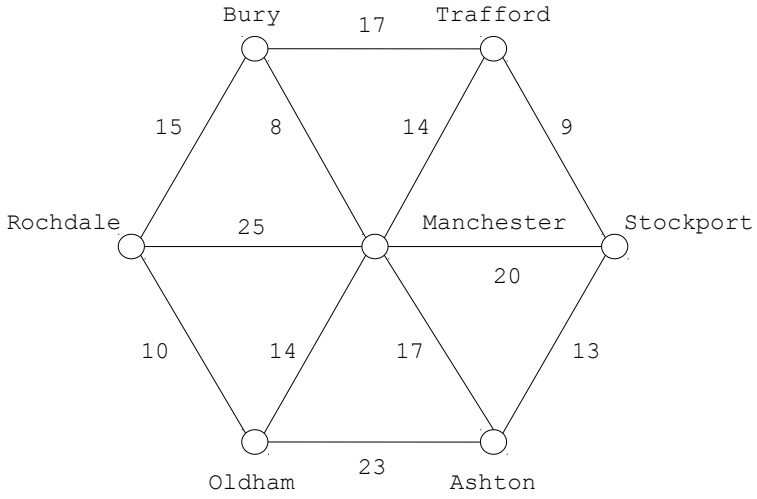


Figure 1.4: A simplified map of the region of Manchester

This gives her a clear picture of much time she will spend driving every day if she accepts the position, and probably good reasons to go for the train. ∎

1.3 Discussion

1.3.1 How many executions are needed?

The sample size is an important feature of any empirical study in which the goal is to make inferences about a population from a sample. This applies indeed to Monte-Carlo simulation. Larger number of executions lead in general to an increased precision when estimating unknown parameters. The law of large numbers and the central limit theorem are useful mathematical tools to describe this phenomenon. Namely, by the strong law of large numbers, we now that the estimated mean of a distribution $\bar{x}$ tends with a probability 1.0 to its mean μ as n goes to infinity. Moreover, by the central limit theorem, we can calculate an error factor and a confidence interval, as defined by equations A.12 and A.13.

These results can be interpreted as follows.

- For a fixed number of executions n, if we want to increase our confidence in the result, i.e. to reduce α, we need to widen the confidence interval, and vice-versa.
- If we want to increase our confidence in the result without widening the confidence interval or to narrow down the confidence interval without decreasing our confidence of the result, we need to increase n.
- However, the improvement is governed by the $\frac{1}{\sqrt{n}}$ factor: to reduce the width of the confidence interval by 2, we need 4 times more executions.

Note that, and it is a remarkable fact, the above results do not depend on the size of the population, i.e. in our case of the number of possible executions. This is what makes polls possible. With a relatively limited sample, it is possible to get a good picture of the whole population.

In the framework of model-based systems engineering, executions are in general not too computationally costly as models have a limited size and systems are observed on limited mission-times. As a rule of thumb, it is thus reasonable not to do less than 10,000 executions. This number is also sufficient in many cases.

In some cases, the increase in precision for larger sample sizes is minimal, as illustrated by the following classical example.

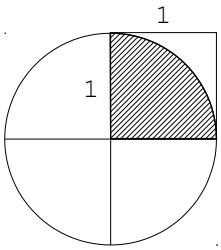


Figure 1.5: Estimation of π by sampling points in the unit square

■ **Example 1.5 – Estimation of of π.** Assume we want to estimate the value of π. We can design a Monte-Carlo simulation to do so.

The idea is as follows. Each execution consists in drawing two numbers x and y uniformly at random in the range $[0,1]$. The point (x,y) lies thus somewhere is the unit square. Now there are two cases: either the point lies in the (quarter of the) unit disk (the region bounded by the unit circle), i.e. $x^2+y^2 \leq 1$, or not, as illustrated Figure 1.5.

By the law of large numbers, when the number of points n in the sample tends to infinity, the proportion k/n of points that lie within the unit tends to the surface of that unit disk, i.e. $\frac{1}{4}(\pi \times r^2) = \pi/4$. To get an estimate $\overline{\pi}$ of π it suffices thus to take the proportion of points (x,y) in the sample such that $x^2+y^2 \leq 1$ and to multiply it by 4.

In theory, this works fine. In practice, here follows the estimate obtained from different size of samples (by means of a S2ML+DFE model).

n	$\overline{\pi}$	95% confidence range
10,000	3.1688	$[3.13699, 3.20061]$
100,000	3.14876	$[3.13861, 3.15891]$
1,000,000	3.14404	$[3.14082, 3.14725]$
10,000,000	3.14136	$[3.14035, 3.14238]$

As the reader can see, the improvement obtained by taking a sample of 10 millions of executions rather than one 100 times smaller is not very significant, as the result is coarse anyway.

This does not say that the law of large numbers and the central limit theorem do not apply in this case, just that the convergence rate towards the limit is low. ■

1.3.2 Rare events

It is of primary importance to understand that above considerations apply if the mean of the distribution is not too low. In case of rare events, the picture is different as we have to consider not only the error factor, but also the relative error factor, i.e. $\frac{\mathrm{EF}_\alpha(x)}{\overline{x}}$.

Consider that we want to perform a Monte-Carlo simulation to estimate a parameter that is distributed according to a Bernoulli distribution (see page 61) of parameter p. Then, the variance of this parameter is $p(1-p)$. We have thus:

$$\begin{aligned}
\frac{\mathrm{EF}_\alpha(x)}{\overline{x}} &= \frac{t_\alpha \overline{\sigma}(x)}{\sqrt{n}\overline{x}} \\
&\approx t_\alpha \times \frac{\sqrt{\overline{x}(1-\overline{x})}}{\sqrt{n\overline{x}^2}} \\
&= t_\alpha \times \sqrt{\frac{(1-\overline{x})}{n\overline{x}}}
\end{aligned}$$

For a fixed n above quantity tends to infinity as p (and thus $\overline{x}$) tends to 0.

To fix ideas, assume that we want the relative error no to exceed a certain ratio $\frac{1}{\tau}$. Then, n must be as follows.

$$n \approx (t_\alpha \tau)^2 \times \frac{(1-p)}{p}$$

If p is small, then $\frac{(1-p)}{p} \approx \frac{1}{p}$.

Assume we want to consider 95% confidence intervals, i.e. $t_\alpha \approx 2$, and that we accept a relative error of 50%, i.e. $\tau = 2$. Then, $(t_\alpha \tau)^2 \approx 16$. This means that if $p = 1.00 \times 10^{-4}$, we need to perform about $150,000$ executions. If we accept only a relative error of 10%, i.e. $\tau = 10$ and $(t_\alpha \tau)^2 \approx 400$, we need to perform about 4 millions executions.

If now $p = 1.00 \times 10^{-6}$, the number of executions to be performed will be respectively 15 and 400 millions executions.

To put things the other way round, if $p \approx 10^{-k}$ and we perform 100×10^k executions, then our relative error is about 20%, i.e. there are 95% chances that μ lies between 80% $\bar{x}$ and 120% $\bar{x}$.

This explains why Monte-Carlo simulation cannot be applied directly to rare events. Acceleration techniques are required. An exposition of these techniques goes beyond the scope of this book.

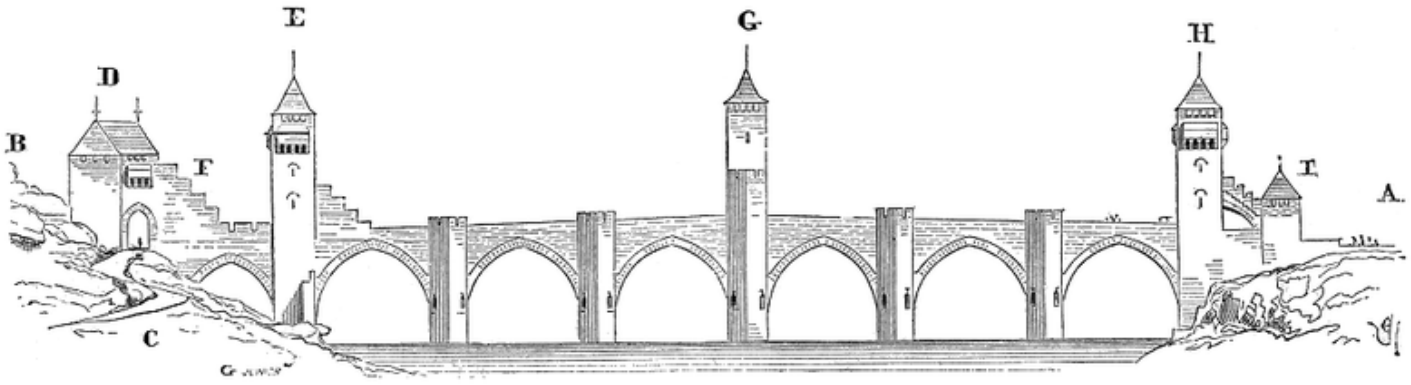
1.4 Further Readings

The book by Dubi (Dubi 2000) provides an introduction to applications of the Monte-Carlo method to systems engineering.

The book by Zio (Zio 2013) gives a snapshot of the use of Monte-Carlo simulation in the context of reliability engineering.

The book by Robino and Tuffin (Rubino and Tuffin 2009) gives a comprehensive overview of acceleration techniques for rare event simulation.

Finally, the readers interested by the notion of randomness will probably enjoy Chaitin's book (Chaitin 2001).



2. Get It Applied with Janos

Key Concepts
- S2ML+DFE
- `Janos`

This chapter presents the tool `Janos`, a pedagogical Monte-Carlo simulator, and its associated language S2ML+DFE.

S2ML+DFE is one of the simplest modeling language one may imagine, yet it proves useful in the context of model-based systems engineering. Its underlying mathematical model consists of systems of data-flow stochastic equations (DFE). On top of this mathematical models, it provides object- and prototype-oriented constructs to architect models. These constructs are gathered into S2ML. It makes it possible to check (simple) properties of systems.

`Janos` comes as a command interpreter, making it possible to perform various studies. This chapter specifies S2ML+DFE as well as `Janos` commands to manage models and launch simulations.

The current version of `Janos` is developed in Python, for pedagogical purposes only. It is by orders of magnitude less efficient than available academic and commercial tools. Its primary objective is to familiarize students with Monte-Carlo simulation and modeling languages of the S2ML+X family.

2.1 The S2ML+DFE Modeling Language

2.1.1 Overview

S2ML+DFE implements all of the constructs of S2ML seen in the chapter "Architecture of Models" of the MBRE Book (Rauzy 2022). In addition, it implements specific ports and connections. There are three types of ports in S2ML+DFE:

- Parameters, which are essentially named values. As variables, parameters are defined by means of equations, however their definition must be deterministic and involve only other parameters. Their is decided once for all and stays the same in all executions.

– Variables, which are the "real" ports. The value of variables may vary from one execution to the other.
– Observers, which are the indicators whose values are reported when doing deterministic calculations and on which statistics are made when performing Monte-Carlos simulations.

In S2ML+DFE , the distinction between parameters and variables is somehow arbitrary. However, it is not in other languages of the S2ML+X family. This is the reason why they are kept separated in S2ML+DFE as well.

There is only one type of connections in S2ML+DFE: namely equations in the form "$v := E$", where v is a variable and E is an expression that depends on parameters and other variables. The set of equations encoded by a S2ML+DFE must be data-flow, i.e. that a variable (or a parameter) cannot depend eventually on itself.

Two types of calculations can be made with `Janos`:

– Deterministic calculations that consists in calculating the values of variables and then the values of observers.
– Monte-Carlo simulations that consists in repeating the above process a number of times, then in making statistics on the values of observers.

Deterministic calculations are possible even if the model involves random deviates. The latter are then set to the default values, e.g. $\frac{l+h}{2}$ for a uniform deviate with lower bound l and upper bound h, see Appendix A.2 and **??** for more details.

We shall now review modeling constructs that are specific to S2ML+DFE.

2.1.2 Domains

Parameters, variables and observers take their values into domain, which can be either predefined domains or user defined domains.

Predefined domains are the following.

Boolean The domain with two values `false` and `true`.

Integer The set of integers, e.g. `42`, `-33`...

Real The set of floating point numbers, e.g. `-373.15`, `1.23e-6`...

Symbol The set of all possible symbolic constants, e.g. `LOW`, `XYZ`, `FAILED_UNDETECTED`... Although this is not required by the syntax of the languages, symbolic constants are usually written in upper case letters.

User defined domain are either *enumeration* of symbolic constants or *range* of integers. They are declared, using the `domain` directive, at the top block level (like classes). E.g.

```
1 domain OnOff {ON, OFF}
2 domain Row [1, 8]
```

2.1.3 Ports

S2ML+DFE defines three types of ports: parameters, variables and observers.

Parameters

Parameters are constant quantities introduced in models for the sake of clarity. They are introduced by the keyword `parameter`, have a domain and a value, which can be modified subsequently, e.g. after instantiation. E.g.

```
1 parameter Real failureRate = 1.0e-3;
```

Note that it is possible to define the value of a parameter by an expression that depends itself on the value of other parameters. E.g.

```
parameter Real failureRate1 = 1.0e-3;
parameter Real failureRate2 = 2.0e-3;
parameter Real failureRate = failureRate1 + failureRate2;
```

However, a parameter cannot depend eventually on itself. It cannot depends neither on variables or observers.

Variables

Variables are the main category of ports. Variables are declared together with their domain (their type). E.g.

```
Boolean input, output;
OnOff state;
```

The value of variables is defined by equations (see Section 2.1.4). A variable may depend on parameters and other variables. It cannot depend eventually on itself nor on observer.

Observers

Observers are typed and calculated like variables. They are the results calculated and printed out by `Janos`. They are declared similarly to parameters, but introduced by the keyword `observer`. E.g.

```
observer Real production = Unit1.production + U2.production;
```

Observers can depend on parameters and variables. They cannot depend on other observers.

2.1.4 Connections

Any S2ML+DFE model is eventually equivalent to a set of *stochastic equations* $\{v_1 := E_1, \ldots, v_n := E_n\}$, where the v_i's are variables and the E_i's are stochastic expressions. E.g.

```
C.AM1.output := if C.AM1.state==WORKING then C.AM1.input else false;
```

Stochastic expressions are built using parameters, variables, constants and operators. Constants and operators that can be used to write expressions are described Appendix B. They are common to most, if not all, languages of the S2ML+X family.

For instance, in the above equation, the left-hand variable `C.AM1.output` depends on the variables showing up in the right-hand expression, i.e. `C.AM1.state` and `C.AM1.input`, and on the variables these variables depend on, and so on.

The set S of equations represented by a S2ML+DFE model M must obey the following conditions:

- For each variable v of M, there must be exactly one equation in S whose left member is v.
- S must be *data-flow*, i.e. that no variable v of M can depend on itself in S.

2.1.5 S2ML constructs

Aside the elements presented in the two previous sub-sections, S2ML+DFE provides all S2ML constructs presented in Chapter "Architecture of Models" of the MBRE Book (Rauzy 2022):

- Blocks (that are prototypes in the sense of object-oriented theory);
- Classes;
- Composition;
- Cloning (of blocks);

– Instantiation (of classes);
– Inheritance;
– Paths and aggregation;
– Operators and functions.

Note that attributes are not used in S2ML+DFE.

2.1.6 Application to the case study

To illustrate S2ML+DFE constructs, consider again the website project described in example 1.1. Figure 2.1 shows a possible S2ML+DFE model for this example.

```
class Task
  Real start, duration, completion;
  assertion
    completion := start + duration;
end

block Website
  Task ImproveSearchEngine;
  Task DesignNewTheme;
  Task ApplyNewTheme;
  Task ImproveShoppingCart;
  Task TestSite;
  Integer projectDuration;
  assertion
    ImproveSearchEngine.duration := triangularDeviate(15, 25, 20);
    DesignNewTheme.duration := triangularDeviate(12, 20, 15);
    ApplyNewTheme.duration := triangularDeviate(2, 4, 3);
    ImproveShoppingCart.duration := triangularDeviate(10, 20, 12);
    TestSite.duration := triangularDeviate(4, 7, 5);
    ImproveSearchEngine.start := 0;
    DesignNewTheme.start := 0;
    ImproveShoppingCart.start := 0;
    ApplyNewTheme.start :=
      max(DesignNewTheme.completion, ImproveShoppingCart.completion);
    TestSite.start :=
      max(ImproveSearchEngine.completion, ApplyNewTheme.completion);
    projectDuration := ceil(TestSite.completion);
  observer Integer ProjectDuration = projectDuration;
  observer Integer ExpectedProfit =
    if projectDuration<=25
    then 5000
    else 5000 - 1000*(projectDuration - 25);
end
```

Figure 2.1: S2ML+DFE model for the website project

First, a class `Task` is declared to implement the generic notion of task. A task is essentially characterized by three real valued variables: its start date `start`, its duration `duration` and its completion date `completion`. The two former variables are not defined in the class. The value of `start` depends actually of the position of the task in the project. The duration is undefined for the sake of genericity of the class `Task`. It would have been possible to assume that the duration of all tasks obeys a triangular distribution and to declared the parameters of this distribution as... parameters. E.g.

```
class Task
  parameter Real lowerBound = 0;
  parameter Real upperBound = 0;
  parameter Real mode = 0;
  Real start, duration, completion;
  assertion
    duration := triangularDeviate(lowerBound, upperBound, mode);
    completion := start + duration;
end
```

Then, to set the value of parameters at instantiation. E.g.

```
  Task ImproveSearchEngine
    parameter Real lowerBound = 15;
    parameter Real upperBound = 25;
    parameter Real mode = 20;
  end
```

The value of `completion` can be set at task level as it is always defined in the same way.

The model given in Figure 2.1 declares two observers, one for the project duration, the other one for the expected benefit, assuming these are the two quantities of interest.

2.2 Janos Scripts

2.2.1 Commands and Scripts

`Janos` comes as a command interpreter. It reads *scripts*, i.e. sequences of commands, into files and executes one after the other these commands.

Commands are used to:

- Read and write models into files.
- Instantiate models, i.e. transform them into sets of equations from which calculations can be performed.
- Set up the various parameters of calculations.
- Launch calculations and print out results into files.

In a script, each command spans normally written over one line. However , if the line contains a character "\", then all character starting from that one until the end of the line are ignored and the line is merged with the next line.

Figure 2.2 shows a typical `Janos` script, which can be used to assess the model we designed for our Website project example given in Figure 2.1.

In this script, line 1 is a comment. Comments start with a character "#" and spread until the end of the line.

The command `load model...` line 4 loads a model written in the file `website.dfe` located in the current working directory. It is also possible to load script, using the option `script` instead of the option `model`.

Although this is not strictly mandatory, S2ML+DFE models are stored in file with the extension `.dfe` and `Janos` scripts into files with the extension `.janos`. The S2ML+X toolbox distribution includes highlighting styles for the text editor `Notepad++` for each language and for the scripts.

The command `flatten` line 8 flattens the model, i.e. transforms it into a set of equations on which calculations can be performed. This step must be performed prior to any calculation. Commented lines 5 and 9 show how the print out the model as designed and the model as assessed.

Lines 12- 23 contains commands to set up the parameters of the Monte-Carlo simulation. We shall describe their respective roles below.

```
1  # Script file for the Website project example
2
3  # Loading of the model
4  load model "website.dfe"
5  # print model output="model.dfe"
6
7  # Flattening of the model
8  flatten model
9  # print target-model output="instantiated-model.dfe"
10
11 # Parameters of the stochastic simulation
12 set seed 23456
13 set number-of-tries 10000
14
15 profile set mean profile1 true
16 profile set standard-deviation profile1 true
17 profile set confidence-interval-90 profile1 false
18 profile set confidence-interval-95 profile1 true
19 profile set confidence-interval-99 profile1 false
20 profile set extrema profile1 true
21 profile set quantiles profile1 false
22 profile set distribution profile1 false
23 profile set cumulative-distribution profile1 false
24
25 # Stochastic simulation
26 compute observers Website mission-times=[0] output="website.csv" \
27     mode=write
```

Figure 2.2: A typical `Janos` script

Finally, the command `compute observers...` lines 26 and 27 launches the actual computation.

The first argument of this command, i.e. the option `observers`, just recalls that only values of observers are printed out in the result file. Therefore, if you are interested in a particular value, you must create the corresponding observer.

The second argument is the name of the top-level block on which the calculation are performed. In this case, this block is named `Website`.

The third argument is the list of mission-times at which the calculation must be performed. It is actually often the case in model-based systems engineering that the quantities to be calculated, even if they can be computed deterministically, vary through the time. The built-in expression `mission-time()` makes it possible these variations into account. It is given successively the values passed in argument of the command `compute`, and the calculations are performed for each of these values. Mission times are given as list of floating point numbers separated with commas and surrounded with square brackets. In the script of Figure 2.2, the calculation is performed only at time 0. To perform at times 0, 100, 200 and 300, it suffices to pass the argument `mission-times=[0, 100, 200, 300]`. By default, i.e. if no argument `mission-times` is given, the calculation is performed at time 0.

The last two arguments specify respectively the name (actually the path) of the result file and its opening mode. Result files can be opened either in `write` mode or in `append` mode. In the former, if there existed a file with the given name before the execution of the command, its the content is erased and replaced by the results of the command. In the latter, the results of the

command are appended to the file.

2.2.2 Result files

All result files produced by tools of the S2ML+X Toolbox are at the tsv format. The tsv format, also called csv format, is a textual format that is easy to load into spreadsheet tools such as Microsoft Excel™.

Each line of the file encodes a row of the spreadsheet. Values (columns) are separated by tabulation characters.

The result file for the "Website" model is as follows.

```
Model Website
Number-of-tries 10000
Execution-time  00:00:06.53
Observer  ProjectDuration
  Mission-time  0.0
      Mean  Standard-deviation  95% confidence interval   Minimum Maximum
      25.51 1.76  25.47 25.54 19  31
Observer  ExpectedProfit
  Mission-time  0.0
      Mean  Standard-deviation  95% confidence interval   Minimum Maximum
      4030.80 1208.89 4007.11 4054.49 -1000 5000
```

Such an output format makes easy the post-processing of results within your favorite spreadsheet tool or using Python.

2.2.3 Launching a Monte-Carlo simulation

In order to perform a Monte-Carlo simulation rather than a simple deterministic calculation, it suffices to set up the option `number-of-tries` to the number of executions we want to perform.

The simplest script to perform a Monte-Carlo simulation is thus as follows.

```
load model "website.dfe"
flatten model
set number-of-tries 10000
compute observers Main output="result.csv" mode=write
```

To perform deterministic calculations, or equivalently calculations with default values of random deviates, it suffices to set the number of tries to 0.

It is possible to perform the simulation at different mission-times using the option `mission-times`:

```
compute observers Main mission-times=[1000, 2000, 3000] \
    output="result.csv" mode=write
```

The Monte-Carlo simulation involves drawing numbers at pseudo-random. As explained in Appendix A.4, algorithmic random number generators calculate the next value to be generated from the previous one. It is possible to set the initial value of the sequences, called the *seed*, via the command `set seed`. E.g.

```
set seed 12345
```

2.2.4 Defining the statistics to be made

The command `compute observers` perform a Monte-Carlo simulation. It makes statistics on the values taken by each observer, at each mission time, over the executions. Appendix A.3 recalls

mathematical definitions of indicators mentioned in this section.

Janos makes it possible to define several *mission profiles* that describe which indicators should be calculated. A *profile* have name and records which indicators should be calculated and which not. Profiles can be created, cloned, deleted and configured by means of the command `profile`.

By default, the predefined profile `profile1`. It is possible to modify this profile. It is also possible to perform computation with another profile as follows.

```
compute observers Main mission-times=[1000, 2000, 3000] \
    output="result.csv" mode=write \
    profile=myProfile
```

Creation, cloning and deletion

Commands to create a new profile `myProfile` (or any other name), to clone the profile `myProfile` into another profile `myOtherProfile`, and finally to delete the profile `myProfile` are as follows.

```
profile new myProfile
profile clone myProfile myOtherProfile
profile delete myProfile
```

Moments

The current version of Janos makes it possible to calculate the following indicators for each observer and each mission time:

- The expected value, or mean.
- The standard-deviation.
- The 90%-, 95%- and 99%-confidence intervals.
- The extreme, i.e. the minimum and maximum values taken by the observer.

The command `profile` is used to define which of these indicators should be calculated. Assume for instance, we want to get the mean, the standard-deviation and the 99% confidence interval in the profile `myProfile`. Commands to do so are as follows.

```
profile set mean myProfile true
profile set standard-deviation myProfile true
profile set confidence-interval-90 myProfile false
profile set confidence-interval-95 myProfile false
profile set confidence-interval-99 myProfile true
profile set extrema myProfile false
```

Quantiles and distributions

It is also possible to calculate quantiles, the distribution and the cumulative distribution of each observer at each mission time.

The user must however that this requires storing a lot of data, which may slow down significantly the process.

For both quantiles and distributions, at most 100 points can be generated.

Commands to set up the calculation of quantiles are as follows.

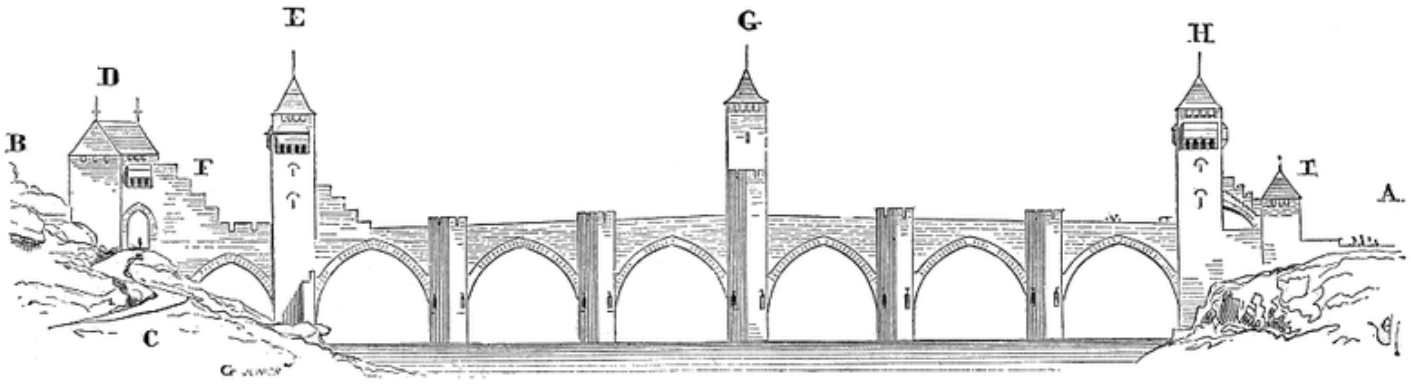
```
profile set quantiles myProfile true
profile set number-of-quantiles 4
```

These two commands can be given in any order.

Commands to set the calculation of distribution and cumulative-distribution are as follows.

```
1 profile set distribution myProfile true
2 profile set cumulative-distribution myProfile true
3 profile set number-of-points 20
```

It is of course possible to calculate the distribution of the observer without calculating its cumulative distribution and vice-versa. Note that in both cases, the number of points is set by the same command and is thus the same for the distribution and the cumulative distribution.



3. Deterministic Models

Key Concepts
- Deterministic Models

This chapter contains a series of exercises aiming at designing deterministic `Janos` models.

3.1 Simple Models

Exercise 3.1 – Quadratic Equation. This little exercise aims at familiarizing you with the syntax of S2ML+DFE and of the `Janos` scripts.

Question 1. Design a class that solves the quadratic equation $ax^2+bx+c=0$. The coefficients a, b and c will be parameters of your class. The class will have three outputs, i.e. it must calculate (at least) the value of three variables: `numberOfSolutions`, `solution1` and `solution2`, with their obvious meaning.

Question 2. Use the class designed in the previous question to calculate the solutions of the following equations:
- $x^2-4x-5=0$
- $-\frac{1}{2}x^2-\frac{11}{3}x-\frac{7}{6}=0$

■

Exercise 3.2 – Stick Game. Alice and Bob play the following game with wooden sticks.

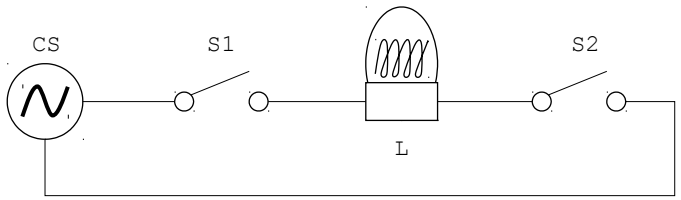
At the beginning of the game, there are twelve sticks on the table. Each player takes in turn 1, 2 or 3 sticks. The player who takes the last stick loses the game. Alice plays first.

Question 1. Enumerate all possible configurations of the game.

Question 2. Use these configurations to design a S2ML+DFE that determines whether Alice has a winning strategy.

3.2 Advanced Models

Exercise 3.3 – Electric Circuit. Consider the simple electric circuit pictured below.



Question 1. Design a S2ML+DFE model for this circuit.

Hint: Use Boolean variables to represent whether the current is circulating into the circuit. You will need two sets of variables: one to represent the circulation from left to right and another one to represent the circulation from from right to left.

Question 2. Assume that switches are automatically open and closed at the following times. Determine with your model when the light is on.

Switch/Hour	0:00	7:00	13:00	17:00	23:00
S1	closed	closed	open	closed	closed
S2	open	closed	closed	closed	open

Exercise 3.4 – Attack Tree. Figure 3.1 shows an attack tree. Attack trees have been introduced by Schneier (Schneier 1999) to model threats against computer systems.

In the attack tree of the figure, one consider two types of attacks: those requiring a special equipment, and those that do not. Of course, if an attack is possible without special equipment, it is *a fortiori* possible with.

Each type of attack has a cost (some attacks are however impossible without a special equipment).

There are three types of nodes in the tree:

- Leaves, which represent basic attacks, may require or not a special equipment and have a cost.
- "Or" internal nodes (default), which represent the different possibilities to perform an attack. They may describe also an cost attached to attacks with and without special equipment.
- "And" internal nodes (pictured with an ellipse), which represent how threats can be combined to perform an attack. They may describe also an cost attached to attacks with and without special equipment.

Question 1. Design a S2ML+DFE to encode the attack tree of Figure 3.1.

Question 2. Use the model to calculate the minimum cost of an attack with and without special equipment.

Exercise 3.5 – Manifold. Figure 3.2 shows a (simplified) manifold, as one can find in oil and gas extraction fields.

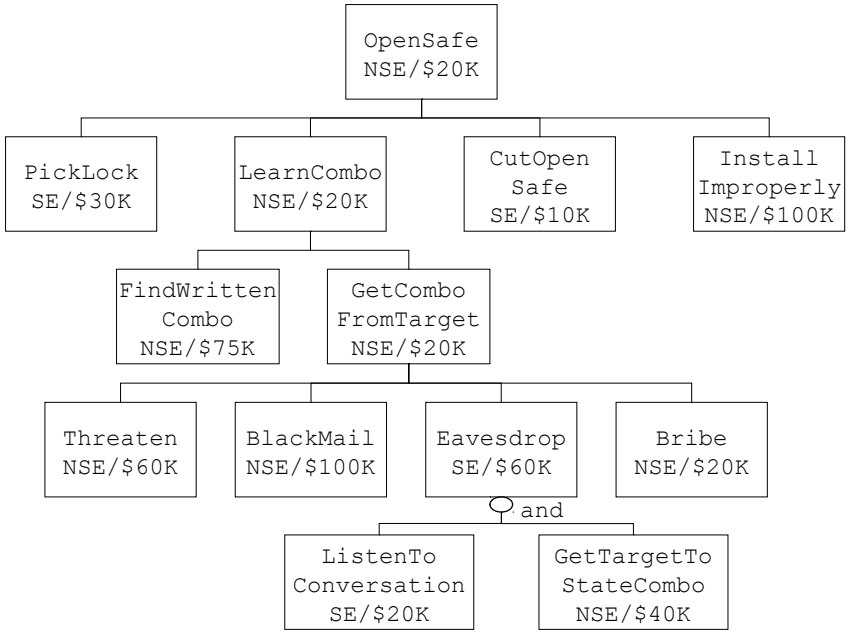


Figure 3.1: An attack tree

Question 1. Design S2ML+DFE classes to model wells and valves. Assume that the flow rate of fluid (mix of oil, gas and water) coming from wells is some real number (represented by a parameter).

Question 2. Using the classes designed in the previous section, design a S2ML+DFE model calculating the flow rate of fluid going from the wells to the production facility and to the test separator, depending on the states of the valves.

Question 3. Test your model using `Janos`, by representing the following 24 hours schedule on the configuration of the manifold. In the following table, P stands for production and T for test. This means that the flow of fluid coming from well 1 is going to the production all the time but from 7:00 to 8:00 where it is diverted to the test separator.

Switch/Hour	0:00	7:00	8:00	15:00	16:00	23:00	24:00
`Well 1`	P	T	P	P	P	P	P
`Well 2`	P	P	P	T	P	P	P
`Well 3`	P	P	P	P	P	T	P

■

Exercise 3.6 – Order Process. The BMPN diagram (White and Miers 2008) pictured Figure 3.3 represents an ordering process in which three are actors involved: `Customer`, `Accoun-ting` and `Shipment`. Rounded rectangles represent tasks of the process. Arrows represent precedences among the tasks (which task follows which other). Circles represent initial and final states. Finally, diamonds with a plus symbol, called gates, represent sub-processes that can be performed in parallel.

Durations (in days) of tasks are given in the following table.

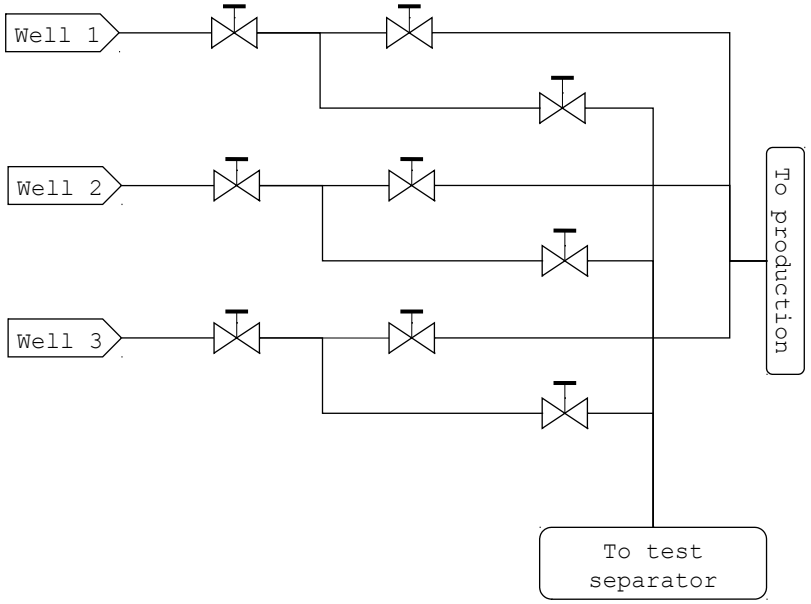


Figure 3.2: A simplified manifold

Task	Duration
`PrepareOrder`	3
`CheckOrder`	2
`ConfirmOrder`	5
`PrepareShipment`	4
`AcceptOrder`	1

Tasks have thus a starting and a completion dates (that must be calculated).

Question 1. Design S2ML+DFE classes for states, tasks and gates. Be careful: despite they are graphically represented alike, initial and terminal states have a different semantics. Similarly, fork and merge gates have a different semantics.

Question 2. Use these classes to design a S2ML+DFE model for the whole BPMN diagram. Use your model to calculate the total duration of the process.

■

Exercise 3.7 – School. An elementary school delivers classes to pupils from first to fifth grades. For each grade, there are four classes. The number of girls and boys in each class is the following table.

Grade/Class	A	B	C	D
1st	11/12	13/12	10/14	13/11
2nd	12/12	11/12	11/13	13/10
3rd	11/12	12/11	11/11	14/10
4th	15/9	11/12	10/13	12/12
5th	13/11	11/13	12/12	10/14

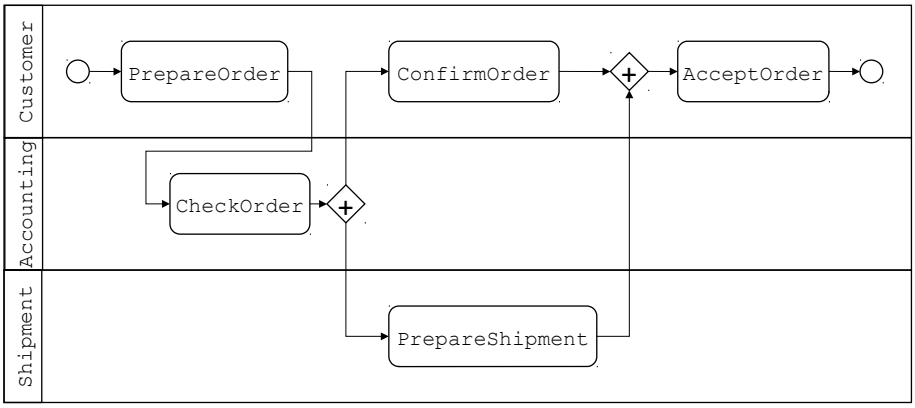


Figure 3.3: The BPMN diagram for the "order" business process

Question 1. Design a S2ML+DFE hierarchical model, using class/instance and prototype/clone mechanisms, to represent the school and to count the total number of girls, boys and pupils.

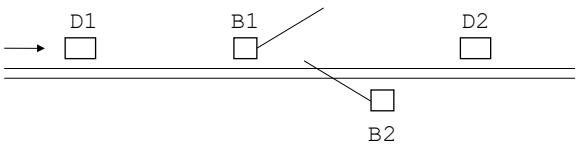
Question 2. The school has a total budget of 5000 euros this year for the music classes. Modify your model so to calculate the budget allocated to each class, given that the budget is allocated *prorata* the number of pupils in the class.

Question 3. Each class has its own main teacher. However, some disciplines are taught by specialized teachers. Alice, Bob and Carol are in charge of music:
- Alice teaches to all classes A, plus 1st and 2nd grades of class D.
- Bob teaches to all classes B, plus 3rd and 4th grades of class D.
- Carol teaches to all classes C, plus 5th grades of class D.

Extends your model to be able to count the numbers of boys, girls and pupils, Alice, Bob and Carol are teaching.
Hint: you can use aggregation.

■

Exercise 3.8 – Railway Crossing. The following diagram represents a railway crossing. The train arrives from the left and goes to the right. When it reaches the detector `D1`, a signal is sent to the barriers `B1` and `B2` which start closing. When it reaches the detector `D2`, another signal is sent to the barriers `B1` and `B2` which start opening.



The barriers take 30 seconds to open and to close. For safety reasons, they must be closed one minute before the train passes. Detectors are located at 1km of the barriers.

Question 1. Design a S2ML+DFE model to represent the above system. Assuming that the train goes 30km/hour on this portion of tracks, use the model to determine when it reaches the

barriers, the detector `D2` whether safety constraints are obeyed and for how long barriers will be closed.

Question 2. Determine experimentally (by tries and errors) the maximum speed the train can go while obeying safety constraints.

■

Exercise 3.9 – Chest Clinic. The diagram pictured Figure 3.4 is an example of Bayesian network taken from (Lauritzen and Spiegelhalter 1988). It is a simplified version of a network that could be used to diagnose patients arriving at a clinic.

Each node in the network corresponds to some condition of the patient, for example, `VisitToAsia` indicates whether the patient recently visited Asia. Each node has thus a certain probability of occurrence. This probability is given directly for source nodes, e.g. `0.01` for `VisitToAsia`. Otherwise, it is calculated from the in-coming edges.

The edges between any two nodes indicate that there are probability relationships that are known to exist between the states of those two nodes. Thus, smoking increases the chances of getting lung cancer and of getting bronchitis. Both lung cancer and bronchitis increase the chances of getting dyspnea (shortness of breath). Both lung cancer and tuberculosis, but not usually bronchitis, can cause an abnormal lung x-ray. And so on. Each edge from the node s to the node t comes with the two conditional probabilities $p(t|\text{not } s)$ and $p(t|s)$, in this order.

For nodes t with only one in-coming edge $s \to t$, the calculation of the probability of the node is thus easy. It is simply $p(t) = p_0 + p_1$, where $p_0 = p(t|\text{not } s) \times (1 - p(s))$ and $p_1 = p(t|s) \times p(s)$.

For nodes t with two in-coming edges $s_1 \to t$ and $s_2 \to t$, the calculation of the probability of the node goes as follows: $p(t) = p_{10} \times p_{20} + p_{10} \times p_{21} + p_{11} \times p_{20} + p_{11} \times p_{21}$, where $p_{10} = p(t|\text{not } s_1) \times (1 - p(s_1))$, $p_{11} = p(t|s_1) \times p(s_1)$ $p_{20} = p(t|\text{not } s_2) \times (1 - p(s_2))$, and $p_{21} = p(t|s_2) \times p(s_2)$.

The objective of the exercise is to create a library of S2ML+DFE classes to implement such networks and then to use these classes to model the above network.

Question 1. Create a class for edges. The two conditional probabilities $p(t|\text{not } s)$ and $p(t|s)$ are encoded as parameters. Given a input probability $p(s)$, the class updates the two probabilities $p_0 = p(t|\text{not } s) \times (1 - p(s))$ and $p_1 = p(t|s) \times p(s)$.

Question 2. Create a class for each type of nodes: source nodes, node with one parent and nodes with two parents. Theses class must maintain the probability of the node.

Question 3. Use classes defined in the previous question to design a S2ML+DFE model of the "Chest Clinic" network.

■

Exercise 3.10 – 3-out-of-5. Assume we want to calculate the probability of failure of a 3-out-of-5 system, i.e. a system that works if at least 3 out of its 5 components work, or reciprocally that is failed if at least 3 out of its 5 components are. We shall name components `A`, `B`, `C`, `D` and `E`.

Assume moreover that the five components fail at random and independently according to a negative exponential distribution of parameter $\lambda = 1.00 \times 10^{-4}$, i.e. that the probability $p_c(t)$ that the component c is failed at time t is defined as follows

$$p_c(t) \stackrel{def}{=} 1 - e^{-\lambda \times t}$$

To calculate the probability of failure of the system as a whole, we shall create the binary

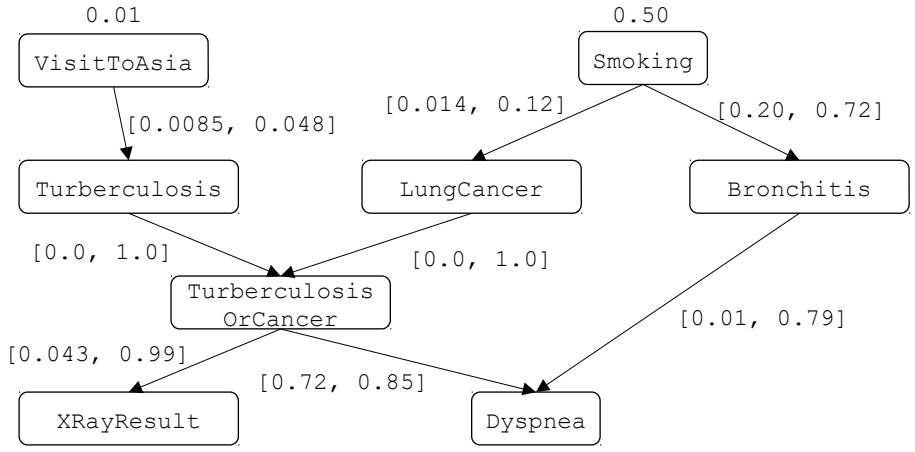


Figure 3.4: The "Asia" or "Chest Clinic" Bayesian network

decision diagram pictured Figure 3.5. This diagram encodes the state of the system according to the state of its components. It reads top-down and is made of two types of nodes:

– Leaves 0 and 1, represented by squares.
– Internal nodes, represented by circles.

Each internal node is labeled with a component has two out-edges: a then out-edge, represented by a plain line, and a else out-edge, represented by a dashed line. The then out-edge represents the case where the component is failed, while the else out-edge represents the case where the component is not failed, i.e. is working.

The leaves 0 and 1 represent thus respectively the cases where the system is working and is failed.

The probability of each node is calculated bottom-up:

– The probabilities of leaves 0 and 1 are respectively 0.0 and 1.0.
– The probability of an internal node n labeled with the component c and whose then and else out-edges point respectively to nodes t and e is calculated via the Shannon decomposition:

$$p_n(t) \quad = p_c(t) \times p_t(t) + (1.0 - p_c(t)) \times p_e(t)$$

The objective of the exercise is to design a S2ML+DFE model to encode the diagram of Figure 3.5 and to calculate, from this diagram, the probability of failure of the system at different mission-times.

Question 1. Design a class to represent components (use the built-in probability distribution `exponentialDistribution`).

Question 2. Design a class to represent generic nodes, then derive from this class one class to represent leaves and another one to represent internal nodes.

Question 3. Encode the diagram of Figure 3.5 into a main block.

Question 4. Using the diagram, calculate the probability of failure of the system at times 0, 1000, 5000 and 10000.

■

Exercise 3.11 – Oil Production Plant. Figure 3.6 shows a oil production plant that consists in seven units. Gas separated from the well fluid at upstream side is fed to the facility, treated through

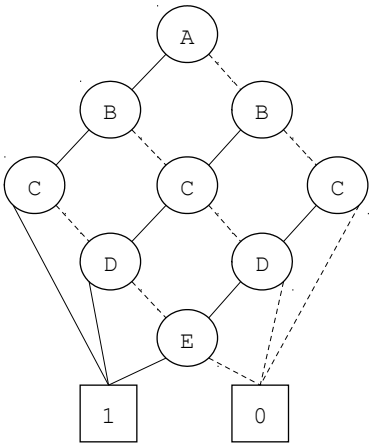


Figure 3.5: The binary decision diagram encoding the state of a 3-out-of-5 system

high pressure separators (`HPS-A`, `HPS-B` and `HPS-C`) and dehydrators (`DEH-A` and `DEH-B`). It is then led to compressors (`CMP-A` and `CMP-B`). The maximum production capacity for each unit is shown on the figure.

Each unit may fail and be repaired. Failures are randomly distributed according to exponential distributions. Failure rates for high pressure separators, dehydrators and compressors are respectively 8.91×10^{-5}, 3.11×10^{-5} and 3.50×10^{-5} per hour. Repair times are assumed to be also randomly distributed according to exponential distributions. Repair rates for high pressure separators, dehydrators and compressors are respectively 2.54×10^{-3}, 3.95×10^{-3} and 5.14×10^{-3} per hour.

The probability that a unit is failed at time t is thus described by the following equation.

$$Q(t) \stackrel{def}{=} \frac{\lambda}{\lambda+\mu} \times \left(1.0 - e^{-(\lambda+\mu)\times t}\right)$$

where λ denotes the failure rate and μ denotes the repair rate of the unit.

Depending on the state of each production unit, the plant can produce at a certain level.

The expected production capacity at time t is thus as follows.

$$EP(t) \stackrel{def}{=} \sum s \in SP_s \times Q_s(t)$$

where S stands for the set of global states, P_s stands for the production of the plant in the state s and $Q_s(t)$ stands for the probability that the plant is the state s at time t.

Question 1. Design a generic class to represent units (use the built-in probability distribution `GLMDistribution`). Design a derived class for each type of unit. The derived class should calculate the probability for the unit to produce at its maximum capacity and at the capacity 0 (because it is failed).

Question 2. Design a S2ML+DFE model to calculate the expected production of the plant. This model should be made of three stations in series: the separation, the dehydration and the compression station. For each station, you have to determine the possible production levels of the station and to calculate the probability that the station is in each of these levels. Same thing at plant level.

Question 3. Using the model, calculate the expected production of the plant at times 0, 1000, 5000 and 10000.

■

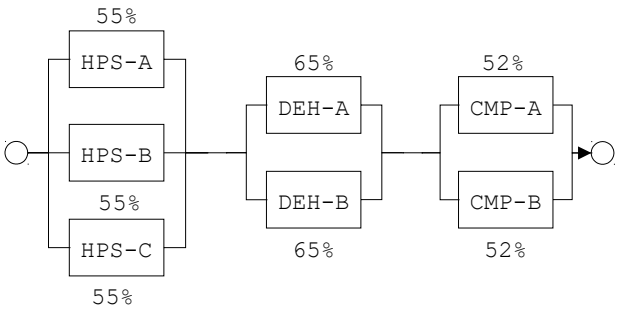
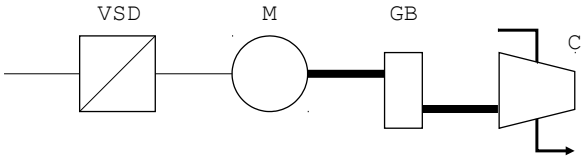


Figure 3.6: A (very simplified) oil production plant

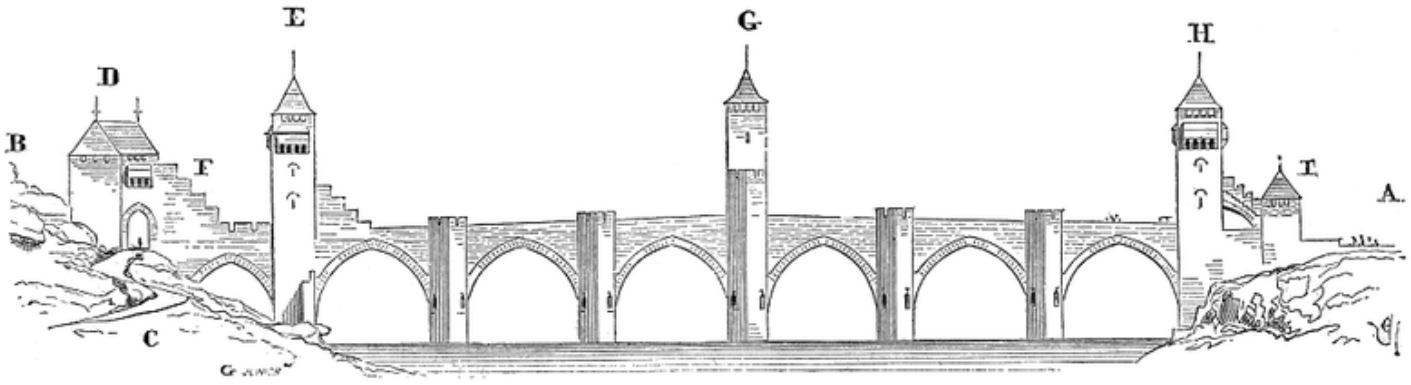
Exercise 3.12 – Compressor Drive. A compressor drive system, located in Norway, is in charge of compressing the gas extracted in north sea to send it to the consumers in France and Great Britain. This huge machine is made of 6 compressor trains working in parallel. Each compressor train is itself made of a variable speed drive `VSD`, a motor `M`, a gear box `GB` and a compressor `C` working in series, as illustrated below.



During winter (6 months), there is an important need of gas, therefore the six trains are working actually in parallel. During summer (6 other months), only 3 trains are needed. The 3 other trains are stopped, and one is maintained. The maintenance policy works according to the round robin principle: On month 1, trains 1, 2 and 3 are stopped and train 1 is maintained. On month 2, trains 2, 3 and 4 are stopped and train 2 is maintained. And so on.

Question 1. Design a S2ML+DFE model for this system that represents which unit is working and which is not, month by month, over a 5 years period.
Hint: It is a good idea to create a controller that sends the status – working, stopped or maintained – to each train. Trains are sending in turn this status to its components. Observers are created on each component to observe the status.

■



4. Stochastic Models

Key Concepts
- Stochastic Models

This chapter contains a series of exercises aiming at designing stochastic `Janos` models.

4.1 Simple Models

Exercise 4.1 – Assessment of the value of π. The objective of this exercise is to assess the value of π using `Janos`.

Question 1. Design a `Janos` model to assess the value of π according to the principle described in example **??** and perform a Monte-Carlo simulation for different seeds of the random number generator and different numbers of with 10,000 and 100,000 executions. Calculate the distribution for 10 points.

The model we just designed is rather naive. There is room to improve it. For instance, we can split intervals $[0,1]$ into 2 parts: $[0,0.5]$ and $[0.5,1]$. Now, we can look independently the four possible cases:

- $x \in [0,0.5]$ and $y \in [0,0.5]$,
- $x \in [0,0.5]$ and $y \in [0.5,1]$,
- $x \in [0.5,1]$ and $y \in [0,0.5]$,
- $x \in [0.5,1]$ and $y \in [0,0.5]$.

Question 1. Design a new model according to the above suggestion. Perform a Monte-Carlo simulation for different seeds of the random number generator and different numbers of tries. Compare the results and the computation time with those of the naive implementation. Perform a Monte-Carlo simulation with 10,000 and 100,000 executions. Calculate the distribution for 10 points.

■

Exercise 4.2 – Cold Redundancy. Consider a system made of two components A and B in cold redundancy, i.e. B is started when A fails. Assume that both components are subject to random failures and that these failures are distributed according to Weibull laws of scale parameter $\alpha = 2.0 \times 10^4$ and shape parameter $\beta = 3$, i.e. that the probability $R(t)$ that the component is still working at time t is given by the formula:

$$R(t) \quad = \quad \exp\left(-\frac{t}{\alpha}\right)^{\beta}$$

To assess the reliability of this system, we can calculate an analytical solution and then apply this analytical solution and the mission-time t of interest. This works fine for small examples, but is infeasible when the system is complex.

An alternative solution consists in drawing at random failure dates for components A and B, summing these failure dates and comparing the result with the mission time. By repeating this experiment sufficiently many times, we get an estimate of the reliability of the system.

Question 1. Design a S2ML+DFE model according to the above specifications.

Question 2. Calculate the probability of failure of the system at time 0, 730 (1 month), 1460 (2 months), ... 8760 (1 year).

■

Exercise 4.3 – Assessment of Integrals. Assume we want to assess the value of the following integral.

$$F \quad = \quad \int_0^{\pi/6} \frac{\cos(x)}{\cos(x) - \sin(x)}\, dx$$

For bright students in calculus, this is not a problem. After a few attempts of applying classical recipes, they will get the analytical result:

$$F \quad = \quad -\frac{1}{2}\ln\left(\frac{\sqrt{3}-1}{2}\right) + \frac{\pi}{12} \approx 0.7643$$

A piece of cake.

For the others, which includes the author, this may turn to be a little more tricky.

There is a solution however: just perform a Monte-Carlo simulation. The idea is to draw uniformly at random sufficiently many numbers between 0 and $\pi/6$, to calculate for each of these numbers x, the value of $\frac{\cos(x)}{\cos(x)-\sin(x)}$, then to take the mean over the calculated values.

Question 1. Design a S2ML+DFE model according to the above specifications. Use this model to assess the value of F.

Question 2. Same question with the following integrals.

$$\begin{aligned} X_1 &= \int_{-1}^{1} 2x^2+3x+1\ dx \\ X_2 &= \int_{0}^{\pi/2} \sin x\ dx \\ X_3 &= \int_{3}^{5}\int_{1}^{2} 2x+5y^2\ dy\ dx \\ X_4 &= \int_{0}^{1}\int_{0}^{1-x} x+y\ dy\ dx \end{aligned}$$

■

Exercise 4.4 – Chi Square. The following distribution, where the x_i's are normally distributed random variables with mean 0 and standard-deviation 1 are of special interest.

$$f(x_1,x_2,x_3,x_4) = \sqrt{x_1^2+x_2^2+x_3^2+x_4^2}$$

Question 1. Design a S2ML+DFE model to study the distribution described above. We are in particular interested to determine the probability that this sum exceeds 2.5.

■

Exercise 4.5 – Degrading Unit. Consider a unit that goes through 3 successive periods during its life-time: infancy, regular operation and wear out. The mission time of the unit is 43800 hours (5 years).

The infancy period lasts 1% of the mission time. The wear-out period lasts the last 10% of the mission-time. The regular operation period lasts the 89% in between the infancy period and the wear-out period.

The probability of failure of the unit in each period is assumed to be exponentially distributed, with a failure rate that depends on the period. Moreover, failure rates are not know precisely. They are assumed to be uniformly distributed:

Period	Low rate	High rate
Infancy	1.00×10^{-4}	3.00×10^{-4}
Regular operation	1.00×10^{-7}	3.00×10^{-7}
Wear out	1.00×10^{-5}	3.00×10^{-5}

Question 1. Design a S2ML+DFE model and perform a Monte-Carlo simulation to study the reliability of this unit.

■

Exercise 4.6 – Degradation Process. The state automaton pictured Figure 4.1 represents the degradation process of a component.

Initially the component is working (state `W`). It may degrade (transitions `d`) and go in a first degraded state (`D1`). Then it may degrade again and go in state (`D1`). A third degradation takes it to the failed state (`F`).

In any of the states `F`, `D1` and `D2` the component can also experience a failure (transitions `f`) that takes it directly to the state `F`.

Question 1. Assuming that time to degradation and failure are both exponentially distributed with,

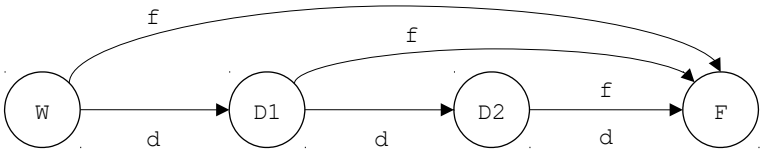


Figure 4.1: State automaton of a degradation process

respectively a degradation rate of 1.00×10^{-4} and a failure rate of 2.00×10^{-5} per hour, design a S2ML+DFE model to estimate the probability that the component is failed after 1000, 5000 and 10000 hours.

■

Exercise 4.7 – Investments. You are proposed an investment plan which consists in investing 20,000 U (where U is your currency unit) at the beginning of each year during the next 10 years. You will be served interests at the end of each year, however the interest rate for each year varies and is known only up to an uncertainty: it is normally distributed with a mean at 2% and a standard deviation of 1.5% (this means that it may be negative).

Question 1. Design a S2ML+DFE model to study what is your final capital at the end of the 10 years period (given that you will not withdraw the money you have invested before the end of the period).

Question 2. Calculate a distribution to have a better view of what can happen.

■

4.2 Advanced Models

Exercise 4.8 – Simple Queue. Let us come back on Example 1.3.

Question 1. Design a S2ML+DFE model according to specifications of the example.

Question 2. Calculate the mean waiting time for 20 clients.

■

Exercise 4.9 – Manchester. Let us come back on Example 1.4.

Question 1. Design a S2ML+DFE model according to specifications of the example.

Question 2. Calculate quartiles as in the example.

■

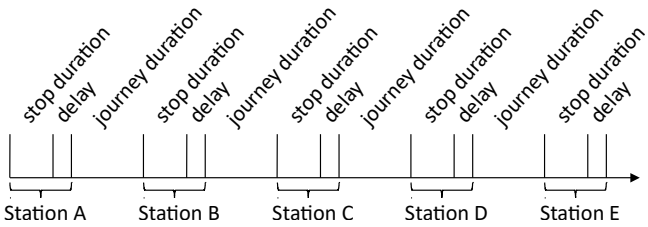
Exercise 4.10 – Metro Line. A metro line goes from station A to station E, through stations B, C and D (in order). The journey from a station to the next one has a certain duration (measured in seconds). Similarly, the stop at each station has a certain duration. However, due to the number of passengers, the metro main be delayed at each station. It has been observed that this delay is uniformly distributed between two bounds. The schedule of a journey can thus be pictured follows.

Table 4.1: Metro line: duration of stops and potential delays and each station

Station	Stop duration	Low delay	high delay
A	30	0	30
B	30	0	40
C	30	0	30
D	30	0	10
E	30	0	5

Table 4.2: Metro line: durations of travels between two consecutive stations

Station	Duration	Delay reduction capacity
A to B	120	10
B to C	90	5
C to D	120	10
D to E	90	5



Question 1. Design a S2ML+DFE model to assess the journey time and the delay when leaving station E (use data provided Tables 4.1 and 4.2). Calculate deciles.

Question 2. Assuming now that the driver can accelerate a bit the journey between two stations so to compensate the delay. Modify your model to take that into account. Calculate deciles.

■

Exercise 4.11 – Driveway. To drive from home to work, Alice has the choice among different itineraries, basically depending when she gets in and out the parkway (see Figure 4.2). The travel time on each portion of the road depends heavily on the traffic, which is itself rather uncertain. Fortunately, some statistics have been made. Using the statistics given Table 4.3, design a S2ML+DFE model to assess Alice's (best) travel time to work. ■

Exercise 4.12 – Fire Protection System. Figure 4.3 shows a fire protection system is installed in a warehouse. It is in charge of extinguishing fire outbreaks. When a fire outbreak (smoke) is detected, water is pumped in a tank and sprayed over the fire via sprinklers.

The water sprinkling is launched by operators. The detection of fire outbreaks is in charge of operators or of another system.

The tank and the sprinklers are assumed to be perfectly reliable. Other components may fail and their failure is assumed to be exponentially distributed, with the failure rates given in the following table.

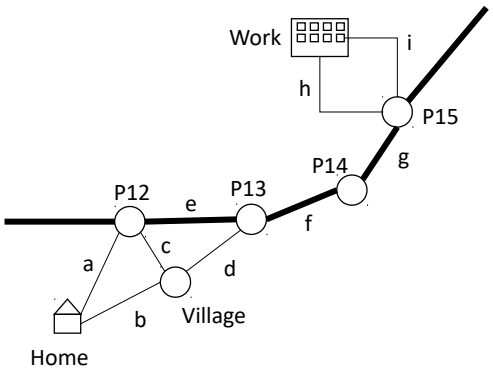


Figure 4.2: Alice's drive way

Table 4.3: DriveWay: durations of travels

Road	Distribution	Low	High	Mode
a	triangular	2	4	3
b	triangular	3	5	4
c	triangular	1	2	1.8
d	triangular	3	6	5
e	uniform	3	5	
f	uniform	4	6	
g	uniform	2	4	
h	triangular	1	3	2.2
i	triangular	2	2.6	2.2

Component	Failure rate (per hour)
Electric Source	1.00×10^{-8}
Pump 1	1.00×10^{-5}
Pump 2	1.00×10^{-5}
Valve	2.00×10^{-8}

Question 1. Design a fault tree and implement it as a S2ML+DFE model to assess the reliability of this system.

Question 2. Design a reliability block diagram and implement it as a S2ML+DFE model to assess the reliability of this system.

■

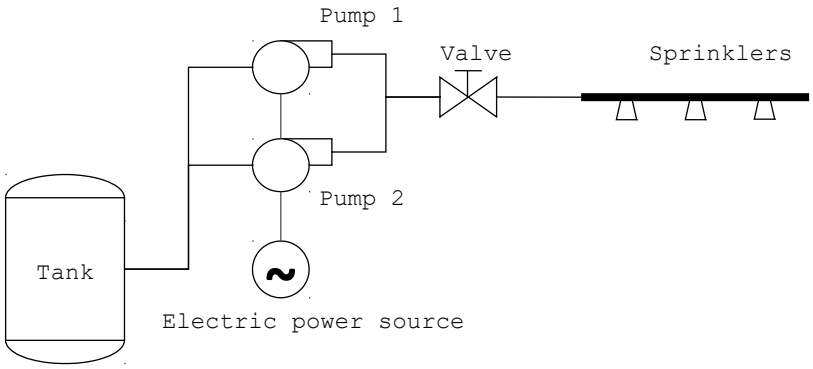
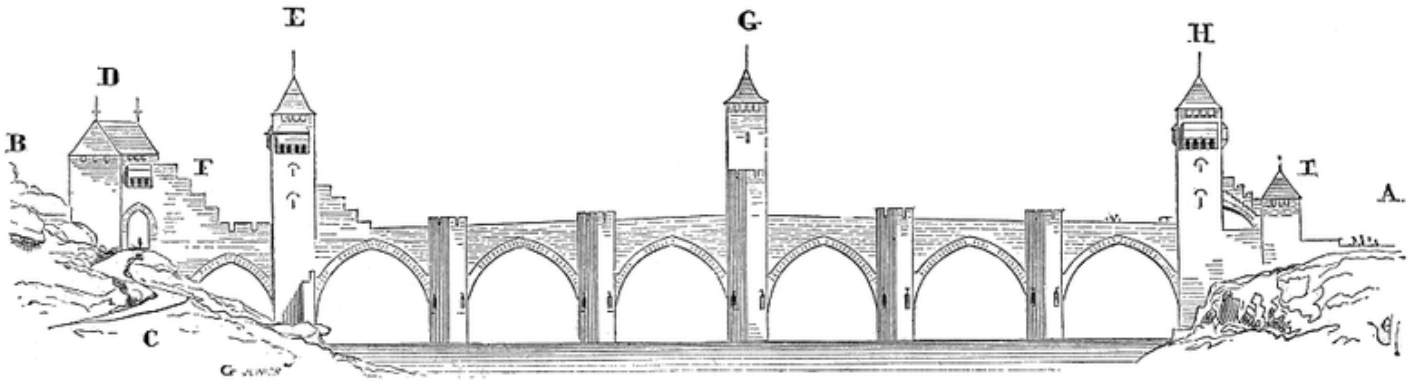


Figure 4.3: A fire protection system

Bibliography



Bibliography

Batteux, Michel, Tatiana Prosvirnova, and Antoine Rauzy (Oct. 2018). "From Models of Structures to Structures of Models". In: *IEEE International Symposium on Systems Engineering (ISSE 2018)*. Best paper award. Roma, Italy: IEEE. DOI: 10.1109/SysEng.2018.8544424 (cited on page 1).

— (2019). "AltaRica 3.0 in 10 Modeling Patterns". In: *International Journal of Critical Computer-Based Systems* 9.1–2, pages 133–165. DOI: 10.1504/IJCCBS.2019.098809 (cited on page 1).

Box, George E. P. and Mervin E. Muller (1958). "A Note on the Generation of Random Normal Deviates". In: *The Annals of Mathematical Statistics* 29.1, pages 610–611. DOI: 10.1214/aoms/1177706645 (cited on page 68).

Chaitin, Gregory (2001). *Exploring Randomness*. London, England: Springer-Verlag. ISBN: 1-85233-417-7 (cited on page 15).

Dubi, Arie (2000). *Monte Carlo Applications in Systems Engineering*. Chichester, West Sussex, England: Wiley. ISBN: 978-0471981725 (cited on page 15).

Fritzson, Peter (2015). *Principles of Object-Oriented Modeling and Simulation with Modelica 3.3: A Cyber-Physical Approach*. Hoboken, NJ 07030-5774, USA: Wiley-IEEE Press. ISBN: 978-1118859124 (cited on page 1).

Kaplan, Edward Lynn and Paul Meier (1958). "Nonparametric estimation from incomplete observations". In: *Journal of American Statistics Association* 53.282, pages 457–481. DOI: 10.2307/2281868 (cited on page 61).

Kolmogorov, Andrei N. (1933). *Grundbegriffe der Wahrscheinlichkeitsrechnung*. Berlin, Germany: Springer. ISBN: 978-3540061106 (cited on page 51).

Lauritzen, Steffen L. and David J. Spiegelhalter (1988). "Local computations with probabilities on graphical structures and their application to expert systems". In: *Journal Royal Statistics Society B* 50.2, pages 157–194 (cited on page 32).

Matsumoto, Makoto and Takuji Nishimura (1998). "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator". In: *ACM Transactions on Modeling and Computer Simulation* 8.1, pages 3–30. DOI: 10.1145/272991.272995 (cited on page 68).

Metropolis, Nicholas Constantine (1987). “The beginning of the Monte Carlo method”. In: *Los Alamos Science* 15 (Special issue in memory of Stanislaw Ulam 1909-1984), pages 125–130 (cited on page 5).

Rauzy, Antoine (2020). *Performance Engineering in Python*. Les Essarts le Roi, France: AltaRica Association. ISBN: 978-82-692273-1-4 (cited on page 2).

— (2022). *Model-Based Reliability Engineering – An Introduction from First Principles*. Les Essarts le Roi, France: AltaRica Association. ISBN: 978-82-692273-2-1 (cited on pages 3, 17, 19).

Rauzy, Antoine and Cecilia Haskins (2019). “Foundations for Model-Based Systems Engineering and Model-Based Safety Assessment”. In: *Journal of Systems Engineering* 22, pages 146–155. DOI: 10.1002/sys.21469 (cited on page 1).

Rossum, Guido van (May 1995). *Python tutorial*. Technical report CS-R9526. Amsterdam, The Neederland: Centrum voor Wiskunde en Informatica (CWI) (cited on page 2).

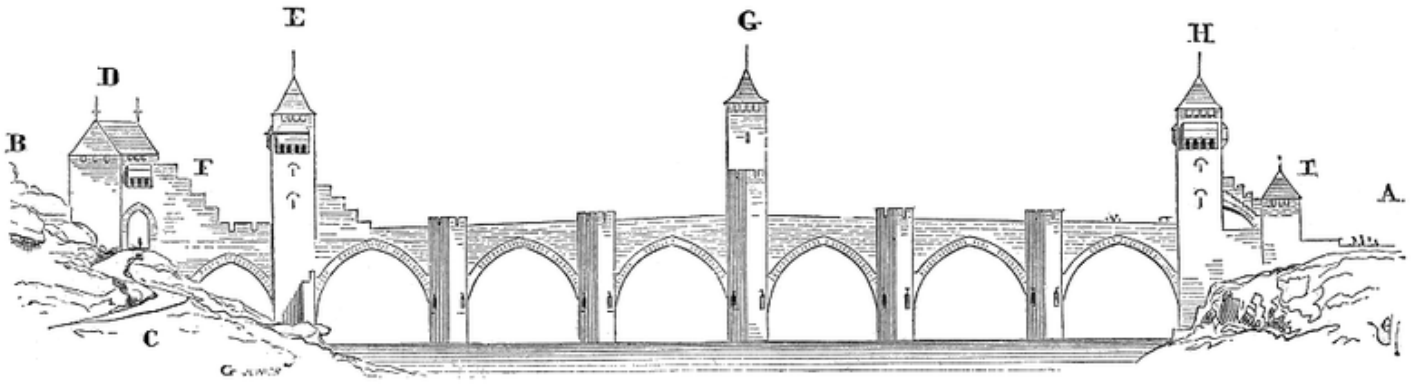
Rubino, Gerardo and Bruno Tuffin (2009). *Rare Event Simulation using Monte Carlo Methods*. Hoboken, NJ, USA: John Wiley and Sons. ISBN: 9780470772690 (cited on page 15).

Schneier, Bruce (1999). “Attack Trees”. In: *Dr Dobb’s Journal* 24.12 (cited on page 28).

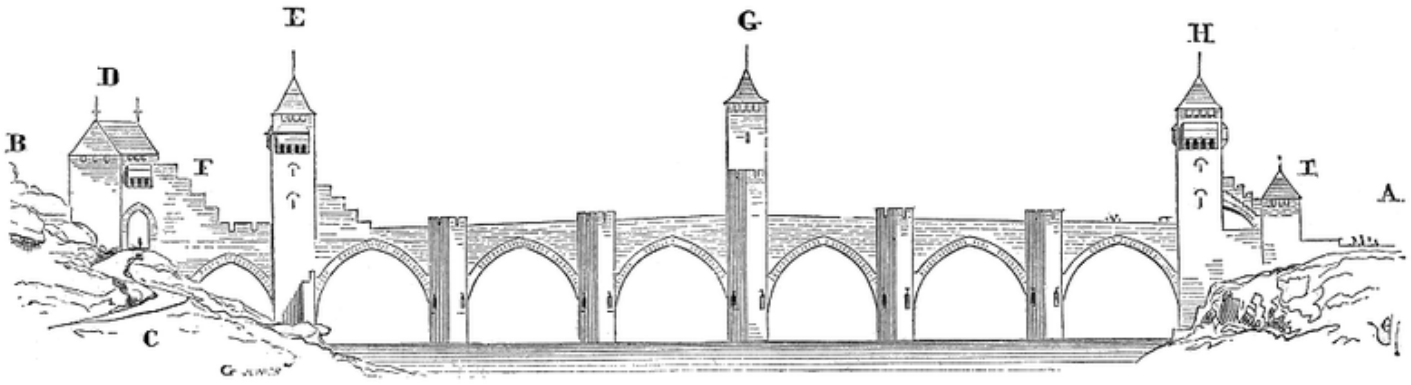
Welford, B. P. (Aug. 1962). “Note on a method for calculating corrected sums of squares and products”. In: *Technometrics* 4.3, pages 419–420. DOI: 10.2307/1266577 (cited on page 64).

White, Stephen and Derek Miers (2008). *BPMN Modeling and Reference Guide: Understanding and Using BPMN*. Lighthouse Point, FL, USA: Future Strategies Inc. ISBN: 978-0977752720 (cited on page 29).

Zio, Enrico (2013). *The Monte Carlo Simulation Method for System Reliability and Risk Analysis*. Springer Series in Reliability Engineering. London, England: Springer London. ISBN: 978-1-4471-4587-5 (cited on page 15).



Index



A. Probability Theory and Statistics

Key Concepts

- Axiomatic of probability theory
- Sample space, event, σ-algebra
- Probability measure
- Sylvester-Poincaré development
- Conditional probabilities, Bayes's theorem
- Random variable, cumulative distribution function
- Expected value
- Mean, variance, standard deviation
- Laws of large numbers, central limit theorem
- Important probability distributions
- Quantiles
- Random number generators

Part of model-based systems engineering relies on probability theory. This appendix recalls the main definitions and results of probability theory, gives some important probability distributions and recalls basic statistics.

A.1 Probability Theory

A.1.1 Axiomatic

As a branch of mathematics, modern probability theory is defined by means of an axiomatic. The axiomatic of probability theory has been proposed by Kolmogorov (Kolmogorov 1933). It has the advantage to cover both the discrete and the continuous cases.

Definition A.1.1 – Sample space. A *sample space* is the set of all the possible outcomes of a non-deterministic experiment. An experiment is *deterministic* if it gives always the same result in the same condition and *non-deterministic* otherwise.

In probability theory, the sample space is usualy called Ω.

Definition A.1.2 – Event. An *event* over a sample space Ω is a subset of Ω. An event gather all possible outcomes of the experience that fullfil a certain property.

For the probability theory to be well defined, the set $\mathscr{A}$ of events should have a particular mathematical structure, namely it should be a Borel σ-algebra, or σ-algebra for short.

Definition A.1.3 – σ-algebra. Let Ω be a set. A σ-*algebra* over Ω is a set $\mathscr{A} \subseteq 2^{\Omega}$, where 2^{Ω} denotes the *power set*, i.e. the set of subsets of $\mathscr{A}$, such that:

$\mathscr{A} \neq \emptyset$	$\mathscr{A}$ is not empty.
$\forall A \in \mathscr{A}, \Omega \setminus A \in \mathscr{A}$	$\mathscr{A}$ is closed by complementation.
If $\forall n \in \mathbb{N}$, $B_n \in \mathscr{A}$, then $\bigcup_{n\in\mathbb{N}} B_n \in \mathscr{A}$	$\mathscr{A}$ is closed by countable union.

The above definition implies that:
– The empty event $\emptyset$ and the total event Ω belong to $\mathscr{A}$.
– $\mathscr{A}$ is closed by countable intersection.
The pair $(\Omega, \mathscr{A})$ is a probabilisable space, i.e. it is possible to define a probability measure on it.

Definition A.1.4 – Probability measure. Let Ω be a set and $\mathscr{A}$ be a σ-algebra over Ω. A *probability measure* p is a function from $\mathscr{A}$ to the real interval $[0,1]$ such that:

1. $\forall A \in \mathscr{A}$, $0 \leq p(A) \leq 1$ (*positivity*).
2. $p(\Omega) = 1$ (*unitary mass*).
3. If $\forall n \in \mathbb{N}$, $A_n \in \mathscr{A}$ and if moreover $\forall i, j \in \mathbb{N}$, $i \neq j$, $A_i \cap A_j = \emptyset$, then $p\left(\bigcup_{n\in\mathbb{N}} A_n\right) = \sum_{n\in\mathbb{N}} p(A_n)$ (*additivity*).

A.1.2 Additional Definitions and Properties

The above definition induces a number of well-known properties.

Proposition A.1 – Basic properties of probability measures. Let p be a probability measure over the space $(\Omega, \mathscr{A})$. Then, the following equalities hold for all $A, B \in \mathscr{A}$.

$$\begin{aligned} p(\emptyset) &= 0 \\ p(\Omega \setminus A) &= 1 - p(A) \\ p(A \cup B) &= p(A) + p(B) - p(A \cap B) \end{aligned}$$

The above third equality is extensible to finite unions of events via the so-called Sylvester-Poincaré development.

Proposition A.2 – Sylvester-Poincaré development. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A_1, A_2, \ldots A_n \in \mathscr{A}$. Then, the following equality holds.

$$p\left(\bigcup_{i=1}^{n} A_i\right) = \sum_{k=1}^{n} \left((-1)^{k-1} \sum_{1 \leq i_1 < i_2 < \ldots < i_n \leq n} p(A_{i_1} \cap \ldots \cap A_{i_k}) \right)$$

The notion of independence plays an important role in probabilistic risk analysis. It is defined as follows.

Definition A.1.5 – Independent events. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A, B \in \mathscr{A}$. Then, A and B are *independent* if $p(A \cap B) = p(A) \times p(B)$.

The notion of conditional probability captures the idea of measuring the probability of occur-

rence of an event, given that another event occurred

Definition A.1.6 – Conditional probability. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A, B \in \mathscr{A}$ such that $p(B) \neq 0$. The *conditional probability* of A given B, denoted as $p(A \mid B)$, is defined as follows.

$$p(A \mid B) \stackrel{def}{=} \frac{p(A \cap B)}{p(B)}$$

It follows immediately from the definitions that if A and B are independent events, the following equality holds.

$$p(A \mid B) = p(A) \tag{A.1}$$

We shall conclude this section by the very useful Bayes's theorem, also called theorem about the probability of causes.

Theorem A.3 – Bayes's theorem. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let $A, B \in \mathscr{A}$. Then the following equality holds.

$$p(A \mid B) = \frac{p(B \mid A) \times p(A)}{p(B)}$$

A.1.3 Random Variables

Often, some numerical value calculated from the result of a non-deterministic experiment is more interesting than the experiment itself. These numerical values are called random variables. We shall introduce here only real-valued random variables, but the notion of random variables applies to any measurable set.

Definition A.1.7 – Random variable. Let p be a probability measure over the space $(\Omega, \mathscr{A})$. A (real-valued) *random variable* X is a function from Ω into $\mathbb{R}$ such that:

$$\forall r \in \mathbb{R}, \{\omega \in \Omega : X(\omega) \leq r\} \in \mathscr{A}$$

The cumulative distribution function of a random variable is the probability that this random variable takes a value less or equal to a certain threshold.

Definition A.1.8 – Cumulative distribution function. Let p be a probability measure over the space $(\Omega, \mathscr{A})$ and let X be a random variable built over p. The *cumulative distribution function* of X is the function F_X from $\mathbb{R}$ into $[0,1]$ defined as follows (for all $r \in \mathbb{R}$).

$$F_X(r) \stackrel{def}{=} p(\{\omega \in \Omega : X(\omega) \leq r\})$$

Intuitively, the *expected value* a random variable X, also called the *expectation* of X, is the long-run average value of repetitions of the same experiment X represents.

In the finite or denumerable case, this is formalized as follows.

Definition A.1.9 – Expected value (finite or denumerable case). Let X be a random variable with a countable set of finite outcomes x_1, x_2, ..., occurring with probabilities p_1, p_2, ..., respectively, such that the infinite sum $\sum_{i=1}^{\infty} |x_i| p_i$ converges. The *expected value* of X, denoted

$E[X]$, is defined as following series.

$$E(X) \stackrel{def}{=} \sum_{i=1}^{\infty} x_i \times p_i$$

In the infinite uncoutable case, the formal definition is as follows.

Definition A.1.10 – Expected value (uncountable case). Let X be a random variable whose cumulative distribution function admits a density $f(x)$, then the *expected value* of X is defined as the following Lebesgue integral.

$$E(X) \stackrel{def}{=} \int_{\mathbb{R}} x f(x)\, dx$$

Here follows a basic property of expected value.

Proposition A.4 – Basic property of expected value. Let X and Y be two random variables and $c \in \mathbb{R}$ be a constant. Then,

$$\begin{aligned} E[X+Y] &= E[X]+E[Y] \\ E[cX] &= cE[X] \end{aligned}$$

It is often of interest to know how closely a distribution is packed around its expected value. The variance provides such a measure.

Definition A.1.11 – Variance. Let X be a random variable. The *variance* of X, denoted $\mathrm{Var}(X)$, is the expectation of the squared deviation of X from its expected value.

$$\mathrm{Var}(X) \stackrel{def}{=} E\left[(X - E[X])^2\right]$$

The standard deviation has a more direct interpretation than the variance because it is in the same units as the random variable. It is defined as follows.

Definition A.1.12 – Standard deviation. Let X be a random variable. The *standard-deviation* of a random variable X, denoted $\sigma(X)$, if the square root of its variance.

$$\sigma(X) \stackrel{def}{=} \sqrt{\mathrm{Var}(X)}$$

A.1.4 Laws of Large Numbers and Theorem Central Limit

Laws of large numbers

The frequentist interpretation of probability states that if an experiment is repeated a large number of times under the same conditions and independently, then the relative frequency with which an event E occurs and the probability of that event E should be approximately the same.

A mathematical formulation of this interpretation is the law of large numbers, which exists under two forms: the weak law and the strong law.

The weak law of large numbers states that the sample average converges in probability towards the expected value.

Theorem A.5 – Weak law of large numbers. Let $X_1, X_2 \ldots$ a denumerable family of random variables identically distributed with expected value $E(X_1) = E(X_2) = \ldots = \mu$. Let $\overline{X}_n$ be the

average of the sample made of the n first variables, i.e.

$$\overline{X}_n \stackrel{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$

Then, for any positive number ε,

$$\lim_{n\to\infty} Pr\left(\left|\overline{X}_n - \mu\right| > \varepsilon\right) = 0$$

The strong law of large numbers states that the sample average converges almost surely to the expected value.

Theorem A.6 – Strong law of large numbers. Let $X_1, X_2 \ldots$ a denumerable family of random variables identically distributed with expected value $E(X_1) = E(X_2) = \ldots = \mu$. Let $\overline{X}_n$ be the average of the sample made of the n first variables, i.e.

$$\overline{X}_n \stackrel{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$

Then,

$$\lim_{n\to\infty} Pr\left(\overline{X}_n = \mu\right) = 1$$

The weak law states that for a specified large n, the average of the sample $\overline{X}_n$ is likely to be near μ. Thus, it leaves open the possibility that $\left|\overline{X}_n - \mu\right| > \varepsilon$ happens an infinite number of times, although at infrequent intervals.

The strong law shows that this almost surely will not occur. In particular, it implies that with probability 1, for any $\varepsilon > 0$, there exists a n_0 such that for all $n > n_0$, $\left|\overline{X}_n - \mu\right| < \varepsilon$ holds.

There are special cases, for which the weak law is verified but not the strong one.

Theorem central limit

The central limit theorem states that, in some situations, when independent random variables are added, their properly normalized sum tends toward a normal distribution even if the original variables themselves are not normally distributed. This theorem plays a central role in probability theory because it implies that probabilistic and statistical methods that work for normal distributions can be applicable to many problems involving other types of distributions.

Theorem A.7 – Theorem Central Limit. Let $X_1, \ldots, X_n$ be independent random variables having a common distribution with expectation μ and variance σ^2. Let $\overline{X}_n$ and Z_n defined as follows.

$$\overline{X}_n \stackrel{def}{=} \frac{1}{n}(X_1 + \ldots + X_n)$$
$$Z_n \stackrel{def}{=} \frac{\overline{X}_n - \mu}{n/\sqrt{n}}$$

Let $\Phi(z)$ be the distribution function of the normal law $\mathcal{N}(0,1)$, i.e. the normal law of mean 0 and variance 1. Then for all $z \in \mathbb{R}$,

$$\lim_{n\to\infty} Pr(Z_n \leq z) = \Phi(z)$$

Table A.1: Characteristics of the uniform distribution

Probability density function	$f(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{1}{b-a} & \text{if } a \leq x \leq b \\ 0 & \text{if } x > b \end{cases}$
Cumulative probability function	$F(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{x-a}{x-b} & \text{if } a \leq x \leq b \\ 1 & \text{if } x > b \end{cases}$
Mean	$\frac{1}{2}(b-a)$
Median	$\frac{1}{2}(b-a)$
Variance	$\frac{1}{12}(b-a)^2$

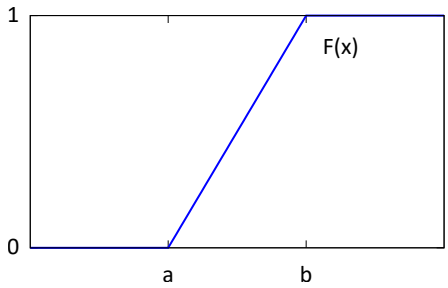


Figure A.1: Cumulative distribution function of the uniform distribution

A.2 Some Important Probability Distributions

Many modeling techniques (beyond systems engineering) require the association of probability distributions with parameters of the model. In practice, there are two ways of defining these probability distributions:

- The first one consists in using parametric distributions.
- The second one consists in using so-called empirical distributions, i.e. distributions defined by a set of points between which the value of the function is interpolated.

We shall review them in turn.

A.2.1 Parametric distributions

The following list of parametric distributions gathers only those that are frequently used in the framework of model-based systems engineering. There exists indeed many others, used in different contexts.

Uniform distribution

The *continuous uniform distribution*, or simply *uniform distribution*, is such that for each member of the family, all intervals of the same length on the distribution's support are equally probable. The support is defined by the two parameters, a and b, $a \leq b$, which are its minimum and maximum values.

Table A.1 gives the characteristics of the uniform distribution.

Figure A.1 shows the shape the cumulative distribution function of the uniform distribution.

Table A.2: Characteristics of the normal distribution

Probability density function	$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$
Cumulative probability function	$F(x) = \frac{1}{2}\left(1 + \text{erf}\left(\frac{x-\mu}{\sigma\sqrt{2}}\right)\right)$
Mean	μ
Median	μ
Variance	σ^2

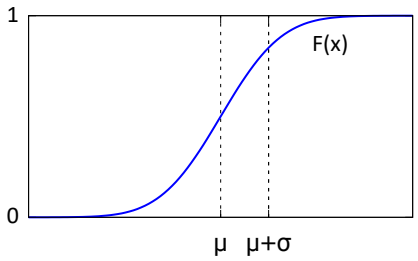


Figure A.2: Cumulative distribution function of the normal distribution

Normal distribution

The *normal distribution*, also called (or *Gaussian* or *Gauss* or *Gauss-Laplace distribution*) is one of the most convenient to represent phenomena issued from several random sources. It is defined by means of two parameters: its mean, usually denoted as μ, and its standard-deviation, usually denoted as σ, (or equivalently its variance σ^2). It is denoted $\mathcal{N}(\mu,\sigma)$.

Table A.2 gives the characteristics of the normal distribution. In this table, the function $\text{erf}(x)$ is the error function defined as follows.

$$\text{erf}(x) \stackrel{def}{=} \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\, dt \tag{A.2}$$

It gives the probability for a random variable with normal distribution of mean 0 and variance $1/2$ to fall in the interval $[-x,x]$.

Figure A.2 shows the shape the cumulative distribution function of the normal distribution.

Lognormal distribution

A *lognormal distribution* is a continuous probability distribution of a random variable whose logarithm is normally distributed. Thus, if the random variable X is lognormally distributed, then $Y = \ln X$ has a normal distribution. A lognormal process is the statistical realization of the multiplicative product of many independent random variables, each of which is positive. As for the normal distribution, it is characterized by its mean μ and standard-deviation σ.

Table A.3 gives the characteristics of the lognormal distribution.

Figure A.3 shows the shape the cumulative distribution function of the lognormal distribution.

Exponential distribution

The *exponential distribution* represents typically the life-span of a component without memory, aging nor wearing (Markovian hypothesis). The probability that the component is working at least $t+d$ hours knowing that it worked already t hours is the same as the probability that it works d hours after its entry into service. In other words, the fact that the component worked correctly for t hours does not change its expected life duration after this delay.

Table A.3: Characteristics of the lognormal distribution

Probability density function	$f(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x-\mu)^2}{2\sigma^2}\right)$
Cumulative probability function	$F(x) = \frac{1}{2}\left(1+\text{erf}\left(\frac{\ln x-\mu}{\sigma\sqrt{2}}\right)\right)$
Mean	$\exp\left(\mu+\frac{\sigma^2}{2}\right)$
Median	$\exp(\mu)$
Variance	$\left(\exp\left(\sigma^2\right)-1\right)\exp\left(2\mu+\sigma^2\right)$

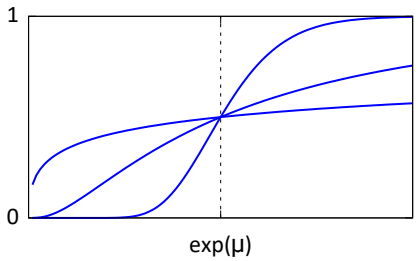


Figure A.3: Cumulative distribution function of the lognormal distribution

The exponential distribution is defined by means of a single parameter, the transition rate, usually denoted by λ. This transition rate is the inverse of the mean life expectation.

Table A.4 gives the characteristics of the exponential distribution.

Figure A.4 shows the shape the cumulative distribution function of the exponential distribution.

Weibull distribution

The exponential distribution assumes a constant failure rate over the time. This is not always realistic because of aging effects: at the beginning of its life the component has a decreasing failure rate, corresponding to debug (or infant mortality), then for a long while, its failure rate remains constant, then the wearout period starts where the failure rate increases. This is the so-called bathtub curve.

This phenomenon is (piece wisely) captured by the *Weibull distribution* which takes two parameters: he shape parameter, usually denoted α, and the scale parameter, usually denoted β.

Table A.5 gives the characteristics of the Weibull distribution. In this table, the Γ function is defined as follows.

$$\Gamma(z) \overset{def}{=} \int_0^\infty x^{z-1}e^{-x}dx \tag{A.3}$$

Figure A.4 shows the shape the cumulative distribution function of the Weibull distribution.

Table A.4: Characteristics of the exponential distribution

Probability density function	$f(x) = \lambda e^{-\lambda x}$
Cumulative probability function	$F(x) = 1-e^{-\lambda x}$
Mean	λ^{-1}
Median	$\lambda^{-1}\ln 2$
Variance	λ^{-2}

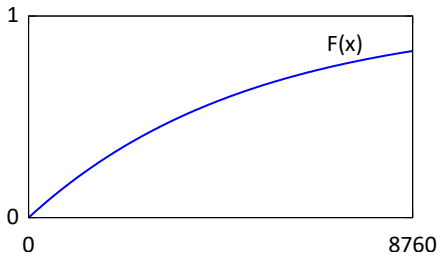


Figure A.4: Cumulative distribution function of the exponential distribution

Table A.5: Characteristics of the Weibull distribution

Probability density function	$f(x) = \frac{\beta}{\alpha}\left(\frac{x}{\alpha}\right)^{\beta-1} e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$
Cumulative probability function	$F(x) = 1 - e^{-\left(\frac{x}{\alpha}\right)^{\beta}}$
Mean	$\alpha\Gamma\left(1+\frac{1}{\beta}\right)$
Median	$\alpha(\ln 2)^{\frac{1}{\beta}}$
Variance	$\alpha^2\left(\Gamma\left(1+\frac{2}{\beta}\right) - \left(\Gamma\left(1+\frac{1}{\beta}\right)\right)^2\right)$

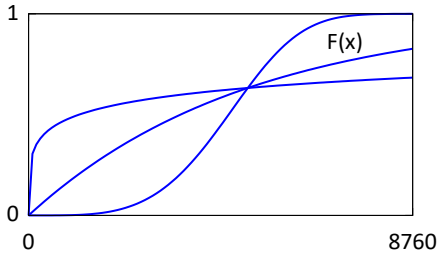


Figure A.5: Cumulative distribution function of the Weibull distribution

Table A.6: Characteristics of the triangular distribution

Probability density function	$f(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{2(x-a)}{(b-a)(c-a)} & \text{if } a \le x < c \\ \frac{2}{b-a} & \text{if } x = c \\ \frac{2(b-x)}{(b-a)(b-c)} & \text{if } c < x \le b \\ 0 & \text{if } x > b \end{cases}$
Cumulative probability function	$F(x) = \begin{cases} 0 & \text{if } x \le a \\ \frac{(x-a)^2}{(b-a)(c-a)} & \text{if } a < x \le c \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & \text{if } c < x < b \\ 1 & \text{if } x \ge b \end{cases}$
Mean	$\frac{a+b+c}{3}$
Median	$\begin{cases} a + \sqrt{\frac{(b-a)(c-a)}{2}} & \text{if } c \ge \frac{a+b}{2} \\ b - \sqrt{\frac{(b-a)(c-a)}{2}} & \text{if } c \le \frac{a+b}{2} \end{cases}$
Variance	$\frac{a^2+b^2+c^2-ab-ac-bc}{18}($

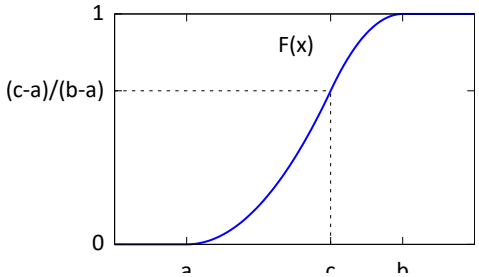


Figure A.6: Cumulative distribution function of the triangular distribution

Triangular distribution

The *triangular distribution* is a continuous probability distribution with lower limit a, upper limit b and mode c, where $a < b$ and $a \le c \le b$.

The triangular distribution is typically used as a subjective description of a population for which there is only limited sample data. It is therefore often used in business decision making when not much is known about the distribution of an outcome (say, only its smallest, largest and most likely values).

Table A.6 gives the characteristics of the triangular distribution.

Figure A.6 shows the shape the cumulative distribution function of the triangular distribution.

Binomial distribution

The *binomial distribution* of parameters n and p is a discrete probability distribution. It represents the number of successes in a series of n independent experiments, each asking a yes–no question. Each experiment gives the answer yes with the probability p and no with the probability $q = 1 - p$.

The binomial distribution is frequently used to model the number of successes in a sample of size n drawn with replacement from a population of size N. In case the sample is drawn without replacement,so the resulting distribution is a hypergeometric distribution. However, if N much larger than n, the probability to draw twice the same individual is very low, so the binomial

Table A.7: Characteristics of the binomial distribution

Probability mass function	$f(k) = \binom{n}{k} p^k (1-p)^{n-k}$
Cumulative probability function	$F(k) = \sum_{i=0}^{\lfloor k \rfloor} \binom{n}{i} p^i (1-p)^{n-i}$
Mean	np
Median	$\lfloor np \rfloor$ or $\lceil np \rceil$
Variance	$np(n-p)$

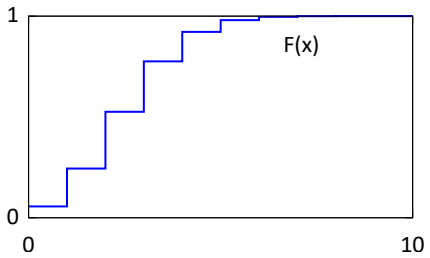


Figure A.7: Cumulative distribution function of the binomial distribution for $n = 10$ and $p = 0.25$

distribution is a good approximation of the hypergeometric distribution.

Table A.7 gives the characteristics of the binomial distribution.

Recall that the expression for the binomial is as follows.

$$\binom{n}{k} \stackrel{def}{=} \frac{n!}{k!(n-k)!} \tag{A.4}$$

Figure A.7 shows the shape the cumulative distribution function of the binomial distribution for $n = 10$ and $p = 0.25$.

The *Bernoulli distribution* is a special case of the binomial distribution where a single trial is conducted, i.e. $n = 1$.

A.2.2 Empirical distributions

Empirical distributions are given by a list of points $(x_1, y_1), \ldots, (x_n, y_n)$, $n \geq 2$, such that $x_1 < \ldots < x_n$ and, if one considers cumulative distribution functions, $y_1 \leq \ldots \leq y_n$. They are typically obtained via series of observations.

In between points, the value of the function is obtained by interpolation. There are basically two ways to do this interpolation:

– To consider the distribution as a *stepwise distribution*, i.e.

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases} \tag{A.5}$$

– To consider the distribution as a *piecewise uniform distribution*, i.e.

$$F(x) = \begin{cases} y_1 & \text{if } x < x_1 \\ y_i + (y_{i+1} - y_i)\frac{x - x_i}{x_{i+1} - x_i} & \text{if } x_i \leq x < x_{i+1} \\ y_n & \text{if } x \geq x_n \end{cases} \tag{A.6}$$

■ **Example A.1 – Piecewise uniform distribution.** Figure A.8 shows a piecewise uniform distribution defined by 4 points: (0,0), (1000, 0.3), (7000, 0.4) and (8760, 0.876). ■

Until recently, empirical distributions were mostly used when it was hard to fit them with any parametric distribution. For instance, the *Kaplan–Meier estimator* (Kaplan and Meier 1958), also

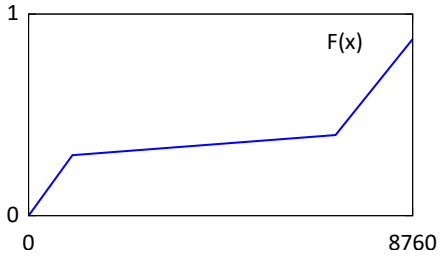


Figure A.8: Cumulative distribution function of the piecewise uniform distribution

known as the product limit estimator, is a non-parametric statistic used to estimate the survival function from lifetime data. In reliability engineering, Kaplan–Meier estimators may be used to measure the time-to-failure of machine parts. In medical research, they are often used to measure the fraction of patients living for a certain amount of time after treatment. This situation may generalize with easy internet communications: there is no more need to store data into a very compact form (the name of the parametric function and its parameters). Therefore, there is less and less reasons to spend time and energy in fitting empirical observations with parametric distributions, not to speak about the arbitrariness of the choice of the parametric distribution.

A.3 Basic Statistics

Essentially two types of indicators can be observed during Monte-Carlo simulations or similar types of experiments:

- Numerical indicators that are assimilated to real-valued variables.
- Discrete indicators such as Boolean variables or variables taking their values into a small set of symbolic constants.

The statistics made on these two types of indicators are different. For numerical indicators, one is mostly interested in how the values of the indicator are distributed. For discrete indicators, one is mostly interested in the frequency of each value of the indicator.

We shall consider in turn both cases. But before doing so, an important practical remark must be made: To get significant results, one often needs to consider very large sample, i.e. number of executions of the model. But in such case, storing all individual values taken by the indicators would lead to a memory overflow. It would be indeed possible to store values into an external memory (e.g. hard disk), but accesses to external memories are very slow. Doing so would therefore slow down dramatically the Monte-Carlo simulation. Consequently, statistics must be made using a reasonable amount of memory. As for modeling in general, there is here a tradeoff to find between the accuracy of the description and the ability to perform calculations.

A.3.1 Moments

In statistics, a *moment* is a specific quantitative measure of the shape of a function. If the function is a probability distribution, then the first moment is the *mean*, the second central moment is the *variance*, the third standardized moment is the *skewness*, and the fourth standardized moment is the *kurtosis*. The mathematical concept is closely related to the concept of moment in physics.

The n-th moment μ_n of a real-valued continuous function f of a real variable about a value c is defined as follows.

$$\mu_n \stackrel{def}{=} \int_{-\infty}^{\infty} (x-c)^n f(x) dx \tag{A.7}$$

The moment of a function, without further details, refers usually to the above expression with $c = 0$.

Note that the n-moment does not necessarily exist (because the above integral may be infinite).

If f is a probability density function, then the value of the integral above is called the n-th moment of the probability distribution. More generally, if F is a cumulative probability distribution function of any probability distribution, which may not have a density function, then the n-th moment of the probability distribution is given by the following integral.

$$\mu'_n \stackrel{def}{=} E[X^n] = \int_{-\infty}^{\infty} x^n dF(x) \tag{A.8}$$

where X is a random variable that has this cumulative distribution F, and E is the expectation operator or mean.

Mean

The *mean* of a probability distribution is thus the long-run arithmetic average value of a random variable having that distribution. In this context, it is also known as the *expected value* (see Section A.1).

For a data set $S = \{x_1, x_2, \ldots, x_n\}$, typically obtained via a Monte-Carlo simulation, the *arithmetic mean*, also called the *mathematical expectation* or *average*, typically denoted by $\bar{x}$ (pronounced "x bar"), is the sum of the values divided by the number of values in the data set.

$$\bar{x} \stackrel{def}{=} \frac{\sum_{i=1}^{n} x_i}{n} \tag{A.9}$$

$\bar{x}$ must be distinguished from the mean μ of the underlying distribution. However, the law of large numbers ensures that the larger the size of the sample, the more likely it is that its mean is close to the actual population mean.

As pointed out above, in practice, it may not be possible to store all of the x_i's of the sample. It is however always possible to calculate their sum "on-the-fly", i.e. each time a new value is obtained, it is added to the current sum. Then, when the mean is to be calculated, it suffices to divide the accumulated sum by the number of values in the sample.

■ **Example A.2 – Muffins.** A cafeteria manager wants to analyze the number of muffins sold each day at the cafeteria, in order to better serve clients and avoid losses. To do so, she samples ten days at random over one month and gets the following results.

day	1	2	3	4	5	6	7	8	9	10
x_i	38	11	36	28	10	18	37	12	14	11

To estimate the mean number of muffins sold each day, she can just record, day after day, the sum of the numbers of muffins sold so far.

day	1	2	3	4	5	6	7	8	9	10
$\sum x_i$	38	49	85	113	123	141	178	190	204	215

Then, to get her estimate, she has just to divide the last sum by the number of days: $\bar{x} = \frac{215}{10} = 21.5$. Note that the mean is not an integer, while obviously the cafeteria does not sold fractions of muffins.

■

Variance and standard-deviation

As for the mean, we need to estimate the variance and the standard-deviation from the sample (see Appendix A for mathematical developments on these two indicators).

Recall that the *variance* of a random variable X, denoted $\mathrm{Var}(X)$, is the expectation of the squared deviation of X from its mean μ:

$$\mathrm{Var}(X) \stackrel{def}{=} E\left[(X-\mu)^2\right]$$

Intuitively, $\mathrm{Var}(X)$ measures how far a set of (random) numbers are spread out from their average value.

The *standard-deviation* of a random variable X, denoted $\sigma(X)$, is the square root of its variance.

$$\sigma(X) \stackrel{def}{=} \sqrt{\mathrm{Var}(X)}$$

The expression defining the variance can be expanded:

$$\begin{aligned}\mathrm{Var}(X) &\stackrel{def}{=} E\left[(X-E[X])^2\right]\\ &= E\left[X^2-2XE[X]+E[X]^2\right]\\ &= E[X^2]-2E[X]E[X]+E[X]^2\\ &= E[X^2]-E[X]^2\end{aligned}$$

The naïve algorithm to estimate the empirical variance $\overline{\mathrm{Var}}(X)$ (and the empirical standard deviation $\overline{\sigma}(X)$) from a sample consists simply in applying the above formula, i.e. in calculating the mean of the squares and the square of the mean of the values in the sample, and in dividing their difference by the number of values, i.e. To do so, it suffices to accumulate the sum of the squares of the values and the sum of the values, as for the mean.

$$\overline{\mathrm{Var}}(X) \stackrel{def}{=} \frac{\sum_{i=1}^n x_i^2}{n} - \left(\frac{\sum_{i=1}^n x_i}{n}\right)^2 \tag{A.10}$$

$$\overline{\sigma}(X) \stackrel{def}{=} \sqrt{\overline{\mathrm{Var}}(X)} \tag{A.11}$$

In case the two components of the right hand side of equation A.10 are similar in magnitude, this algorithm may suffer from biases and numerical instability. Fortunately, there exist better algorithms to handle these cases. A first correction is provided by the Bessel's formula, which consists in multiplying the variance obtained by the naïve algorithm by a factor $\frac{n}{n-1}$. This does not solve however numerical instability. The Welford's online algorithm makes it possible to get a good estimate of the variance "on-the-fly" (Welford 1962).

In most of practical cases however, the naïve algorithm is good enough.

■ **Example A.3 – Muffins (bis).** Consider again our cafeteria example. To estimate the variance and the standard-deviation of the number of muffins sold, the cafeteria manager can just record, day after day, the sum of the squares of the numbers of muffins sold (in addition to the sum of the numbers of muffins sold):

day	1	2	3	4	5	6	7	8	9	10
x_i	20	16	30	28	21	19	20	21	18	22
$\sum x_i$	20	256	900	784	441	361	400	441	324	215
x_i^2	400	121	1296	784	100	324	1369	144	196	484
$\sum x_i^2$	400	656	1556	2340	2781	3142	3542	3983	4307	4791

Then, she can just apply the naïve algorithm: $\overline{\text{Var}}(X) = \frac{4791}{10} - 21.5^2 = 16.85$ and $\overline{\sigma}(X) = 4.10$. She can notice that, as intuitively expected from the observations, the variance and the standard-deviation are rather large. ■

Error factors and confidence intervals

A *point estimate* is a single value given as the estimate of a population parameter of interest, for example, the mean. In contrast, an *interval estimate* specifies a range within which the parameter is estimated to lie *Confidence intervals* are commonly reported along with point estimates of the same parameters, to show the reliability of the estimates. A confidence interval comes with a confidence level, which specifies the probability that the actual value of the parameter lies in the given interval. For a same sample, the smaller the confidence range, the smaller the confidence level, and vice-versa, the larger the confidence level the larger the confidence range. Most commonly, the 95% confidence level is used. However, other confidence levels are also used, for example, the 90% and the 99% confidence levels.

The *margin of error* or *error factor* is usually defined as the "radius" (or half the width) of a confidence interval.

Let $\bar{x}$ and $\overline{\sigma}(x)$ be respectively the observed mean and standard-deviation on a sample of size n. Then, the error factor $\text{EF}_\alpha(x)$ corresponding to a confidence level α is defined as follows.

$$\text{EF}_\alpha(x) \stackrel{def}{=} t_\alpha \times \frac{\overline{\sigma}(x)}{\sqrt{n}} \tag{A.12}$$

The confidence interval $\text{CI}_\alpha(x)$ is then defined as follows.

$$\text{CI}_\alpha(x) \stackrel{def}{=} [\bar{x} - \text{EF}_\alpha(x), \bar{x} + \text{EF}_\alpha(x)] \tag{A.13}$$

The factor t_α is obtained, assuming a normal distribution of the values, by looking at the table defining the normal law. Typical values chosen for t_α are $t_{90\%} = 1.64$, $t_{95\%} = 1.96$ and $t_{99\%} = 2.58$.

■ **Example A.4 – Muffins (ter).** Consider again the cafeteria example. Based on her previous calculations, the cafeteria manager can now calculate errors factors and confidence intervals for the number of muffins sold daily (recall that $\bar{x} = 21.5$ and that $\sigma(x) = 11.30$).

Confidence level	Error factor	Confidence interval
90%	2.13	[19.37, 23.63]
95%	2.54	[18.96, 24.04]
99%	3.35	[18.15, 24.85]

The above intervals can be interpreted as follows. There are 90 chances out of 100 that the mean number of muffins sold daily lies somewhere between 19.37 and 23.63, 95 chances out of 100 that it lies between 18.96 and 24.04, and finally 99 chances out of 100 that it lies between 18.15 and 24.85.

Note, and this is very important to understand, that the parameter that is estimated is the mean number of muffins daily sold, not the number itself.

Note also that these figures assume a normal distribution for the number of muffins daily sold. However, a simple look at the data shows two "outliers" at day 3 and 4. It may be the case that, these two days are the two Wednesday's of the sample, which turn out to be the day of the weekly gathering of the Knitting Angels, a gang of grey-haired who use to come at the cafeteria to sip a coffee and eat pastries. The indicators calculated so far would much more informative by treating these two days separately. ■

A.3.2 Distributions and quantiles

For symbolic values, the calculation of the moments is sufficient. For numerical values, one may be interested in addition to the distribution of values as well as quantiles.

Distributions

In principle, extracting a distribution is rather simple. Let $x_1,\ldots,x_n$ the n numerical values in the sample. The extraction of the distribution is done in three steps:

1. The minimum and maximum values of the sample are determined (by scanning all x_i's).
2. The interval between the minimum value and the maximum value is split into k sub-intervals of same size $I_1,\ldots,I_k$.
3. By scanning again all x_i's, one determines how many values lie in each interval. The cumulative distribution function is obtained by summing the number of values in intervals preceding the considered intervals.

The problem with this approach is indeed that it requires to store all of the values. So in practice, the minimum and maximum values (and therefore the intervals) are chosen *a priori* in such way that all values lie between them and that they are not too far from the actual minimum and maximum. The number of values in each interval is then maintained "on-the-fly".

■ **Example A.5 – Muffins (quater).** Consider again the cafeteria example. The cafeteria manager may decide *a priori* that the number of muffins sold daily ranges from 15 to 30, and to split this interval into four sub-intervals: [15,18], [19,22], [23,26] [27,30] She can then follow on a day by day basis, the evolution of the number of muffins sold daily falling in each of these four sub-intervals.

day	1	2	3	4	5	6	7	8	9	10
x_i	20	16	30	28	21	19	20	21	18	22
[15,18]	0	1	1	1	1	1	1	1	2	2
[19,22]	1	1	1	1	2	3	4	5	5	6
[23,26]	0	0	0	0	0	0	0	0	0	0
[27,30]	0	0	1	2	21	2	2	2	2	2

This distribution is quite informative: it shows that most of the days, the number of muffins sold lies in the second sub-interval, i.e. between 19 and 22. It shows also that there are two outliers, lying in the fourth sub-interval. The fact that they are "exceptional" is confirmed by the emptiness of the third sub-interval. ■

Quantiles

An alternative approach consists in calculating quantiles.

Quantiles are cut points dividing the range of a probability distribution into successive intervals with equal probabilities, or dividing the observations in a sample in the same way to estimate their values. Common quantiles have special names: *quartiles* (when the probability distribution is divided in 4), *deciles* (when it is divided in 10), *centiles* (when it is divided in 100). The *median* is the point such that half of the values in the sample are below and half are above, i.e. the sample is divided into two sub-intervals.

In principle, estimating quantiles is rather simple. Let $x_1,\ldots,x_n$ the n numerical values in the sample. The extraction of quantiles is done in three steps:

1. The x_i's are sorted in ascendant order.
2. The sorted list of the x_i's is splitted into q sub-list of equal size (where q depends on which quantiles one wants to obtain).
3. The i-th q-quantile is the highest value in the i-th sorted sub-list.

Making the above principle to work "on-the-fly", i.e. without recording all the x_i's is rather tricky: only approximated values can be obtained. A full presentation of algorithms that perform such on-the-fly calculation goes beyond the scope of this book. The main idea is to maintain successive "bins" of equal probabilities. Values accumulated in each bin are considered as random variables, for which a mean, a standard-deviation as well as extremum values can be calculated "on-the-fly".

■ **Example A.6 – Muffins (cinque).** Consider again the cafeteria example. The cafeteria manager may decide to calculate quartile. To do so she needs first to sort x_i's. Then to split the sorted list into 4 sub-lists of about equal size. E.g.

sorted x_i	16	18	19	20	20	21	21	22	28	30
quartile	1		2			3			4	

The first quartile is thus between 18 and 19, the second one that is the median, between 20 and 21, the third one between 22 and 28, finally the last one is 30. ■

A.4 Random-Number Generators

The Monte-Carlo simulation method relies fully on the generation at random of numbers. This section gives some hints on how this generation is performed in practice.

A.4.1 Algorithmic generators

A *random-number generator* is a device that generates a sequence of numbers that cannot be reasonably predicted better than by a random chance. This definition is somehow circular as it relies on the concept of random chance, but the intuitive idea is there. A full mathematical treatment of the subject goes much beyond the scope of this book (see Section 1.4 for reading advices).

Random number generators can be hardware random-number generators, which generate genuinely random numbers, or pseudo-random number generators, i.e. algorithms which generate series of numbers which look random, but are actually deterministic. The former are not very convenient, at least in the context of model-based systems engineering, for they require to connect computers with such devices. The latter present not only the advantage of being cheaper, but also that series of numbers generated by the algorithm can be reproduced at will.

It remains to find "good" generation algorithms, i.e. algorithms that mimic as much as possible "true" randomness, while being not too expensive from a computational view point. Algorithmic generators can actually suffer from several defects:

- Lack of uniformity of distribution for large quantities of generated numbers;
- Correlation of successive values;
- Poor dimensional distribution of the output sequence;
- The distances between where certain values occur may be distributed differently from those in a "true" random sequence distribution;

Congruential pseudo-random number generators (attempt to) fulfill the needs. A *congruential pseudo-random number generators* is a function f that takes an integer as input (coded onto a finite number of bits, e.g. 64 bits on modern computers), called the *seed*, and returns an integer. Given the initial seed z_0, it is thus possible to generate an arbitrary long sequence of numbers: $z_1 = f(z_0)$, $z_2 = f(z_1), \ldots$

In the second half of the 20th century, the standard pseudo-random number generators were linear congruential generators, i.e. generators based on functions of the form:

$$f(z) \stackrel{def}{=} (a \times z + c) \mod m$$

Note that, because congruential generators work with only a finite number of integers (those coded by the machine), there necessary exist two indices i and j, $i < j$, such as $z_i = z_j$. Consequently, at some step, the generator loops. The distance between i and j is called the *period* of the generator. For some seeds, the period may be shorter than for some others.

The periodicity of congruential pseudo-random number generators was really an issue for Monte-Carlo simulations when integers were coded on 32 bits. For large number of tries (e.g. 1 million), they were good chances that the same executions were reproduced (as they started with the same seed). The problem was known to be inadequate, but better methods were unavailable.

Fortunately, this problem is now solved, due to the generalization of 64 bits machines, and more importantly to the introduction of techniques based on linear recurrences on the two-element field. With that respect, the invention of the Mersenne Twister generator in 1997 (Matsumoto and Nishimura 1998) was a major step forward. This generator (or a successor of thereof) is nowadays implemented in random number generation librairies of programming languages such as Python, Matlab, R...

A.4.2 Generation according to probability distributions

The algorithmic random generators we presented so far generate numbers ranging from 0 to the biggest integer `maxint` representable in one machine word (i.e. on 64 bits, for the most part of computers we are using today).

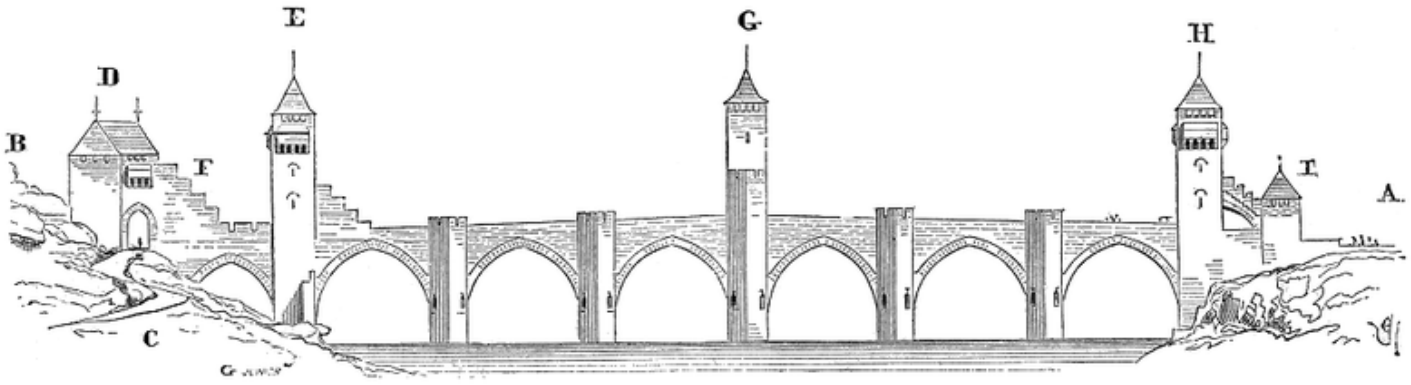
However, what we really would like is to generate real numbers, according to some predefined distribution such as those presented Section A.2.

To get floating point numbers uniformly distributed between 0 and 1, it suffices to divide the generated x_i's by `maxint`.

Having a uniform generator between 0 and 1, it is easy to transform it into a generator according to most of the distributions we have seen so far: for parametric distributions with an invertible cumulative distribution function, it suffices to generate a number z between 0 and 1, and then to take $x = F^{-1}(z)$. This works fine for uniform, exponential, Weibull and triangular distributions. The same idea applies also to empirical distributions.

The situation is more complex when considering normal (and thus lognormal) distributions, as there is no easily calculable inverse function. Fortunately, there exist relatively simple methods to generate a normal distribution from an uniform distribution, e.g. the Box-Muller method (Box and Muller 1958).

Note again that the above mentioned librairies of programming languages such as Python provide built-ins to generate floating point numbers according to wide variety of parametric distributions.



B. S2ML+DFE

B.1 Models

```
Model ::= (DomainDeclaration | ClassDeclaration | BlockDeclaration)*

DomainDeclaration ::=
    domain Identifier "{" Identifier ( "," Identifier )* "}"

ClassDeclaration ::=
    class Identifier BlockField* end

BlockDeclaration ::=
    block Identifier BlockField* end

BlockField ::=
        ParameterDeclaration | VariableDeclaration | ObserverDeclaration
    |   BlockDeclaration | InstanceDeclaration
    |   ExtendsDirective | EmbedsDirective | ClonesDirective
    |   EquationDeclaration
```

B.2 Parameters, Variables and Observers

```
ParameterDeclaration ::=
    parameter DomainIdentifier Path "=" Expression ";"

VariableDeclaration ::=
    DomainIdentifier Path ("," Path )* Attributes? ";"

ObserverDeclaration ::=
    observer DomainIdentifier Path "=" Expression ";"

Attributes ::=
    "(" Attribute ( "," Attribute )+ ")"
Attribute ::=
    Identifier "=" Expression

DomainIdentifier ::=
    Boolean | Integer | Real | Identifier
```

B.3 Equations

```
EquationDeclaration ::=
    assertion Equation*

Equation ::=
    Assignment | DoubleAssignment

Assignment ::=
    Path ":=" Expression ";"

DoubleAssignment ::=
    Path ":=:" Path ";"
```

B.4 Expressions

```
Expression ::=
        Path
    |   ArithmeticExpression | ArithmeticBuiltIn | TrigonometricFunction
    |   BooleanExpression | Inequality
    |   RandomDeviate
    |   ConditionalExpression | SpecialBuiltIn
    |   "(" Expression ")"


ConditionalExpression ::=
    if BooleanExpression then Expression else Expression

SpecialBuiltIn ::=
    missionTime()
```

B.4.1 Arithmetic Expression

```
ArithmeticExpression ::=
        Float
    |   pi
    |   ArithmeticExpression ( "+" ArithmeticExpression )+
    |   ArithmeticExpression ( "-" ArithmeticExpression )+
    |   ArithmeticExpression ( "*" ArithmeticExpression )+
    |   ArithmeticExpression ( "/" ArithmeticExpression )+
    |   "-"  ArithmeticExpression
    |   min "(" ArithmeticExpression ( "," ArithmeticExpression )*  ")"
    |   max "(" ArithmeticExpression ( "," ArithmeticExpression )* ")"
```

B.4.2 Built-Ins

```
ArithmeticBuiltIn ::=
    |   exp "(" ArithmeticExpression ")"
    |   log "(" ArithmeticExpression ")"
    |   pow "(" ArithmeticExpression "," ArithmeticExpression ")"
    |   sqrt "(" ArithmeticExpression ")"
    |   floor "(" ArithmeticExpression ")"
    |   ceil "(" ArithmeticExpression ")"
    |   abs "(" ArithmeticExpression ")"
    |   mod "(" ArithmeticExpression "," ArithmeticExpression ")"
    |   Gamma "(" ArithmeticExpression ")"
```

B.4.3 Trigonometric Function

```
TrigonometricFunction ::=
    |   sin "(" ArithmeticExpression ")"
    |   cos "(" ArithmeticExpression ")"
    |   tan "(" ArithmeticExpression ")"
    |   asin "(" ArithmeticExpression ")"
    |   acos "(" ArithmeticExpression ")"
    |   atan "(" ArithmeticExpression ")"
```

B.4.4 Boolean Expressions and Inequalities

```
BooleanExpression ::=
    |   BooleanExpression ( or BooleanExpression )+
    |   BooleanExpression ( and BooleanExpression )+
    |   not BooleanExpression
    |   count "(" BooleanExpression ( "," BooleanExpression )* ")"

Inequality ::=
    |   Expression "==" Expression
    |   Expression "!=" Expression
    |   ArithmeticExpression "<" ArithmeticExpression
    |   ArithmeticExpression ">" ArithmeticExpression
    |   ArithmeticExpression "<=" ArithmeticExpression
    |   ArithmeticExpression ">=" ArithmeticExpression
```

B.4.5 Random Deviates

```
RandomDeviate ::=
        uniformDeviate "(" [ArithmeticExpression]2 ")"
    |   normalDeviate "(" [ArithmeticExpression]2 ")"
    |   lognormalDeviate "(" [ArithmeticExpression]2 ")"
    |   triangularDeviate "(" [ArithmeticExpression]3 ")"
    |   exponentialDeviate "(" ArithmeticExpression ")"
    |   WeibullDeviate "(" [ArithmeticExpression]2 ")"
```

B.5 S2ML Directives

```
InstanceDeclaration ::=
        Identifier Identifier ";"               # ClassName InstanceName
    |   Identifier Identifier BlockField* end   # idem

ClonesDirective ::=
        clones Path as Identifier ";"           # BlockPath CloneName
    |   clones Path as Identifier BlockField* end   # idem

ExtendsDirective ::=
        extends Identifier ";"                  # ClassName

EmbedsDirective ::=
        embeds Path as Identifier ";"           # BlockPath LocalName
    |   embeds Path as Identifier BlockField* end   # idem
```

B.6 Identifiers, Paths, Constants and Comments

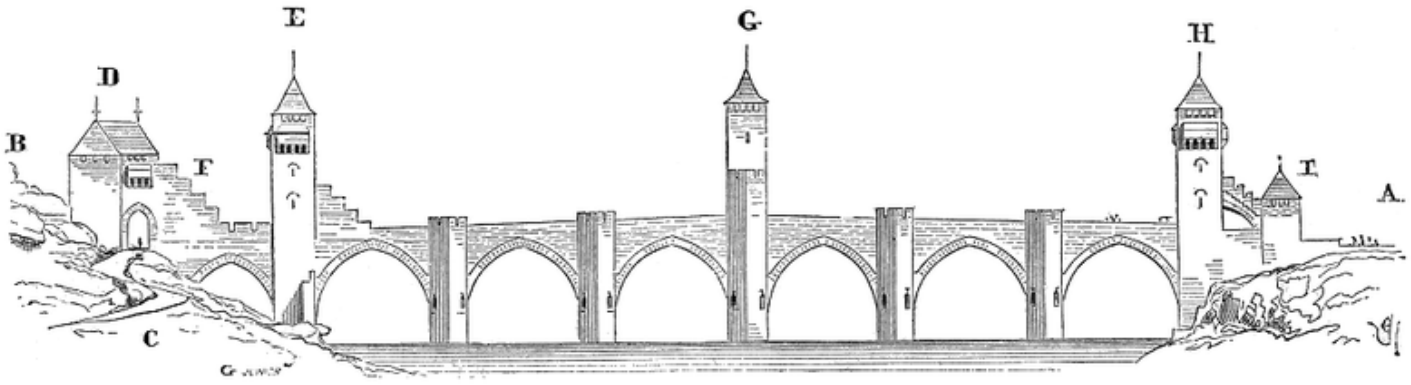
```
Identifier ::= [a-zA-Z_][a-zA-Z0-9_-]+

Path ::= Identifier ( "." Identifier )*

Integer ::= [0-9]+

Float ::= [+-]? [0-9]+ ("." [0-9]+)? ([eE] [+-]? [0-9]+)?
```

Comments can be added everywhere in the code.
- Single line comments introduced by "//", which comment out the rest of the line.
- Multiline comments which comment out the text between "/*" and "*/".



C. Janos Commands

C.1 Scripts

```
Script ::= Command*

Command ::=
        CommandLoad
    |   CommandInstantiate
    |   CommandSet
    |   CommandReset
    |   CommandProfile
    |   CommandCompute
    |   CommandPrint
```

C.2 Commands

```
CommandLoad ::=
        load model fileName
    |   load script fileName

CommandInstantiate ::=
        instantiate model
    |   flatten model
```

```
CommandSet ::=
        set target-block Path
    |   set number-of-tries Integer
    |   set seed Integer
    |   set Identifier Value
```

```
CommandReset ::=
        reset model
    |   reset target-block
```

```
CommandProfile ::=
        profile new Identifier
    |   profile clone Identifier Identifier
    |   profile delete Identifier
    |   profile set mean Identifier Boolean
    |   profile set standard-deviation Identifier Boolean
    |   profile set confidence-range-90 Identifier Boolean
    |   profile set confidence-range-95 Identifier Boolean
    |   profile set confidence-range-99 Identifier Boolean
    |   profile set extrema Identifier Boolean
    |   profile set quantiles Identifier Boolean
    |   profile set number-of-quantiles Identifier Integer
    |   profile set distribution Identifier Boolean
    |   profile set cumulative-distribution Identifier Boolean
    |   profile set number-of-points Identifier Integer
```

```
CommandCompute ::=
    compute observers blockName Option*

Option ::=
        mission-times = "[" Real ("," Real)* "]"
    |   profile "=" Identifier
    |   output "=" String
    |   mode "=" (write|append)
```

```
CommandPrint ::=
        CommandPrintModel
    |   CommandPrintTargetModel
    |   CommandPrintEnvironment

CommandPrintModel ::=
    print model Option*

CommandPrintInstantiatedModel ::=
    print target-model Option*

CommandPrintEnvironment ::=
    print environment Option*

Option ::=
        output "=" String
    |   mode "=" (write|append)
```

All characters comprised between a "#" symbol and the end of the line are considered as part of a comment.