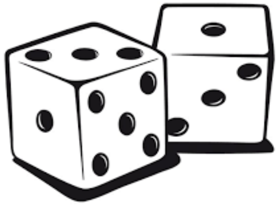
NTNU – Trondheim
Norwegian University of
Science and Technology

Janos

A pedagogical stochastic simulator.



Antoine Rauzy
Antoine.Rauzy@ntnu.no

Licenses & versions

Version	1.2.1
Date	23.03.2022

John von Neumann (Hungarian: Neumann János 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.

John von Neumann
1903 - 1957

Source Wikipedia

Table of contents

INTRODUCTION

Rational

Since its introduction in the late 1940s by Stanislaw Ulam and John von Neumann, the **Monte-Carlo method** is 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 a pedagogical stochastic simulator:

- Models are systems of **data-flow equations** written in the **S2ML+DFE domain specific modeling language**, which is the combination of S2ML (S2ML stands for system structure modeling language), a set of **object-oriented constructs** to structure models and systems of data-flow equations.
- It implements the Monte-Carlo simulation.
- It comes as a **command interpreter**, making it possible to perform various studies.

This presentation specifies S2ML+DFE as well as the commands to manage models and launch simulations.

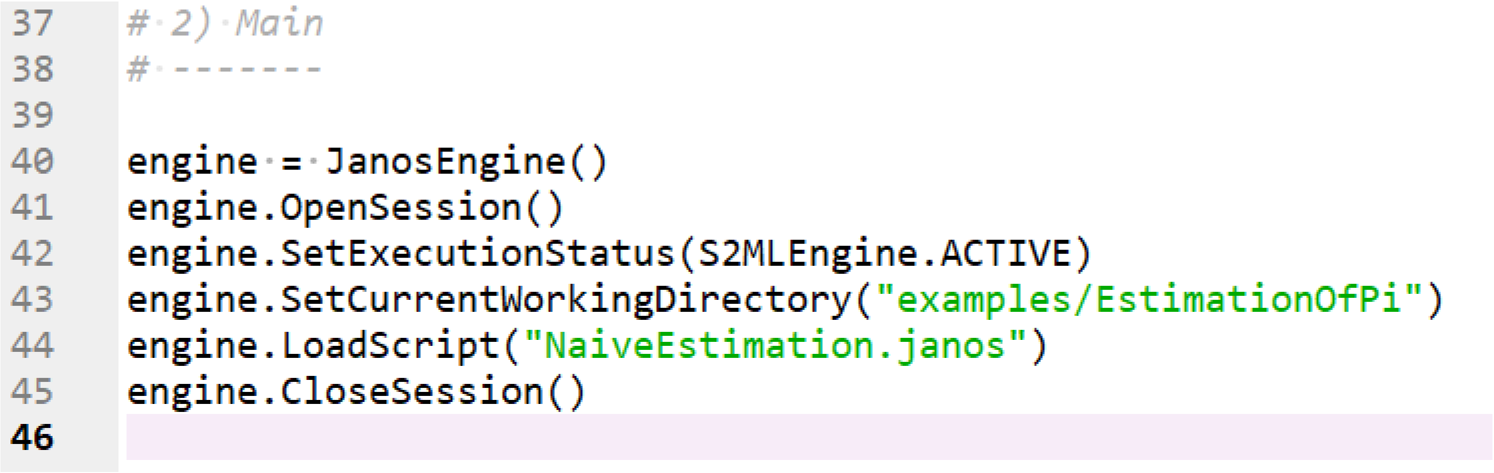
Janos is developed in Python, for pedagogical purposes only. It is by orders of magnitude less efficient than available commercial tools.

The objective is to familiarize students with Monte-Carlo simulation and modeling languages.

Installing and running Janos

To install Janos you just need to decompress the archive "Janos1.2.0.zip" into 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 current directory and the script file and run it.



```
# 2) Main
# -------

engine = JanosEngine()
engine.OpenSession()
engine.SetExecutionStatus(S2MLEngine.ACTIVE)
engine.SetCurrentWorkingDirectory("examples/EstimationOfPi")
engine.LoadScript("NaiveEstimation.janos")
engine.CloseSession()

```

Organization of this document

The remainder of this document is organized as follows.

- Section Getting started is a small introduction to Janos.

The two next sections describe S2ML+DFE:

- Section Basic components presents the core of the language.
- Section Structuring constructs presents object-oriented constructs to structure models.

The next section describes the command interpreter:

- Section Commands describes Janos commands.

Finally, the appendix completes this document.

- Appendix S2ML+DFE gives the Backus-Naur form of the modeling language.
- Appendix Janos gives the Backus-Naur form of Janos commands.
- Appendix Known bugs reports know problems with the current version of Janos.

GETTING STARTED

S2ML+DFE

S2ML+DFE is a textual format to describe systems of data-flow equations. E.g.

```
block Main
    Real x, y;
    Boolean inDisk;
    assertion
        x := uniformDeviate(0.0, 1.0);
        y := uniformDeviate(0.0, 1.0);
        inDisk := x*x + y*y <= 1.0;
    observer Real piEstimator = if inDisk then 4.0 else 0.0;
end
```

- Each model is described in a **block** which contains declarations of objects of the model. A block starts with keyword **block** followed by the name of the model (here `Main`) and ends with the keyword **end**.
- Four types of basic objects are used to define systems of data-flow equations: **parameters**, **variables**, **equations** and **observer**. Variables and parameters must be declared before they are referred to in equations.

S2ML+DFE (bis)

```
block Main
   Real x, y;
   Boolean inCircle;
   assertion
      x := uniformDeviate(0.0, 1.0);
      y := uniformDeviate(0.0, 1.0);
      inDisk := x*x + y*y <= 1.0;
   observer Real piEstimator = if inDisk then 4.0 else 0.0;
end
```

- Variables have a **type** and a **name**. Here three variables are declared: `x` and `y` which are **real** and `inCircle` which is **Boolean**.
- The value of variables is calculated by means of **equations**. Each equation is in the form v := E, where v is a variable and E is an expression. For each variable, there must be one and only one equation whose left member is the variable. Moreover, the set of equations must be **data-flow**, i.e. that a variable cannot depend on itself.

S2ML+DFE (ter)

```
block Main
   Real x, y;
   Boolean inCircle;
   assertion
      x := uniformDeviate(0.0, 1.0);
      y := uniformDeviate(0.0, 1.0);
      inDisk := x*x + y*y <= 1.0;
   observer Real piEstimator = if inDisk then 4.0 else 0.0;
end
```

- Expressions can be **deterministic** or **stochastic**. Here:
  - The values of `x` and `y` are drawn at random uniformly in the range [0, 1].
  - The value of `inCircle` is fully determined by the values of x and y, namely `inCircle` is true if and only if the sum of the squares of `x` and `y` is less or equal to 1.

S2ML+DFE (quater)

```
block Main
   Real x, y;
   Boolean inCircle;
   assertion
      x := uniformDeviate(0.0, 1.0);
      y := uniformDeviate(0.0, 1.0);
      inDisk := x*x + y*y <= 1.0;
   observer Real piEstimator = if inDisk then 4.0 else 0.0;
end
```

- **Observers** have a type and name. They are defined by a expression. This expression must be deterministic. Here:
  - `piEstimator` is equal to 4.0 if `inCircle` is true and 0.0 otherwise.
- Observers are the quantities on which **statistics** are made.

Assessment process

The assessment process of a model is typically made of the following steps:

1. The model is loaded from a text file.

2. The model is checked and rewritten in a form in which the simulation process can start. This step is called instantiation in the S2ML jargon.

3. The parameters of the stochastic simulation are defined.

4. The stochastic simulation is actually performed. Results are printed out into text files.

Scripts

Janos is a **command interpreter**: it reads commands into a text file and execute them. There are commands to perform each of the steps described in the previous slide.

Scripts are text files. Although this is not mandatory, models are usually stored into text files with the extension ".janos".

```
# Step 1: the model is loaded
load "Pi.dfe"
# print model "model.dfe"

# Step 2: the model is instantiated
flatten model
# print target-model "target.dfe"
```

Scripts (bis)

```
# Setting parameters of the stochastic simulation
set seed 23456
set number-of-tries 100000

profile set mean profile1 true
profile set standard-deviation profile1 true
profile set confidence-interval-90 profile1 false
profile set confidence-interval-95 profile1 true
profile set confidence-interval-99 profile1 false
profile set extrema profile1 true
profile set quantiles profile1 false
profile set distribution profile1 true
profile set cumulative-distribution profile1 true
profile set number-of-points profile1 10
# print environment "environment.txt"
```

Scripts (ter)

```
# Launching stochastic simulation
compute observers Main mission-times [0] \
    output="result.csv" mode=write
```

Results

results.csv

Model	Main						
Number-of-tries	100000						
Observer	piEstimator						
	Mission-time	0					
		Mean	Standard-deviation	95% confidence range		Minimum	Maximum
		3.14876	1.63719	3.13861	3.15891	0	4

In result files, items are separated with tabs so that results can be easily loaded into spreadsheets (Excel or equivalent).

S2ML+DFE:

BASIC COMPONENTS

Basic components

Basic components of S2ML+DFE models are:

- **Blocks** that contain declarations of other objects of a model.
- Declarations of **parameters**.
- Declarations of **variables**.
- Declarations of **equations**.
- Declarations of **observers**.

In the sequel, models are described using `this font`. Keywords are underlined using **`this font`**.

S2ML+DFE models must be written using ASCII characters.

Identifiers, i.e. names of blocks, states, ports and parameters (and other objects introduced in the next sections) obeys the following syntax:

- They start with a letter from a to z or from A to Z.
- They are made of any number of letters, digits, underscores "_".

E.g. `Plant`, `failed`, `R3151`, `this_is_a_valid_although_a_big_long_name`.

Comments

It is possible to add **comments** everywhere in a S2ML+DFE model.

- Any sequence of text between `/*` and `*/` is a comment, even if it spreads over several line.
- All characters after `//` until the end of the line is a comment.

In the sequel, we shall color comments *in italic and green*.

```
/*
 * This is a comment before a block declaration
 */
block Plant // This is a comment till the end of the line
    // declarations
end
```

Blocks

Blocks are the basic container of S2ML+DFE. They are **prototypes** in the sense of object-orientation theory. Blocks contain declarations of parameters, states, ports and sources (and other elements that will be described later).

A block declaration starts with the keyword "block", followed by the name of the block. It finishes with keyword "end". E.g.

```
block Main
    …
end
```

Within a block, all elements must have a different name, even though they are of different types, e.g. a variable and a parameter. Elements can be declared in any order, but parameters and variables must be declared before they are used in equations.

Domains (basic types)

Parameters, variables and observers are typed. Basic types are:

- `Boolean`, i.e. either `true` or `false`.
- `Integer`.
- `Real`.

It is also possible to declare **domains**, i.e. set of symbolic constants. Parameters and variables can be then declared with such domain (observers must be integers or reals).

```
domain State {WORKING, DEGRADED, FAILED)

block Main
    …
    State _state;
    …
end
```

Parameters

Parameters are used in arithmetic expressions. They are declared together with the expression that defines them. E.g.

```
parameter Real baseCapacity = 100.0;
parameter Real specialCapacity = 2 * baseCapacity;
```

Expressions defining parameters must be deterministic. Parameters cannot depend on variables and observers. They can depend on other parameters. However, a parameter cannot depend on itself.

Variables and equations

Variables can be declared either individually or several at a time. E.g.

```
Real x, y, z;
```

The value of variables is calculated by means of equations. In the current version of Janos, there are two types of equations:

- Equations of the form x := E; where x is a variable and E is an expression.
- Equations of the form x :=: y; where both x and y are variables.

For each variable, there must be one and only one equation whose left member is the variable. Moreover, the set of equations must be **data-flow**, i.e. that a variable cannot depend on itself.

Equations of the second form are thus automatically rewritten in equations of the first form when the model is instantiated, possibly by swapping left and right members.

Observers

Observers are the quantities on which **statistics** are made. As parameters, they have a type and name. They are defined by a expression.

```
observer Real piEstimator = if inCircle then 4.0 else 0.0;
```

The type of observers is either integer or real. Expressions that define them must be deterministic. They can depend on parameters and variables, but not on other observers.

Expressions

The current version of Janos provides several types of expressions:

- Arithmetic expressions and built-in functions
- Trigonometric functions
- Boolean expressions
- Inequalities
- Random-deviates
- Conditional expressions
- Special built-in functions

In any expression, a reference to a parameter or a variable can be used, under the conditions given in infra, i.e.

- Parameters can only depend on parameters and cannot depend on themselves.
- Variables can depend on parameters and variables but cannot depend on themselves.

Arithmetic expressions

The current version of Janos implements the following arithmetic expressions.

Syntax	#arguments	Semantics
`Floating point number`	0	The number
`pi`	0	π
`e1 + … + en`	≥ 1	Sum of the arguments
`e1 – … – en`	≥ 1	First argument minus the others
`e1 * … * en`	≥ 1	Product of the arguments
`e1 / …  / en`	≥ 1	First argument divided by the others
`– e`	1	Opposite
`min``(e1, …, en)`	≥ 1	Minimum of its arguments
`max``(e1, …, en)`	≥ 1	Maximum of its arguments

Examples:

```
    0.8 * weight          max(f-g, g-f, 1.0)       3*(x+y)
    -e
```

Arithmetic built-in functions

The current version of Janos implements the following arithmetic built-ins.

Syntax	#arguments	Semantics
`exp(e)`	1	exponential
`log(e)`	1	(Natural) logarithm
`pow(e1, e2)`	2	Power
`sqrt(e)`	1	Square root
`floor(e)`	1	Largest integer under
`ceil(e)`	1	Smallest integer above
`abs(e)`	1	Absolute value
`mod(e1, e2)`	2	Modulo
`Gamma(e)`	1	Gamma function

Trigonometric functions

The current version of Janos implements the following arithmetic built-ins.

Syntax	#arguments	Semantics
`sin``(`*`e`*`)`	1	Sine
`cos``(`*`e`*`)`	1	Cosine
`tan``(`*`e`*`)`	1	Tangent
`asin``(`*`e`*`)`	1	Arc sine
`acos``(`*`e`*`)`	1	Arc cosine
`atan``(`*`e`*`)`	1	Arc tangent

Boolean expressions

The current version of Janos implements the following Boolean expressions.

Syntax	#arguments	Semantics
`false, true`	0	Boolean constants
`e1 and … and en`	≥ 1	Conjunction of the arguments
`e1 or … or en`	≥ 1	Disjunction of the aguments
`not e`	1	Negation
`count(e1, …, en)`	≥ 1	Number of true expressions in the list

Examples:

```
a and b          (f and g) or (not f and h)
not e
```

Inequalities

The current version of Janos accepts the following inequalities.

Syntax	#arguments	Semantics
e1 == e2	2	Equal
e1 != e2	2	Different
e1 < e2	2	Less than
e1 > e2	2	Greater than
e1 <= e2	2	Less or equal
e1 >= e2	2	Greater or equal

Examples:

`0.8==weight` `f - g < x * y`

Random deviates

The current version of Janos implements the following random deviates

Syntax	#arg	Semantics
`uniformDeviate``(`*l*`,` *h*`)`	2	The value is drawn at random uniformly in the range [l, h]
`normalDeviate``(`*m*`,` *s*`)`	2	The value is drawn at random according to a normal distribution of mean m and standard deviation s.
`lognormalDeviate``(`*m*`,` *s*`)`	2	The value is drawn at random according to a lognormal distribution of mean m and standard deviation s.
`triangularDeviate``(`*l*`,` *h*`,` *m*`)`	3	The value is drawn at random according to a triangular distribution of lower bound l, higher bound h and mode m.

Random deviates (bis)

The current version of Janos implements the following random deviates

Syntax	#arg	Semantics
`exponentialDeviate``(`*`r`*`)`	1	The value is drawn at random according to a the inverse of a negative exponential distribution of rate r.
`WeibullDeviate``(`*`a`*`, `*`b`*`)`	2	The value is drawn at random according to the inverse of a Weibull distribution of scale parameter a and shape parameter b.

Conditional expressions and special built-in

The current version of Janos implements the following conditional expressions.

Syntax	#arg	Semantics
`if c then e1 else e2`	3	Equal to e1 if c is true and to e2 otherwise

Example:

```
if x<=y then 1.0 else 2.0
```

The current version of Janos implements the following special built-in.

Syntax	#arg	Semantics
`missionTime()`	0	Returns the current mission time

Priority Rules (Precedence of Operators)

S2ML+DFE (and more generally all languages of the S2ML+X family) obeys the usual precedence rules for operators. Parentheses are used to solve ambiguities.

Priority (decreasing order)
or
and
not
==, !=, <, >, <=, >=
- (n-ary)
+
/
*
- (unary)
all others

`f and g or not f and h`

reads

`(f and g) or ((not f) and h)`

`x>=y+0.0 and x<y+1.0`

reads

`(x>=(y+0.0)) and (x<(y+1.0))`

`3*-x+3/4`

reads

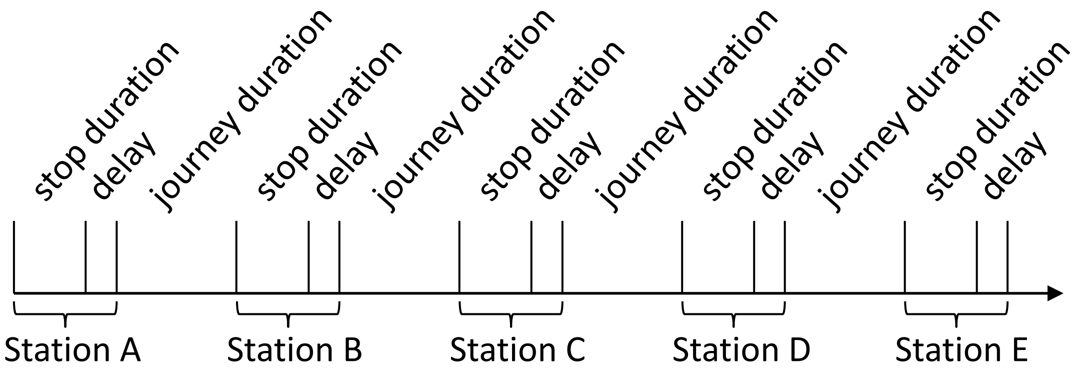
`(3*(-x)) + (3/4)`

S2ML+DFE:
STRUCTURING CONSTRUCTS

Case study: metro line

To present S2ML structuring construct, we shall consider the following case study. 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 whole journey can thus be pictured as:



Blocks in blocks

On way of designing a model for the metro line is to consider stations as sub-models, and then to combine them at line level. E.g.

```
block StationA
   Real arrival, departure, delay;
   parameter Real duration = 40;
   parameter Real low = 0;
   parameter Real high = 20;
   assertion
      delay := uniformDeviate(low, high);
      departure := arrival + duration + delay;
end
```

Blocks in blocks (bis)

The system can then be represented by the following hierarchical model:

```
block MetroLine
   block StationA
      …
   end
   block StationB
      …
   end
   …
   parameter Real AtoBduration = 120;
   …
   parameter Real DtoEduration = 100;
   assertion
      StationA.arrival := 0;
      StationB.arrival := StationA.departure + AtoBduration;
      …
end
```

Instantiated form

The previous hierarchical model is equivalent to the following instantiated model:

```
block MetroLine
   Real StationA.arrival, StationA.departure, StationA.delay;
   parameter Real StationA.duration = 40;
   parameter Real StationA.low = 0;
   parameter Real StationA.high = 20;
   assertion
      StationA.delay := uniformDeviate(StationA.low, StationA.high);
      StationA.departure :=
         StationA.arrival + StationA.duration + StationA.delay;
   …
   parameter Real AtoBduration = 120;
   …
   parameter Real DtoEduration = 100;
   assertion
      StationA.arrival := 0;
      StationB.arrival := StationA.departure + AtoBduration;
      …
end
```

Cloning

Duplicating "by hand" blocks representing similar components would be both tedious and error prone in large systems studies. **Cloning** is a first solution to this problem.

```
block MetroLine
   block StationA
      Real arrival, departure, delay;
      parameter Real duration = 40;
      parameter Real low = 0;
      parameter Real high = 20;
      assertion
         delay := uniformDeviate(low, high);
         departure := arrival + duration + delay;
   end
   clones StationA as StationB;
   …
end
```

Models as scripts

It is possible to change elements of clones in two ways.
Either directly in the clone directive:

```
clones StationA as StationB
   parameter Real high = 30;
   end
```

Or later in the model:

```
clones StationA as StationB;
parameter Real StationB.high = 30;
```

This results of the "**model as script**" concept.

Paths

Thanks to absolute and relative paths, it is possible to refer to any element from any block of hierarchical model.

```
block Network
  block Zone1
    block StationA
      Real departure;
    end
  end
  block Zone2
    block StationB
      Real arrival;
      assertion
        arrival := main.Zone1.StationA.departure + …; // or
        arrival := owner.owner.Zone1.StationA.departure + …;
    end
  end
end
```

Absolute path: the keyword `main` denotes the top level block of the hierarchy.

Relative path: the keyword `owner` denotes the parent block in the hierarchy.

Classes & instances

Another solution consists in creating a on-the-shelf reusable component, via the notion of class. Then to instantiate this component as many times as needed. This makes it possible to create libraries of reusable modeling components.

```
class Station
   Real arrival, departure, delay;
   parameter Real duration = 40;
   parameter Real low = 0;
   parameter Real high = 20;
   assertion
      delay := uniformDeviate(low, high);
      departure := arrival + duration + delay;
end
```

Classes & instances (bis)

Another solution consists in creating a on-the-shelf reusable component, via the notion of class. Then to instantiate this component as many times as needed. This makes it possible to create libraries of reusable modeling components.

```
block MetroLine
   Station A;
   Station B;
   …

   parameter Real AtoBduration = 120;
   …
   parameter Real DtoEduration = 100;
   assertion
      A.arrival := 0;
      B.arrival := A.departure + AtoBduration;
      …
end
```

Models as scripts (bis)

It is possible to change elements of instances in two ways.
Either directly in the instance declaration:

```
Station B
   parameter Real high = 30.0;
   end
```

Or later in the model:

```
Station B;
parameter Real B.high = 30.0;
```

Inheritance

Assume that we have to consider two kinds of stations: simple stations, which only one duration of stops and large stations, with two (or more) stop durations. It is possible to create first a class describing simple stations, then to extends this class for large stations. This mechanism is called **inheritance**.

```
class SimpleStation
   Real arrival, departure, delay;
   parameter Real duration = 40;
   parameter Real low = 0;
   parameter Real high = 20;
   assertion
      delay := uniformDeviate(low, high);
      departure := arrival + duration + delay;
end

class LargeStation
   extends SimpleStation;
   parameter Real high = 40;
end
```

Aggregation

Aggregation denotes a "uses" kind of a relation between blocks (or instances of classes).

```
block MetroLine
    block Controller
        parameter Real regularStopTime = 40;
    end
    block Zone1
        block StationA
            embeds main.Controller as CTRL;
            Real arrival, departure, delay;
            parameter Real duration = CTRL.regularStopTime;
            …
        end
    end
end
```

Aggregation: within the block `StationA`, `CTRL` becomes an alias for `main.Controller`.

JANOS COMMANDS

Role of commands

Categories of commands:

Janos commands can be split into the following categories.

1. Commands to load, to check and to instantiate models.
2. Commands to set parameters of the stochastic simulation and to launch stochastic simulation.
3. Commands to print out information, e.g. models.

Normally, commands of type 1, 2 and 3 are applied in sequence.

Order of arguments:

The order of arguments in a command matters, even though some arguments are optional. You have to follow strictly the syntax described in this section.

Optional arguments:

Optional arguments are given in a special form: `name=value`, where `name` is the name of the argument and `value` its value.

General considerations

One-line commands:

Janos commands spread normally over one line. It is however possible to write a command on several line by escaping the end of line is escaped with an anti-slash "\", e.g.

```
    compute observers Main mission-times=[0, 4380, 8760] \
        output="statistics.txt" mode=append
```

Comments:

Comments can be introduced in Janos scripts. Any character between the character # and the end of the line is a comment. We shall underline comment in green. E.g.

```
# this is a comment
```

File names and modes

File names:

Most of the commands require an input or an output file name as argument. File names can be given directly, e.g. `examples/Pi/Pi.sse` or surrounded with quotes, e.g. `"examples/Pi/Pi.sse"`.

The second form is mandatory when the file name or path includes spaces. It is wise to use it anyway.

Output file modes:

When opening a file to print out something, Janos can do it in two modes: either the file is overwritten if it exists already -- this is the mode `write` --, or the new information is appended at the end of the existing file -- this is the mode `append`. If the file did not exist, it is created in both cases. By default, the mode is `write`. E.g.

```
compute observers Main [8760] "results.txt" mode=append
```

The above command calculates the shortest path from node `A` to node `B` and appends the result to the file `results.txt`.

Result files

Result files are organized in a way they can be loaded into **spreadsheets** (Excel or equivalent). Items are separated with tabs. Methods to load such text files differ from one spreadsheet tool to another.

Model	Main						
Number-of-tries	100000						
Observer	piEstimator						
	Mission-time	0					
		Mean	Standard-deviation	95% confidence range		Minimum	Maximum
		3.14876	1.63719	3.13861	3.15891	0	4

Monte-Carlo Simulation

The command `compute` performs a **Monte-Carlo simulation** when the **number of tries** is positive. Otherwise, it performs a unique simulation. The number of tries is set by the following command.

```
set number-of-tries Integer
```

The simulations involve the generation of numbers at random by means of so-called **pseudo-random number generators**. The one used in Janos is the Mersenne-Twister generator. This generator can be given a starting point, called the **seed**, which is set by means of the following command.

```
set seed Integer
```

Two executions starting with the same seed are strictly the same.

Profiles

Monte-Carlo simulations perform statistics on the values of observers. The statistics to be made are specified via **profiles**. Profiles have names. They can be created, deleted and cloned. E.g.

```
profile new myProfile1
profile clone profile1 myProfile2
profile delete myProfile1
```

By default, the command `compute` is called with the profile named `profile1`. It is possible to call it within another profile by specifying the name of the latter. E.g.

```
compute observers Main mission-times=[0, 4380, 8760] \
    output="statistics.txt" mode=append \
    profile=myProfile2
```

Statistics

Statistics to be performed are defined by means of the command `profile set`.

`profile set mean` *`Identifier Boolean`*

To display the mean of the values of observers.

`profile set standard-deviation` *`Identifier Boolean`*

To display the standard-deviation of the values of observer.

`profile set confidence-range-XX` *`Identifier Boolean`*

To display the XX confidence range of the values of observer. XX must be either 90, 95 or 99.

`profile set extrema` *`Identifier Boolean`*

To display the minimum and the maximum value of the observer.

Statistics (bis)

`profile set quantiles` *`Identifier Boolean`*

To calculate quantiles.

`profile set number-of-quantiles` *`Identifier Integer`*

To set the number of quantiles (should be larger than 0 and smaller than 100).

`profile set distribution` *`Identifier Boolean`*

`profile set cumulative-distribution` *`Identifier Boolean`*

To calculate the distribution and the cumulative distribution of values.

`profile set number-of-points` *`Identifier Integer`*

To set the number of points of the distributions (should be larger than 0 and smaller than 100).

Command to perform stochastic simulations

```
compute observers blockName option*
```

This command launches the stochastic simulation on the given block. The values of variables and observers are calculated at the given mission times and statistics are performed for each observer and each mission time.

Parameters of the stochastic simulation can be either specified using the "set" command (recommended) or by introducing them as options in the command line. E.g. to set the number of tries:

```
set number-of-tries 10000
compute observers Main mission-times=[8760]
```

Commands to print out information

```
print model fileName [mode=(append|write)]
```

This command prints out the original model.

```
print instantiated-model fileName [mode=(append|write)]
```

This command prints out the instantiated model.

```
print environment fileName [mode=(append|write)]
```

This command prints out the environment, i.e. the values of parameters of the stochastic simulation.

REFERENCES

References

Recommend books on stochastic simulation:

Enrico Zio. The Monte Carlo Simulation Method for System Reliability and Risk Analysis. Springer London. London, England. ISBN 978-1-4471-4587-5. 2013.

APPENDIX

GRAMMAR OF S2ML+DFE

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
```

Parameters, variables & 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
```

Equations

```
EquationDeclaration ::=
    assertion Equation*

Equation ::=
    Assignment | DoubleAssignment

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

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

Expressions

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


ConditionalExpression ::=
     if BooleanExpression then Expression else Expression

SpecialBuiltIn ::=
     missionTime()
```

Arithmetic expressions

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

Arithmetic built-in functions & trigonometric functions

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

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

Boolean expressions & 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
```

Random deviates

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

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
```

Identifiers, paths, constants & 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 `*/`.

GRAMMAR OF JANOS COMMAND

Scripts, commands load and instantiate

```
Script ::= Command*

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

CommandLoad ::=
        load model fileName
    |   load script fileName

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

Command set

```
CommandSet ::=
            set seed Integer
     |      set print-mean Boolean
     |      set print-standard-deviation Boolean
     |      set print-confidence-range-90 Boolean
     |      set print-confidence-range-95 Boolean
     |      set print-confidence-range-99 Boolean
     |      set print-extrema Boolean
     |      set print-bins Boolean
     |      set number-of-bins Integer
     |      set number-of-tries Integer
     |      set output String
     |      set mode (write | append)
```

Command compute

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

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

Command print

```
CommandPrint ::=
        CommandPrintModel
    |   CommandPrintInstantiatedModel
    |   CommandPrintEnvironment

CommandPrintModel ::=
    print model fileName [mode=(append|write)]

CommandPrintInstantiatedModel ::=
    print instantiated-model fileName [mode=(append|write)]

CommandPrintEnvironment ::=
    print environment fileName [mode=(append|write)]
```

Comments

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

KNOWN BUGS

Known bugs

Bugs fixed in version 1.2.1:

- Error messages