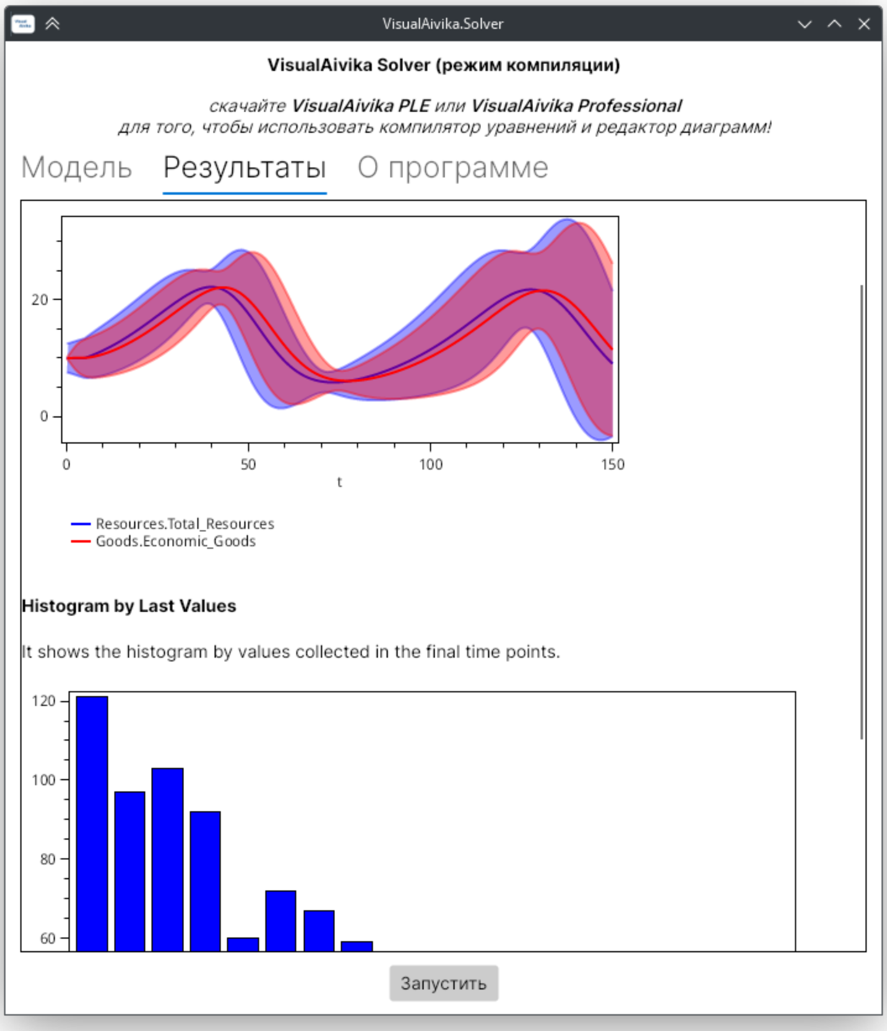
VisualAivika Solver Manual

David E. Sorokin <davsor@mail.ru>,
Yoshkar-Ola, Mari El, Russia

October 16, 2024

Contents


VisualAivika.Solver

VisualAivika Solver (режим компиляции)

*скачайте **VisualAivika PLE** или **VisualAivika Professional***
для того, чтобы использовать компилятор уравнений и редактор диаграмм!

Histogram by Last Values

It shows the histogram by values collected in the final time points.

Figure 1: Displaying simulation results in VisualAivika Solver.

Chapter 1

Getting Started

VisualAivika Solver is a free modeling tool for running models of System Dynamics, which are essentially systems of ordinary differential equations (ODE).

VisualAivika Solver has the simplistic GUI interface, where you define the following model and experiment as demonstrated in figure 1.1.

```
starttime = 0;
stoptime = 10;
dt = 0.01;

ka = 1;
kb = 1;

A = integ(-ka * A, 100);
B = integ(ka * A - kb * B, 0);
C = integ(kb * B, 0);

experiment {
    timeSeries {
        series = [A, B, C];
        width = 500;
        height = 300;
    }

    csvTable {
        series = [time, A, B, C];
        width = 500;
        height = 300;
    }
}
```

These equations correspond to the following mathematical system of equations described in the 5-Minute Tutorial of Berkeley-Madonna[1].

$$\begin{aligned}
\dot{a} &= -ka \times a, \qquad a(t_0) = 100, \\
\dot{b} &= ka \times a - kb \times b, \qquad b(t_0) = 0, \\
\dot{c} &= kb \times b, \qquad c(t_0) = 0, \\
ka &= 1, \\
kb &= 1.
\end{aligned}$$

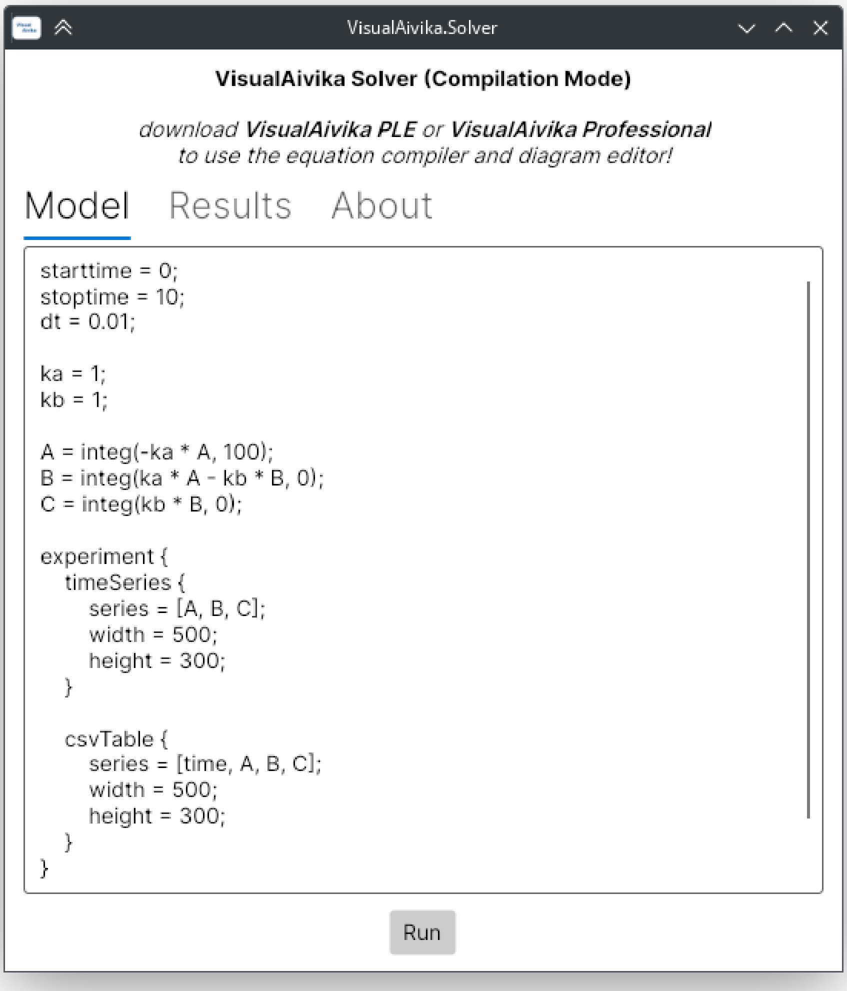
We define the simulations specs that include the predefined variables: `starttime`, `stoptime` and `dt`. They define the start time, the final time and integration time step, respectively.

Then we define equation parameters `ka` and `kb`, after which the three differential equations follow. They use the `integ` function that defines an integral by the specified derivative and initial value.

In the end we define a simulation experiment that contains views. Each view specifies how we want to see the simulation results. Here we define two views. One view is destined for plotting the Time Series chart. Another view is destined for displaying the CSV data that can be then easily copied and exported to another application or saved in the file.

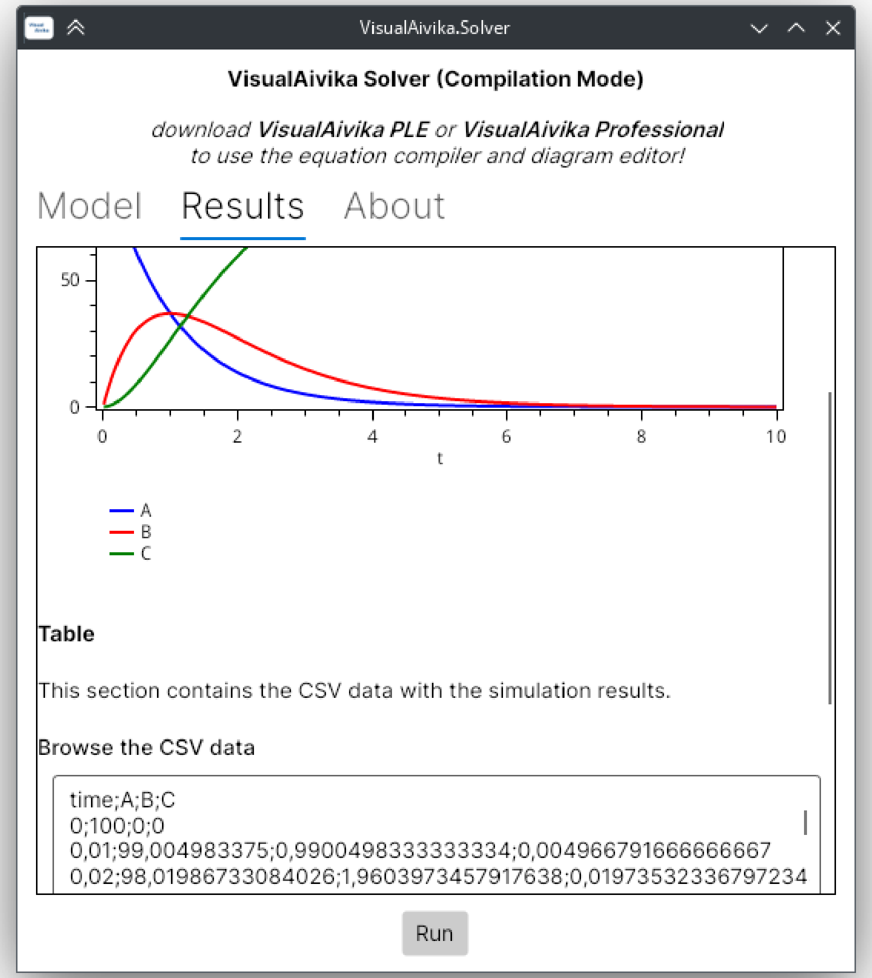
After we click on the *Run* button, the *Results* tab is activated and we can look at the simulation results with help of those views that we defined earlier, when specifying the simulation experiment in the *Model* tab. In case of our model we receive the results as shown in 1.2.

Here we launched one simulation run only, but we can launch multiple simulation runs whithin the same experiment.


VisualAivika.Solver
VisualAivika Solver (Compilation Mode)
download ***VisualAivika PLE*** *or* ***VisualAivika Professional***
to use the equation compiler and diagram editor!
Model
Results
About
Run

```
starttime = 0;
stoptime = 10;
dt = 0.01;

ka = 1;
kb = 1;

A = integ(-ka * A, 100);
B = integ(ka * A - kb * B, 0);
C = integ(kb * B, 0);

experiment {
   timeSeries {
      series = [A, B, C];
      width = 500;
      height = 300;
   }

   csvTable {
      series = [time, A, B, C];
      width = 500;
      height = 300;
   }
}
```

Figure 1.1: The simplistic system of ordinary differential equations.



VisualAivika Solver (Compilation Mode)

*download **VisualAivika PLE** or **VisualAivika Professional***
to use the equation compiler and diagram editor!

Table

This section contains the CSV data with the simulation results.

Browse the CSV data

```
time;A;B;C
0;100;0;0
0,01;99,004983375;0,9900498333333334;0,004966791666666667
0,02;98,01986733084026;1,9603973457917638;0,01973532336797234
```

Figure 1.2: The results that correspond to the system of ordinary differential equations.

Chapter 2

Monte Carlo Experiment

VisualAivika Solver supports the Monte Carlo simulation experiment.

Let us take the same equations from the previous chapter and add other predefined variable `runCount` as shown in figure 2.1.

```
starttime = 0;
stoptime = 10;
dt = 0.01;

runCount = 3;

ka = 1;
kb = 1;

A = integ(-ka * A, 100);
B = integ(ka * A - kb * B, 0);
C = integ(kb * B, 0);

experiment {
    timeSeries {
        series = [A, B, C];
        width = 500;
        height = 300;
    }

    csvTable {
        series = [time, A, B, C];
        width = 500;
        height = 300;
    }
}
```

Here we said that the Monte Carlo experiment should have three simulation runs. Indeed, after we click on the *Run* button, we see that the

Results tab has changed as demonstrated in figure 2.2. Now we have triple charts and triple CSV data, by one for each run.

It is not very useful to have identical charts and CSV data. We can change this by making parameters `ka` and `kb` random that would be updated randomly for each simulation run. They would be constant within the run, but each run might have different values for these parameters.

To add the randomness, let us replace the equations for these parameters:

```
ka = 1 + randomParam(-0.3, 0.3);
kb = 1 + randomParam(-0.2, 0.4);
```

Now `ka` has an uniform distribution in interval [0.7; 1.3], while `kb` is distributed uniformly in interval [0.8, 1.4]. We could define the same intervals directly without using the addition operator.

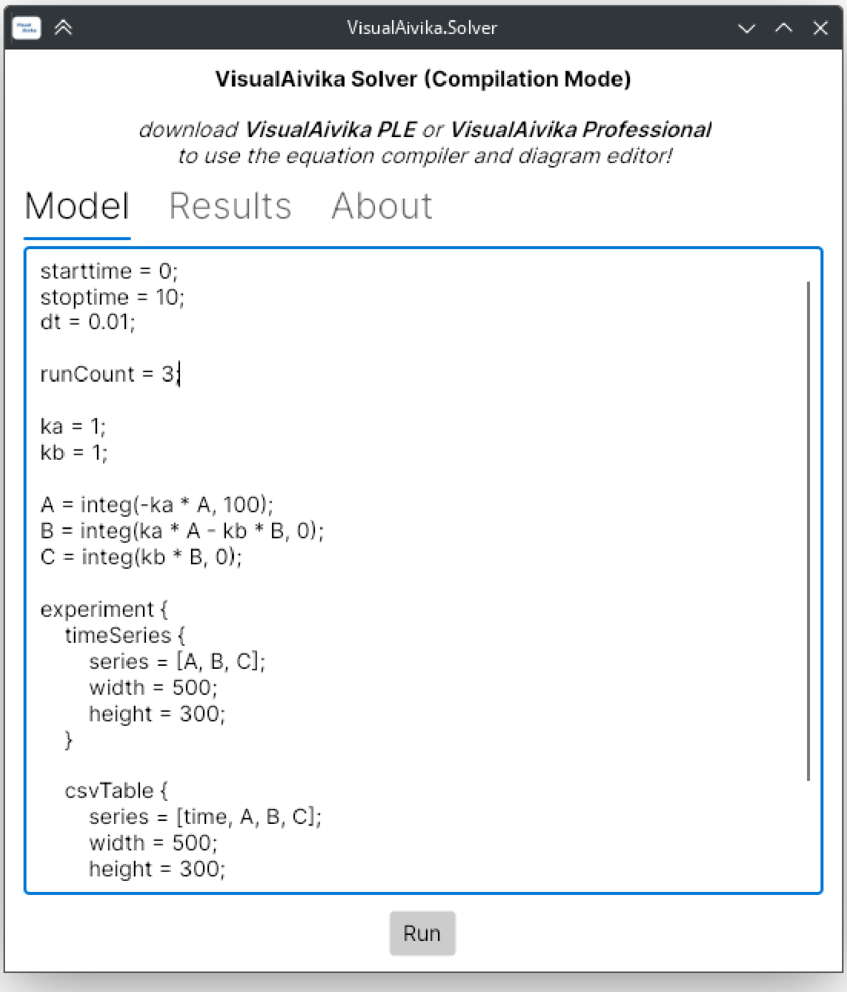
If we re-run the simulation experiment then we can notice that the charts and CSV data will change and be different for each run.

But if we want to start 10000 simulation runs then it is not very productive to use such views that display the Time Series charts and CSV data. Fortunately, there is the Deviation Chart view, which is ideal for the Monte Carlo simulation. The deviation chart plots the trend and confidence intervals by using the 3-sigma rule (based on Chebyshev's inequality).

Let us launch 10000 simulations runs and display the Deviation Chart. For that, we have to update the experiment definition as shown in figure 2.3.

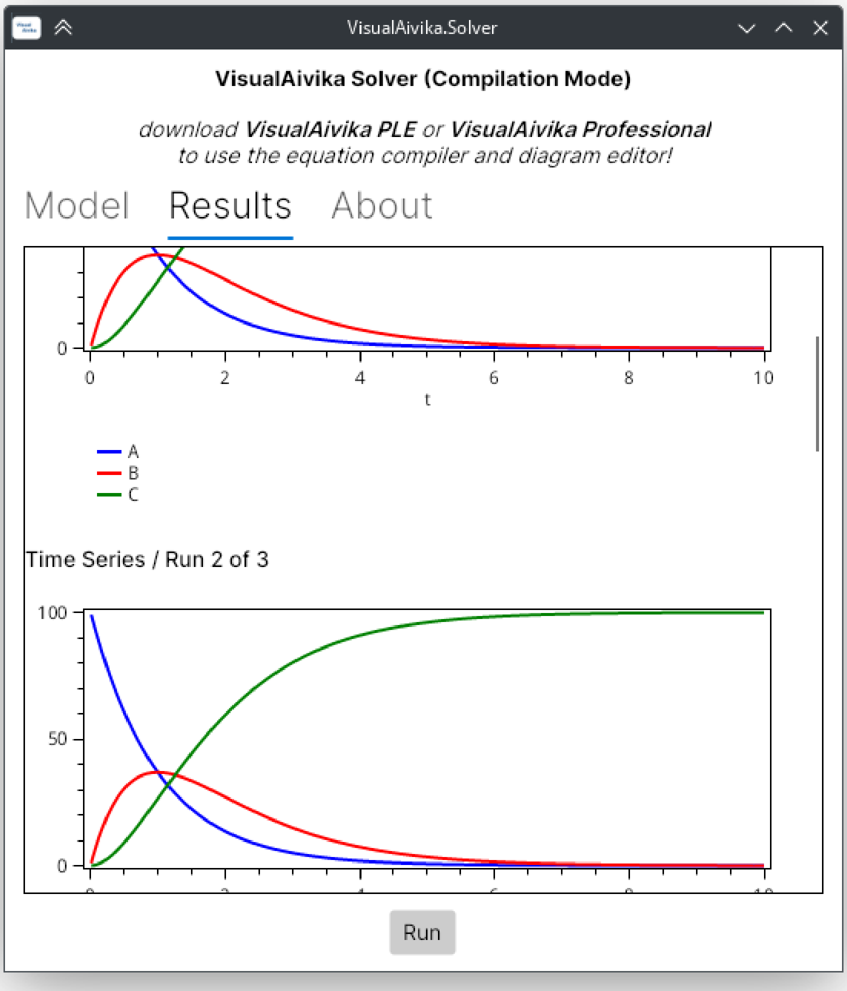
```
starttime = 0;
stoptime = 10;
dt = 0.01;

runCount = 10000;

ka = 1 + randomParam(-0.3, 0.3);
kb = 1 + randomParam(-0.2, 0.4);

A = integ(-ka * A, 100);
B = integ(ka * A - kb * B, 0);
C = integ(kb * B, 0);

experiment {
    deviationChart {
        series = [A, B, C];
        width = 500;
        height = 300;
    }
}
```

The results are displayed in figure 2.4. We can see that this system of equations is quite stable. It converges regardless of the small perturbation of the parameters.


VisualAivika.Solver
VisualAivika Solver (Compilation Mode)
download ***VisualAivika PLE*** *or* ***VisualAivika Professional***
to use the equation compiler and diagram editor!
Model
Results
About
Run

```
starttime = 0;
stoptime = 10;
dt = 0.01;

runCount = 3;

ka = 1;
kb = 1;

A = integ(-ka * A, 100);
B = integ(ka * A - kb * B, 0);
C = integ(kb * B, 0);

experiment {
  timeSeries {
    series = [A, B, C];
    width = 500;
    height = 300;
  }

  csvTable {
    series = [time, A, B, C];
    width = 500;
    height = 300;
```

Figure 2.1: The simplistic Monte Carlo experiment based on the system of ordinary differential equations.


VisualAivika.Solver

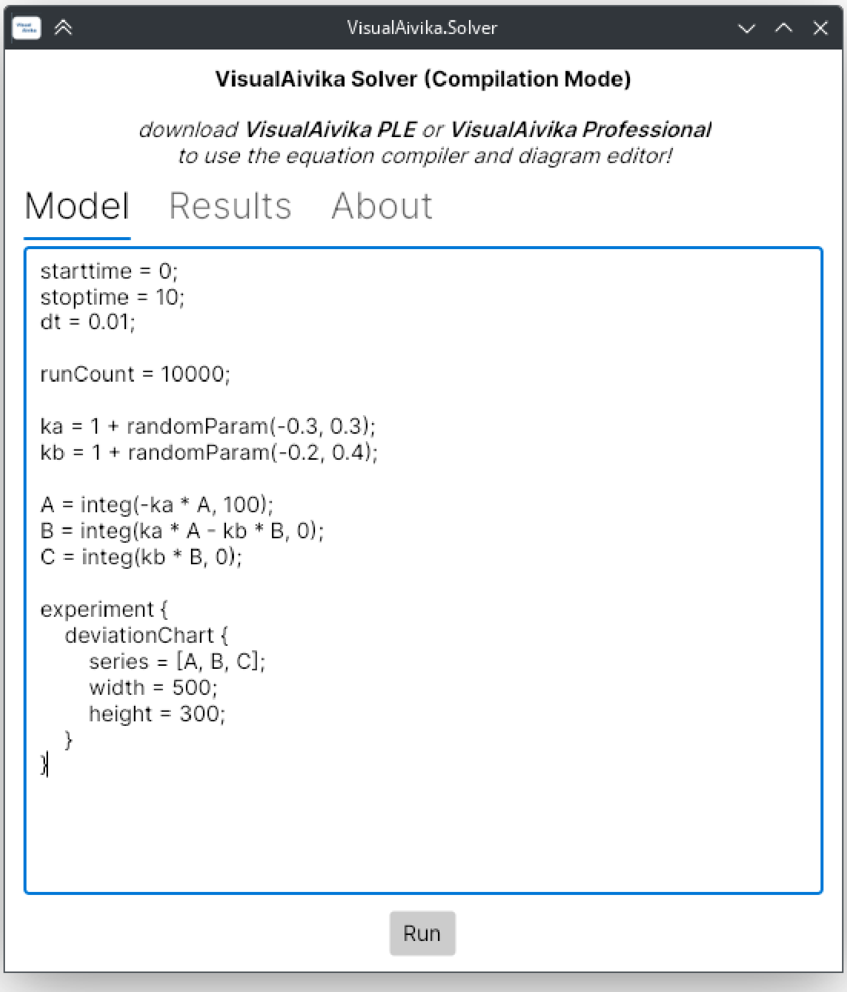
VisualAivika Solver (Compilation Mode)

*download **VisualAivika PLE** or **VisualAivika Professional***
to use the equation compiler and diagram editor!

Model Results About

Time Series / Run 2 of 3

Figure 2.2: The simplistic Monte Carlo experiment results.


VisualAivika.Solver
VisualAivika Solver (Compilation Mode)
download VisualAivika PLE or VisualAivika Professional
to use the equation compiler and diagram editor!
Model
Results
About
Run

```
starttime = 0;
stoptime = 10;
dt = 0.01;

runCount = 10000;

ka = 1 + randomParam(-0.3, 0.3);
kb = 1 + randomParam(-0.2, 0.4);

A = integ(-ka * A, 100);
B = integ(ka * A - kb * B, 0);
C = integ(kb * B, 0);

experiment {
   deviationChart {
      series = [A, B, C];
      width = 500;
      height = 300;
   }
}
```

Figure 2.3: We define the Monte Carlo experiment with 10000 simulation runs.

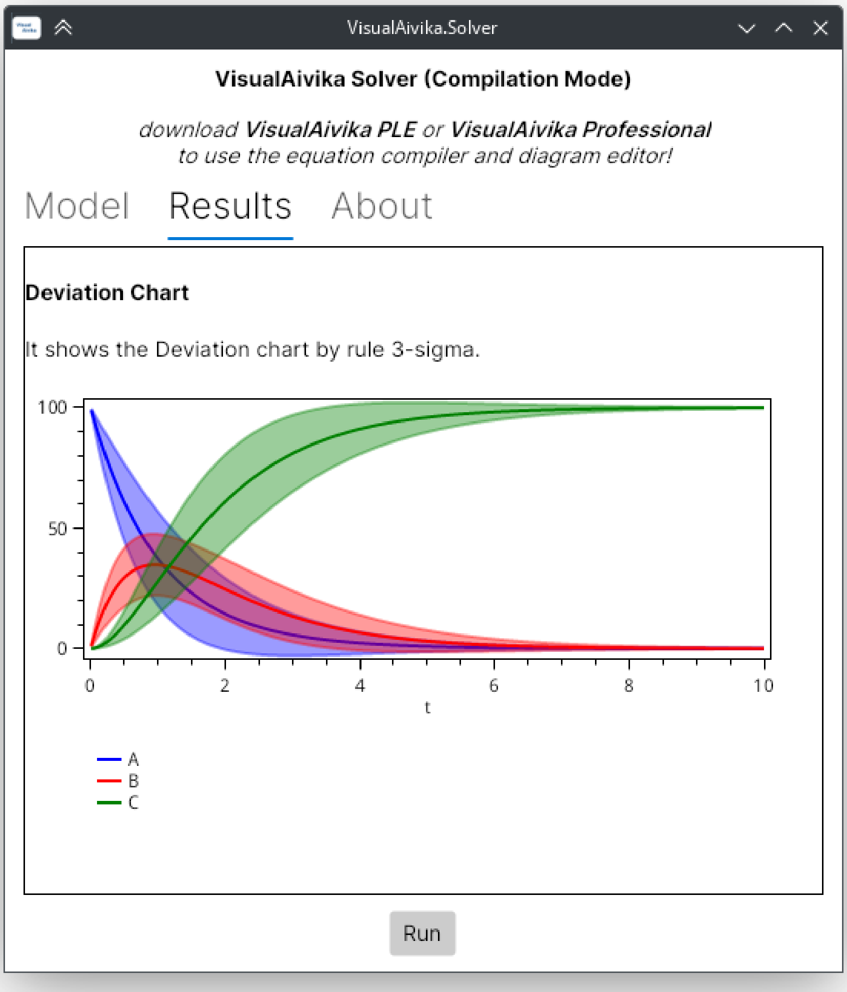

VisualAivika.Solver
VisualAivika Solver (Compilation Mode)
download VisualAivika PLE or VisualAivika Professional
to use the equation compiler and diagram editor!
Model
Results
About
Deviation Chart
It shows the Deviation chart by rule 3-sigma.
100
50
0
0
2
4
6
8
10
t
A
B
C
Run

Figure 2.4: The trends and confidence intervals received with help of the Monte Carlo experiment.

Chapter 3

Modeling Language

The VisualAivika modeling language allows you to specify dynamic systems. The model can consist of stocks and auxiliaries. The stock variable can be an integral (reservoir). The auxiliary variable is a function of other variables.

3.1 Simulation Specs

There are predefined variables that exist in each simulation:

- `starttime` defines the initial time;
- `stoptime` defines the final time;
- `dt` defines the integration time step;
- `time` returns the current integration time.

The first three variables can be defined explicitly for each model.

The current version of VisualAivika Solver supports the 4th order Runge-Kutta method for numerical integration.

In addition, you can specify the number of simulation runs for the Monte-Carlo simulation, which is reflected by the following predefined variables:

- `runIndex` returns the current run index starting from 0;
- `runCount` returns the total number of simulation runs within the experiment.

These two built-in variables are useful for planning the simulation experiment, when providing the Sensitivity Analysis as shown in section 3.9 below.

The `runCount` variable can be defined explicitly.

3.2 Variables

The following list describes the variable types in VisualAivika.

- *Auxiliary*. Any variable that is defined as a function of other variables, or a constant.
- *Stock*. The dynamic variable in the model. At present, it can be a reservoir (integral) only.
- *Flow*. The variable that manages a stock. It can be either an inflow or outflow, bidirectional or unidirectional.
- *Table*. A table function which parameters are the x- and y- coordinates.
- *Range*. An index range for arrays.
- *Array*. An array of other variables.

The variable name in VisualAivika is case sensitive. The first symbol must be a letter or underscore. The next symbols can contain letters, digits and underscore in any sequence.

Example

```
Total_Resources, Scope, x, y
```

3.3 Equations

The VisualAivika modeling language allows you to define the model by writing a set of mathematical equations and expressions. The equations can be written in any order. The order of computation is determined based on dependencies between the variables.

VisualAivika uses standard algebraic expressions with the same rules of precedence as those used by Java, C/C++ and other common languages.

Also VisualAivika supports standard mathematical functions and has additional functions that make writing equations faster and simpler.

Example

```
y = integ (1 + 0.2 * y * sin (t) - 1.5 * t ^ 2, 0);
z = integ ((y - z)^2, 0);
```

Here the `integ` function returns an integral by the specified derivative and initial value.

Example

```
Adequacy_of_Control_Resources =
  Control_Resources / Desired_Control_Resources;
```

Each equation must be finished by the semicolon symbol.

3.4 Operators

VisualAivika supports standard binary and unary operators that observe conventional precedence. The operators are shown in the tables below.

Table 3.1: Unary Operators

`not`	Logical inversion
+ -	Plus and minus

Table 3.2: Binary Operators

ˆ	Arithmetic power
* / + -	Arithmetic operators
< <= > >=	Comparison
== !=	Equality and inequality
`and` `or`	Conjunction and disjunction

3.5 Constants

VisualAivika defines the following constants.

Table 3.3: Built-in Constants

`true`	Logical true

`false`	Logical false
`pi`	The value of π
`infinity`	Represents a positive infinity
`nan`	Represents a value that is not a number

3.6 Functions

Here is a summary table of the basic predefined functions.

Table 3.4: Basic Functions

`abs (x)`	Returns the absolute value of `x`
`sqrt (x)`	Returns the square root of `x`
`int (x)`	Returns the largest whole number less than or equal to `x`
`round (x)`	Returns the number nearest to `x`
`power (x, y)`	Returns the same as x^y
`mod (x, y)`	Returns the remainder of `x/y`
`min (x1, x2, ...)`	Returns the minimum value of `x1`, `x2`, ...
`max (x1, x2, ...)`	Returns the maximum value of `x1`, `x2`, ...
`sum (x1, x2, ...)`	Returns the sum of `x1`, `x2`, ...
`mean (x1, x2, ...)`	Returns the average value of `x1`, `x2`, ...
`sin (x)`	Returns the sine of `x`
`cos (x)`	Returns the cosine of `x`
`tan (x)`	Returns the tangent of `x`
`arcsin (x)`	Returns the inverse sine of `x`
`arccos (x)`	Returns the inverse cosine of `x`
`arctan (x)`	Returns the inverse tangent of `x`
`arctan (y, x)`	Returns the inverse tangent of `(y/x)`
`sinh (x)`	Returns the hyperbolic sine of `x`
`cosh (x)`	Returns the hyperbolic cosine of `x`
`tanh (x)`	Returns the hyperbolic tangent of `x`
`sinwave (a, t)`	Returns the sine wave of amplitude `a` and period `t`
`coswave (a, t)`	Returns the cosine wave of amplitude `a` and period `t`
`exp (x)`	Returns e raised to power `x`

`log (x)`	Returns the natural (base *e*) logarithm of `x`
`log (x, y)`	Returns the logarithm of `x` in base `y`

The `sinwave` and `coswave` functions require some note. They are defined as follows.

Definition

```
sinwave(a, t) = a * sin(2 * pi * time / t);
coswave(a, t) = a * cos(2 * pi * time / t);
```

Table 3.5: Random Number Generators

`random (a, b)`	Returns the uniform random number between `a` and `b`
`normal (m, n)`	Returns the normal random number with mean `m` and deviation `n`
`binomial (p, n)`	Returns the binomial random number on `n` trials of probability `p`
`poisson (m)`	Returns the Poisson random number with mean `m`

Table 3.6: Conditional Expression

`if x then y else z`	Returns `y` if `x` is true, otherwise returns `z`

Table 3.7: Miscellaneous Functions

`step (h, t)`	Returns 0 until time `t` and then returns `h`
`pulse (t, d)`	Returns the pulse of height 1 starting at time `t` with duration `d`
`pulse (t, d, p)`	Returns the pulse of height 1 starting at time `t` with duration `d` and period `p`
`ramp (s, t1, t2)`	Returns 0 until time `t1` and then slopes until time `t2` and then holds `s`

These functions have the following definition. They use the `discrete` operator described further. In short, the operator returns the value, which doesn't change except of the integration dt intervals regardless of the integration method used.

Definition

```
step(h, t) = discrete(if time + dt/2 > t then h else 0);

pulse(t, d) = discrete(if (time + dt/2 > t) and (time + dt/2 < t + d)
                       then 1 else 0);

pulse(t, d, p) = pulse(t + (if (p > 0) and (time > t)
                            then round((time - t) / p)*p else 0), d);

ramp(s, t1, t2) = discrete(if (time + dt/2 > t1)
                           then (if (time - dt/2 < t2)
                                 then s*(time - t1)
                                 else s*(t2 - t1))
                           else 0);
```

Table 3.8: Interpolation

`table ((x1, y1), (x2, y2), ...)`	Creates a table consisting of points `(x1, y1)`, `(x2, y2)`, ..., where the table can be used as a function later
`table ((x1, y1), (x2, y2), ...)  (x)`	Lookup `x` in the list of points `(x1, y1)`, `(x2, y2)`, ... using linear interpolation
`step (t)`	Creates a discrete step-wise interpolation function based on the specified table `t`
`step (table ((x1, y1), (x2, y2), ...))  (x)`	Lookup `x` in the list of points `(x1, y1)`, `(x2, y2)`, ... using the discrete step-wise interpolation

The table functions can be defined directly in equations.

Example

```
Effect_of_Scope_Stability_of_Rules =
  table ((0, 0.5), (0.2, 0.535), (0.4, 0.585), (0.6, 0.675), (0.8, 0.82),
         (1, 1), (1.2, 1.21), (1.4, 1.35), (1.6, 1.42), (1.8, 1.47),
         (2, 1.5))
        (Scope);
```

Table 3.9: Integral Functions

`integ (d, i)`	Returns the integral of rate `d` and initial value `i`
`delay (x, t)`	Returns a first order exponential delay of `x` for time `t` conserving `x`
`delay (x, t, i)`	Returns a first order exponential delay of `x` starting with `i` for time `t` conserving `x`
`delay3 (x, t)`	Returns a third order exponential delay of `x` for time `t` conserving `x`
`delay3 (x, t, i)`	Returns a third order exponential delay of `x` starting with `i` for time `t` conserving `x`
`delayN (x, t, n)`	Returns an `n`'th order exponential delay of `x` for time `t` conserving `x`
`delayN (x, t, n, i)`	Returns an `n`'th order exponential delay of `x` starting with `i` for time `t` conserving `x`
`smooth (x, t)`	Returns a first order exponential smooth of `x` over time `t`
`smooth (x, t, i)`	Returns a first order exponential smooth of `x` over time `t` starting at `i`
`smooth3 (x, t, i)`	Returns a third order exponential smooth of `x` over time `t`
`smooth3 (x, t, i)`	Returns a third order exponential smooth of `x` over time `t` starting at `i`
`smoothN (x, t, n)`	Returns an `n`'th order exponential smooth of `x` over time `t`
`smoothN (x, t, n, i)`	Returns an `n`'th order exponential smooth of `x` over time `t` starting at `i`
`forecast (x, t, h)`	Forecasts for `x` over the time horizon `h` using an average time `t`
`trend (x, t, i)`	Returns the fractional change rate of `x` using the average time `t` and starting with `i`

Here the `delay`-like functions have the following idea, where the `delayN` functions are a generalization of this rule. If you will compare these functions with other simulation software tools, then please note that the `delayN` functions have no effect of the `discrete` operator that may implicitly present in some other tools. In case of need, this operator can be added in the equations manually.

Definition

```
D1=delay(x, t) if and only if
  D1=1/t * integ(x - D1, x*t);

D1I=delay(x, t, i) if and only if
  D1I=1/t * integ(x - D1I, i*t);

D3_3=delay3(x, t) if and only if
  D3_3=1/(t/3) * integ(D3_2 - D3_3, x*t/3);
  D3_2=1/(t/3) * integ(D3_1 - D3_2, x*t/3);
  D3_1=1/(t/3) * integ(x - D3_1, x*t/3);

D3I_3=delay3(x, t, i) if and only if
  D3I_3=1/(t/3) * integ(D3I_2 - D3I_3, i*t/3);
  D3I_2=1/(t/3) * integ(D3I_1 - D3I_2, i*t/3);
  D3I_1=1/(t/3) * integ(x - D3I_1, i*t/3);
```

For example, figure 3.1 demonstrates the first-order and third-order delay functions for the following input and time period.

Example

```
x=sin(time);
t=40*dt;
dt=0.025;
```

The `smooth`-like functions have the next idea, where the `smoothN` functions are a generalization of the following rule. If you will compare these functions with other simulation software tools, then please note that the `smoothN` functions have no effect of the `discrete` operator that may implicitly present in some other tools. In case of need, this operator can be added in the equations manually.

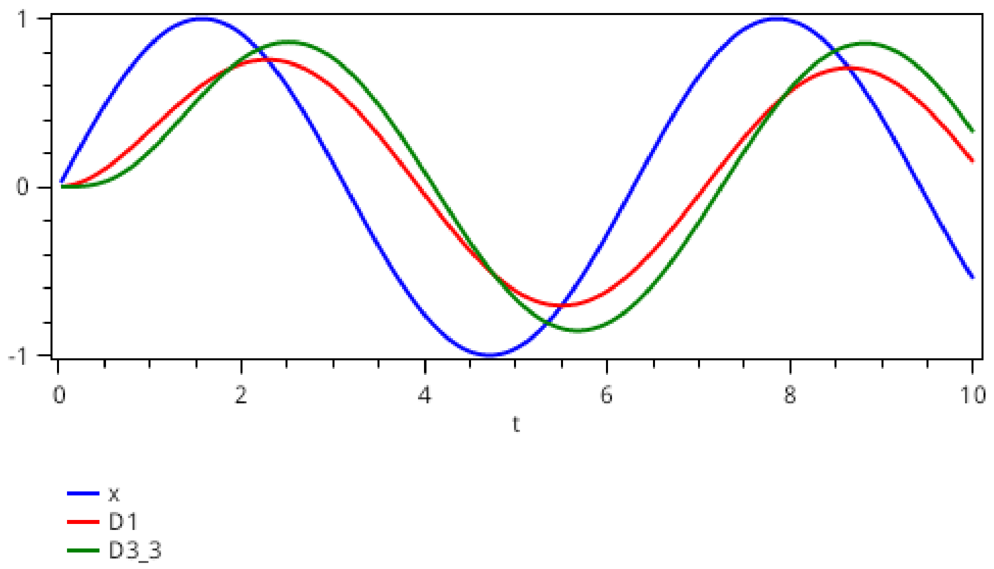


Figure 3.1: The first-order delay, `D1`, and third-order delay function, `D3_3`, for the specified input, `x`, and time period.

Definition

```
S1=smooth(x, t) if and only if
  S1=integ((x - S1)/t, x);

S1I=smooth(x, t, i) if and only if
  S1I=integ((x - S1I)/t, i);

S3_3=smooth3(x, t) if and only if
  S3_3=integ((S3_2 - S3_3)/(t/3), x);
  S3_2=integ((S3_1 - S3_2)/(t/3), x);
  S3_1=integ((x - S3_1)/(t/3), x);

S3I_3=smooth3(x, t, i) if and only if
  S3I_3=integ((S3I_2 - S3I_3)/(t/3), i);
  S3I_2=integ((S3I_1 - S3I_2)/(t/3), i);
  S3I_1=integ((x - S3I_1)/(t/3), i);
```

The `delay` and `smooth` functions behave different when the time period is changing.

The `forecast` and `trend` functions have the following definition, where the `init` operator returns the initial value of the expression at start time of simulation.

Definition

```
forecast(x, t, h) = x * (1 + (x / smooth(x, t) - 1) / t * h);

trend(x, t, i) = (x / smooth(x, t, init(x) / (1 + i * init(t))) - 1) / t;
```

Table 3.10: Financial Functions

`npv (s, r, i, f)`	Returns the Net Present Value (NPV) of stream `s` computed using the specified discount rate `r`, the initial value `i` and some factor `f` (usually 1)
`npve (s, r, i, f)`	Returns the Net Present Value End of period (NPVE) of stream `s` computed using the specified discount rate `r`, the initial value `i` and some factor `f`

In VisualAivika the financial functions have the following definition.

Definition

```
npv=npv(s, r, i, f) if and only if

  npv = (accum + dt * s * df) * f;
  df = integ(- df * r, 1);
  accum = integ(s * df, i);

npve=npve(s, r, i, f) if and only if

  npve = (accum + dt * s * df) * f;
  df = integ(- df * r / beta, 1 / beta);
  accum = integ(s * df, i);
  beta = 1 + r * dt;
```

Table 3.11: Simulation Operators

`discrete (x)`	It returns a value of `x`, which doesn't change except of the integration `dt` intervals regardless of the integration method used
`init (x)`	It returns the initial value of `x`

Unlike other simulation software tools, VisualAivika has slightly nonstandard operators `discrete` and `init`. They are related to the very simulation engine of VisualAivika. Any numeric expression can have the initial value. Also we can treat any numeric expression as something that would be updated as if the Euler's method was applied. Usually, you do not need to apply these operators in your models. Moreover, these two last operators are specific to VisualAivika and they are not transferable between different simulation software tools.

3.7 Ranges

A range is defined by two integer expressions: the low index of the range and the high index of the range. The expressions must be delimited by two dots.

Example

```
N = 10;

I_Range = 1..N;
J_Range = 1..5;
```

The ranges can be defined on like other variables and then be used in the equations. They are needed for arrays.

3.8 Arrays

To define an one-dimensional array, you can use a special syntax:

`[` *Element* `|` *Index* `<-` *Range* `]`

Here the element expression can depend on the index which values will be received from the specified range.

This syntax is generalized for other dimensions too. For example, a two-dimensional array can be defined by adding another index:

`[` *Element* `|` *Index1* `<-` *Range1* `,` *Index2* `<-` *Range2* `]`

VisualAivika supports up to 5 dimensions.

To refer to an element of the one-dimensional array by the specified index, you can use a subscript like this:

Array`[`*Index*`]`

Similarly, the two-dimensional array element can be referenced by two indices:

Array2D[*Index1*, *Index2*]

The rule is generalized for other dimensions too.

Example

```
n = 51;

C = [ if (i == 0) or (i == n + 1) then 0 else M[i] / v | i <- 0..n+1 ];
M = [ integ (q + k*(C[i-1] - C[i]) + k*(C[i + 1] - C[i]), 0) | i <- 1..n ];

q = 1;
k = 2;
v = 0.75;
```

In this example C and M are one-dimensional arrays that have different indices. The C array has values `C[0],...,C[n+1]`, while array M has values `M[1],...,M[n]`.

By the way, the ranges could be defined as separate variables. For simplicity, here the ranges are defined within the arrays. The both methods are acceptable.

3.8.1 Functions for Arrays and Ranges

The following functions are defined for the arrays and ranges.

Table 3.12: Functions for Arrays and Ranges

`length (a)`	Returns the total element count for the specified range or array a
`length (a, d)`	Returns the element count for the specified array a and dimension d starting from 0
`low (a)`	Returns the low index for the specified range or one-dimensional array a
`low (a, d)`	Returns the low index for the specified array a and dimension d starting from 0
`high (a)`	Returns the high index for the specified range or one-dimensional array a
`high (a, d)`	Returns the high index for the specified array a and dimension d starting from 0

`min (a)`	Returns the minimal element of the specified array a
`max (a)`	Returns the maximal element of the specified array a
`sum (a)`	Returns a sum of the elements for the specified array a
`prod (a)`	Returns a product of the elements for the specified array a
`mean (a)`	Returns an average value for the specified array a

The last functions are aggregating. It is useful that we can create intermediate arrays to pass in them to these functions.

The following example is an equivalent to [Vensim 5 Reference Manual, page 30, example 3], from the documentation of Vensim[2].

Example

```
efficiency = prod(factor_efficiency);
US_population = sum(population);
revenue = [ sum([ sales[c, p] * price[p, b] | p <- product ]) |
              c <- country, b <- brand ];
```

Please note how an intermediate array is created, when summing the revenue.

The next example is related to the Theory of Games. It calculates a maximin and minimax by the specified matrix of dimension $n \times n$, respectively.

Example

```
max_min = max([ min([ A[i,j] | j <- 1..n ]) | i <- 1..n ])
min_max = min([ max([ A[i,j] | j <- 1..n ]) | i <- 1..n ])
```

3.8.2 Array Initialization

Some arrays can be defined in a table form without direct using ranges and indices. There is a simplified syntax for that.

Example

```
A1 = [0, 1, 2, 3, 4, 5];
A2 = [[0, 1], [2, 3], [4, 5]];
A3 = [[[0, 1], [2, 3]], [[4, 5], [6, 7]]];
```

Only such arrays will always have indices starting from zero.

3.9 Sensitivity Analysis

As it was mentioned before, there are predefined variables `runIndex` and `runCount` that return the current run index, starting from zero, and the total run count for the Monte-Carlo simulation experiment, respectively.

It is important that the `runIndex` variable is constant within the simulation run, but then it is updated for another run. There are similar random parameters that have the same property. They are constant within the current run, but they are updated for another run.

Table 3.13: Random Parameters

`randomParam (a, b)`	Returns the uniform random parameter between `a` and `b`
`normalParam (m, n)`	Returns the normal random parameter with mean `m` and variance `n`
`binomialParam (p, n)`	Returns the binomial random parameter on `n` trials of probability `p`
`poissonParam (m)`	Returns the Poisson random parameter with mean `m`

Such parameters are useful for providing the Sensitivity analysis. They can be used in the equations.

By redefining some constants as random external parameters, we can test the model for stability. For that, VisualAivika supports the Monte-Carlo simulation to provide the Sensitivity Analysis. The results of this analysis can be displayed on charts like figure 3.2, where the *Deviation Chart* view is used.

Furthermore, combining the array initialization syntax and built-in variables `runIndex` and `runCount`, we can specify sampling for the external parameters.

The idea is quite simple. We define an array with values and then refer to the array by the run index, probably, bound by the array length.

Example

```
A = [[1, 2, 3],
     [2, 3, 1],
     [3, 1, 2]];

Sample = mod(runIndex, length(A, 0));
```

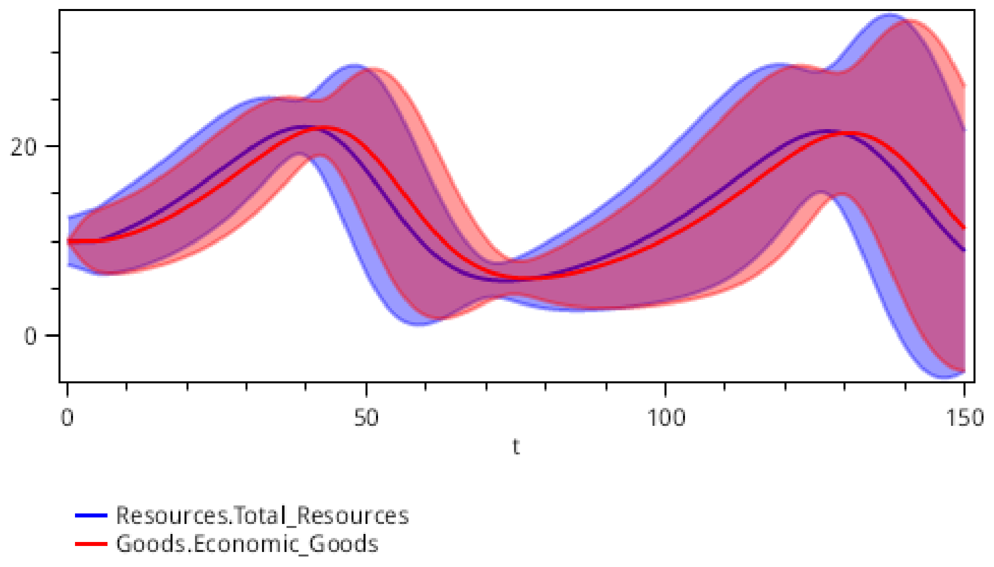


Figure 3.2: The chart shows how stable is the model relative to changes of the external random parameters.

```
    X1 = A[Sample, 0];
    X2 = A[Sample, 1];
    X3 = A[Sample, 2];
```

3.10 Nested Modules

The model can contain modules that can be nested. Each module defines its own name space for variables.

VisualAivika uses modules to keep variables from the same diagram in a separate module.

Example

```
// the global variable
A = 1;

module M1 {

    // variable M1.B
    B = A + 1;
    module M2 {

        // variable M1.M2.C
        C = 2 * B;

        // use the qualified name to M1.B
        D = C - M1.B;
    }
}
```

Chapter 4

Experiment Views

The experiment views allow you to see the simulations results in a preferred way. It can be a chart, or table with CSV data. The view specifies exactly how the results will be represented. The same experiment can contain an arbitrary number of views, enumerated one by one.

4.1 Deviation Chart

The Deviation Chart view is the most universal view, which is applicable both to single and multiple runs. If the run is single, then the Deviation Charts becomes similar to the Time Series chart as described in section 4.2. But if the Monte-Carlo simulation experiment is used, then the Deviation Chart shows the trend and confidence intervals by the 3-sigma rule.

The following table contains the optional properties, which can be used for plotting the Deviation Chart.

Table 4.1: Deviation Chart

`series = [`*series1*`,` *series2*`, ...];`	Enumerates the variable names *series1*, *series2*, . . . , which will be used, when plotting on the chart
`width =` *width*`;`	Specifies the chart width
`height =` *height*`;`	Specifies the chart height

Example

```
experiment {
    deviationChart {
        series = [
            Finance.net_cash_flow,
            Finance.npv_cash_flow
          ];
      }
}
```

The corresponding chart is represented in figure 4.1.

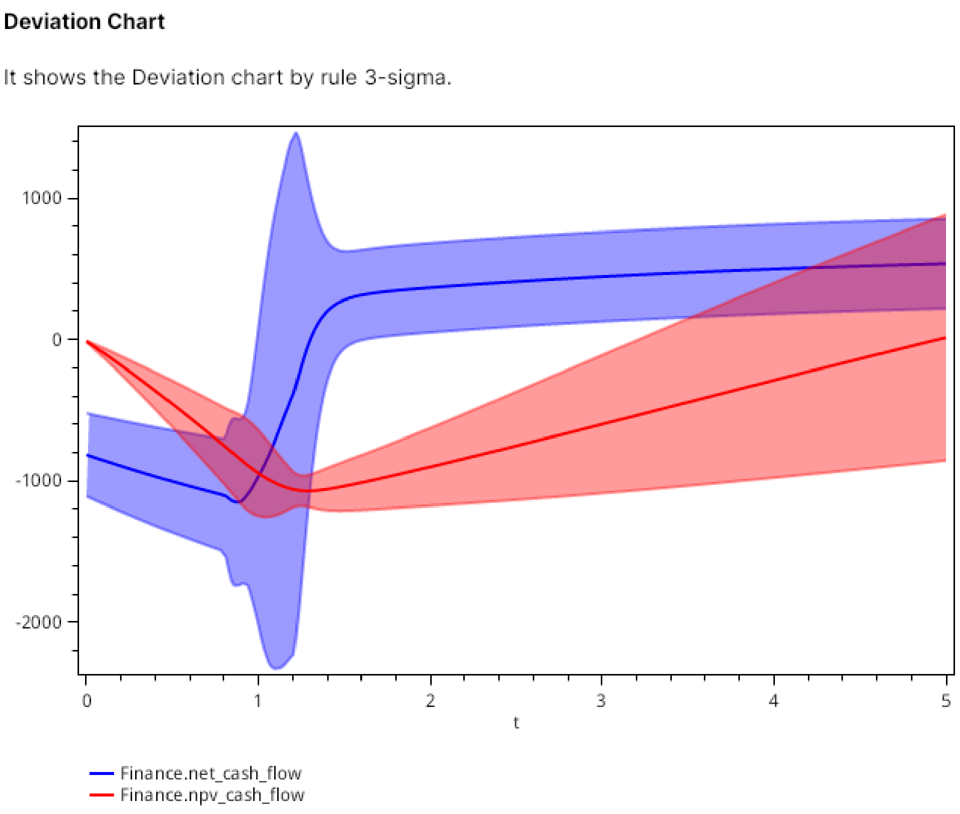


Figure 4.1: The Deviation Chart view representation.

4.2 Time Series

The Time Series chart displays the ordinary chart, where the series depend on the modeling time. If the Monte-Carlo method is used, then the corresponding number of charts will be plotted, by one for each run. Hence, please use the Time Series chart only for single run, or for the Monte-Carlo method with a small number of runs!

The following table contains the optional properties, which can be used for plotting the Time Series.

Table 4.2: Time Series

`series = [`*series1, series2,* `...];`	Enumerates the variable names *series1, series2, . . .*, which will be used, when plotting on the chart

`width =` *width*`;`
`height =` *height*`;`
	Specifies the chart width
	Specifies the chart height

Example

```
x = sin(time);
y = cos(time);

experiment {
    timeSeries {
        series = [x, y];
    }
}
```

The corresponding chart is represented in figure 4.2.

4.3 XY Chart

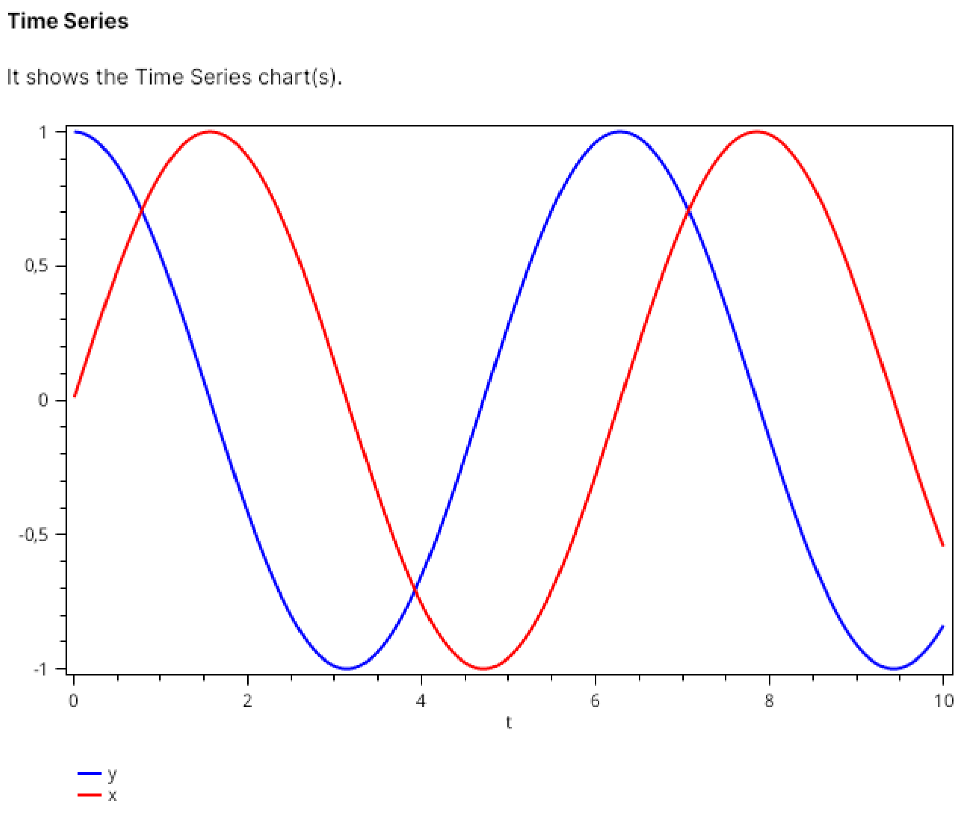
The XY Chart is similar to the Time Series chart described in the previous section 4.2. Only we specify the series that can depend on an arbitrary series. If the Monte-Carlo method is used, then the corresponding number of charts will be plotted, by one for each run. Hence, please use the XY Chart view only for single run, or for the Monte-Carlo method with a small number of runs!

The following table contains the properties, which can be used for plotting the XY Chart. The `seriesX` property is mandatory.

Table 4.3: XY Chart

`seriesX =` *seriesX*`;`	Specifies the series that will be used as X
`seriesY = [`*seriesY1, seriesY2, . . .*`];`	Enumerates the variable names *seriesY1, seriesY2, . . .*, which will be used as Y, when plotting on the chart

`width =` *width*`;`
`height =` *height*`;`
	Specifies the chart width
	Specifies the chart height


Time Series

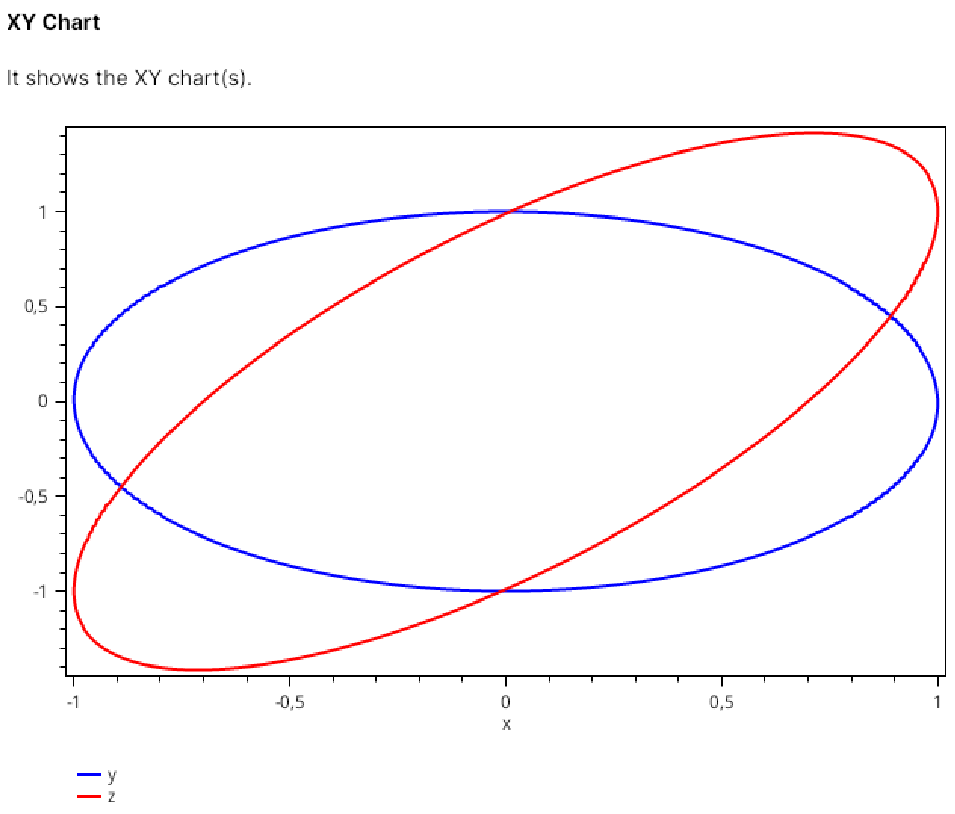
It shows the Time Series chart(s).

Figure 4.2: The Time Series view representation.

Example

```
x = sin(time);
y = cos(time);
z = x + y;

experiment {
    xyChart {
        seriesX = x;
        seriesY = [y, z];
    }
}
```

The corresponding chart is represented in figure 4.3.


XY Chart

It shows the XY chart(s).

Figure 4.3: The XY Chart view representation.

4.4 CSV Table

The CSV Table view is destined for exporting CSV data to other applications, or saving the data in the file. A text block component is displayed, from which you can copy the corresponding CSV data. If the Monte-Carlo method is used, then the corresponding number of components will be created, by one for each run. Hence, please use the CSV Table view only for single run, or for the Monte-Carlo method with a small number of runs!

The following table contains the optional properties, which can be used for displaying the CSV Table.

Table 4.4: CSV Table

`series = [`*series1*`,` *series2*`, ...];`	Enumerates the variable names *series1*, *series2*, . . . , which will be used, when displaying the table

`width =` *width*`;`	Specifies the table component width
`height =` *height*`;`	Specifies the table component height

Example

```
x = sin(time);
y = cos(time);

experiment {
    timeSeries {
        series = [x, y];
    }

    csvTable {
        series = [time, x, y];
    }
}
```

The corresponding table is represented in figure 4.4.

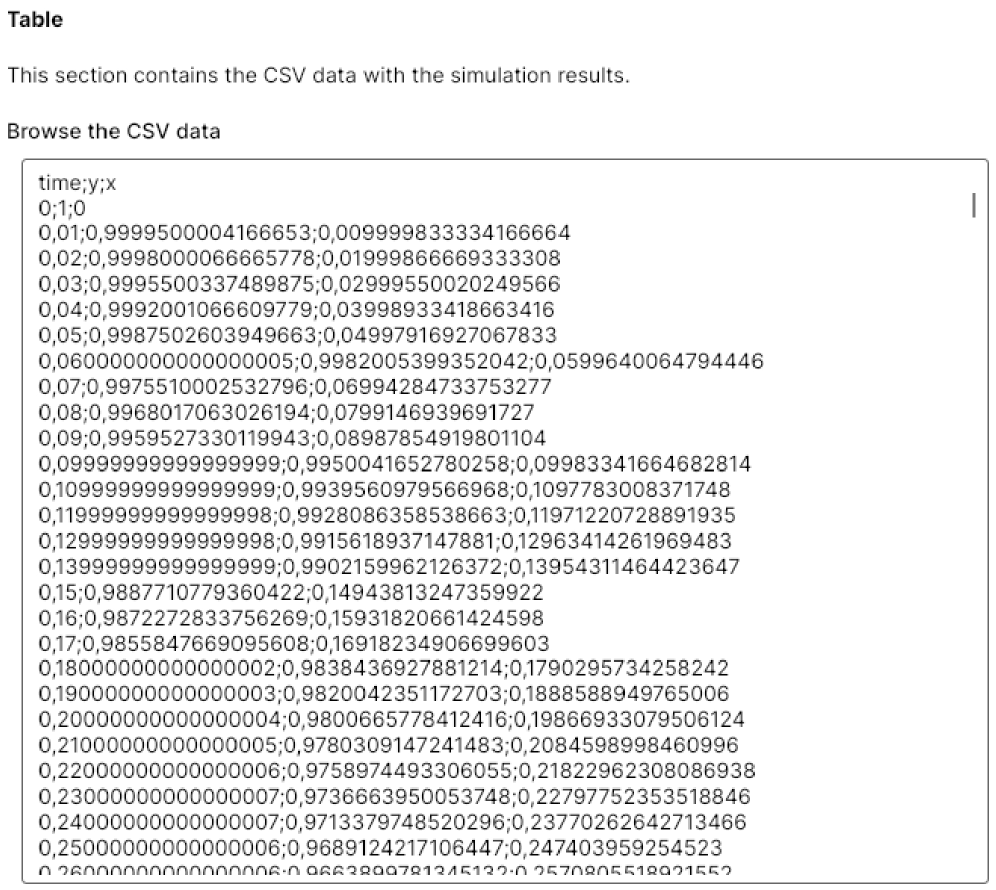
4.5 Last Values

The view of Last Values is destined for displaying the series values at the final time point of simulation. If the Monte-Carlo method is used, then the corresponding number of components will be created, by one for each run. Hence, please use the Last Values only for single run, or for the Monte-Carlo method with a small number of runs!

The following table describes the property, which can be used for displaying the Last Values.

Table 4.5: Last Values

`series = [`*series1*, *series2*, `...];`	Enumerates the variable names *series1*, *series2*, ..., which will be used, when displaying the final values



Table

This section contains the CSV data with the simulation results.

Browse the CSV data

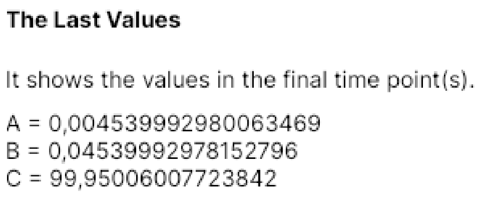
```
time;y;x
0;1;0
0,01;0,9999500004166653;0,009999833334166664
0,02;0,9998000066665778;0,01999866669333308
0,03;0,9995500337489875;0,02999550020249566
0,04;0,9992001066609779;0,03998933418663416
0,05;0,9987502603949663;0,04997916927067833
0,060000000000000005;0,9982005399352042;0,0599640064794446
0,07;0,9975510002532796;0,06994284733753277
0,08;0,9968017063026194;0,0799146939691727
0,09;0,9959527330119943;0,08987854919801104
0,09999999999999999;0,9950041652780258;0,09983341664682814
0,10999999999999999;0,9939560979566968;0,1097783008371748
0,11999999999999998;0,9928086358538663;0,11971220728891935
0,12999999999999998;0,9915618937147881;0,12963414261969483
0,13999999999999999;0,9902159962126372;0,13954311464423647
0,15;0,9887710779360422;0,14943813247359922
0,16;0,9872272833756269;0,15931820661424598
0,17;0,9855847669095608;0,16918234906699603
0,18000000000000002;0,9838436927881214;0,1790295734258242
0,19000000000000003;0,9820042351172703;0,1888588949765006
0,20000000000000004;0,9800665778412416;0,19866933079506124
0,21000000000000005;0,9780309147241483;0,2084598998460996
0,22000000000000006;0,9758974493306055;0,21822962308086938
0,23000000000000007;0,9736663950053748;0,22797752353518846
0,24000000000000007;0,9713379748520296;0,23770262642713466
0,25000000000000006;0,9689124217106447;0,247403959254523
0,26000000000000006;0,9663899781345132;0,2570805518921552
```

Figure 4.4: The CSV Table view representation.

Example

```
dt = 0.01;

ka = 1;
kb = 1;

A = integ(-ka * A, 100);
B = integ(ka * A - kb * B, 0);
C = integ(kb * B, 0);

experiment {
    lastValues {
        series = [A, B, C];
    }

    timeSeries {
        series = [A, B, C];
    }
}
```

The corresponding output is represented in figure 4.5.



The Last Values

It shows the values in the final time point(s).

```
A = 0,004539992980063469
B = 0,04539992978152796
C = 99,95006007723842
```

Figure 4.5: The view representation of Last Values.

4.6 Last Value Histogram

The Last Value Histogram view is destined for plotting the histogram by series values at the final modeling time, when using the Monte-Carlo method with multiple runs.

The following table contains the optional properties, which can be used for plotting the Last Value Histogram.

Table 4.6: Last Value Histogram

`series = [`*series1*`,` *series2*`, ...];`	Enumerates the variable names *series1*, *series2*, ..., which will be used, when plotting the histogram at final time point

`width =` *width*`;`
`height =` *height*`;`
	Specifies the chart width
	Specifies the chart height

Example

```
experiment {
    lastValueHistogram {
        series = [
            Resources.Total_Resources,
            Goods.Economic_Goods
        ];
    }
}
```

The corresponding chart is represented in figure 4.6.

Histogram by Last Values

It shows the histogram by values collected in the final time points.

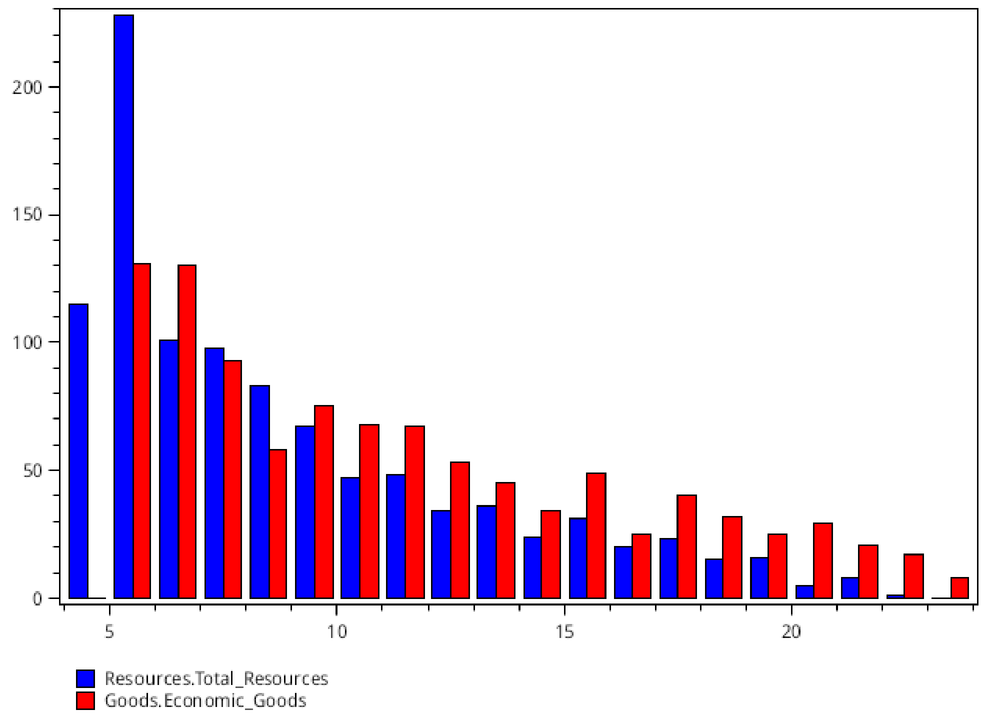


Figure 4.6: The Last Value Histogram view representation.

4.7 Last Value CSV Table

When applying the Monte-Carlo method, the Last Value CSV Table view collects data at final time points, and it is destined for exporting the data to other applications, or saving the data in the file. A text block component is displayed, from which you can copy the corresponding CSV data.

The following table contains the optional properties, which can be used for displaying the Last Value CSV Table.

Table 4.7: Last Value Statistics

`series = [`*series1*`, `*series2*`, ...];`	Enumerates the variable names *series1*, *series2*, ..., which will be used, when collecting data at final time points

`width = `*width*`;`
 	
`height = `*height*`;`
	Specifies the table component width
	Specifies the table component height

Example

```
experiment {
    lastValueCsvTable {
        series = [
            Resources.Total_Resources,
            Goods.Economic_Goods
        ];
    }
}
```

The corresponding table is represented in figure 4.7.

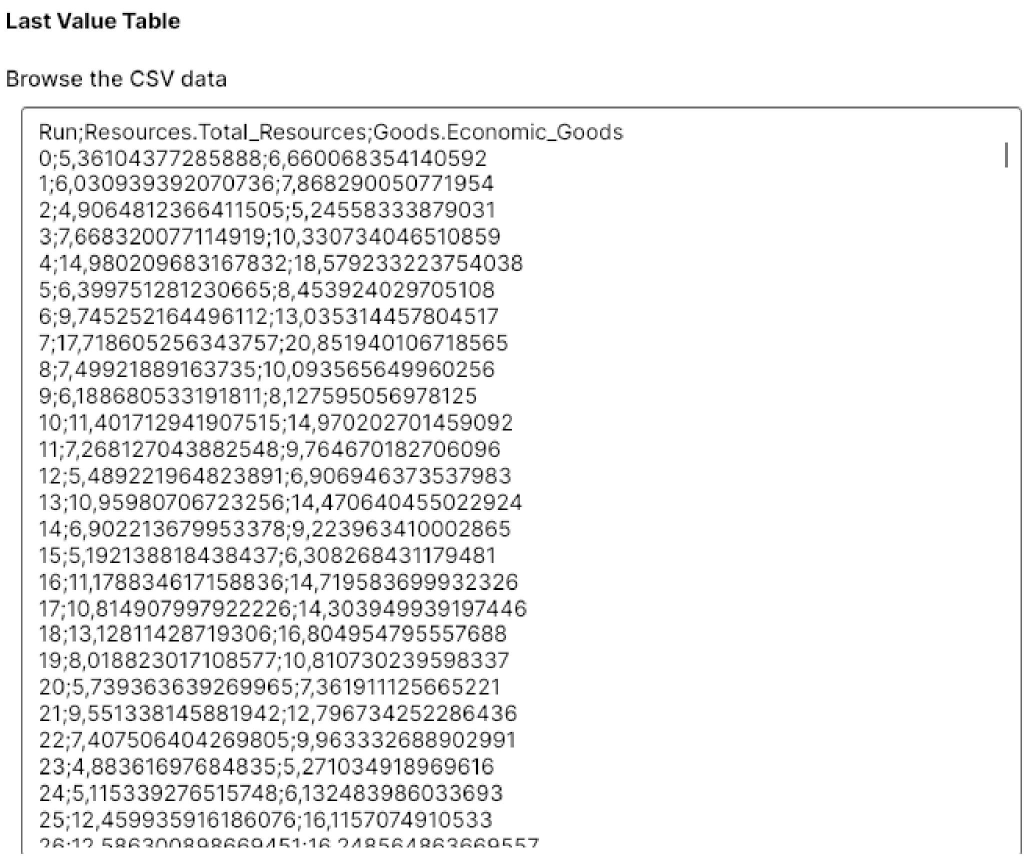
4.8 Last Value Statistics Summary

When applying the Monte-Carlo method, the Last Value Statistics Summary view collects data at final time points and presents the results on the corresponding component.

The following table contains the optional properties, which can be used for displaying the Last Value Statistics Summary.

Table 4.8: Last Value Statistics Summary

`series = [`*series1*`, `*series2*`, ...];`	Enumerates the variable names *series1*, *series2*, ..., which will be used, when collecting data at final time points
`width = `*width*`;`	Specifies the component width



Last Value Table

Browse the CSV data

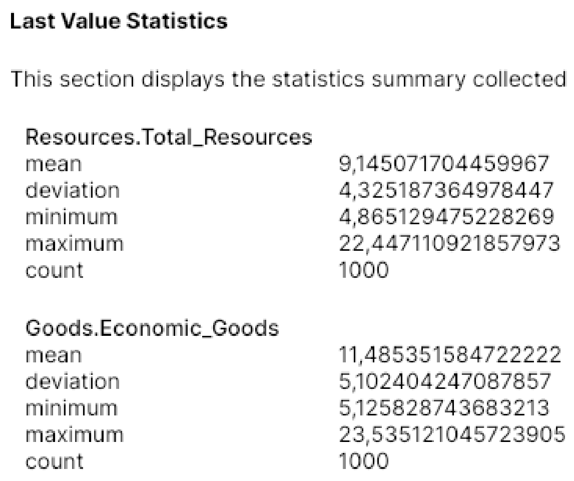
```
Run;Resources.Total_Resources;Goods.Economic_Goods
0;5,36104377285888;6,660068354140592
1;6,030939392070736;7,868290050771954
2;4,9064812366411505;5,24558333879031
3;7,668320077114919;10,330734046510859
4;14,980209683167832;18,579233223754038
5;6,399751281230665;8,453924029705108
6;9,745252164496112;13,035314457804517
7;17,718605256343757;20,851940106718565
8;7,49921889163735;10,093565649960256
9;6,188680533191811;8,127595056978125
10;11,401712941907515;14,970202701459092
11;7,268127043882548;9,764670182706096
12;5,489221964823891;6,906946373537983
13;10,95980706723256;14,470640455022924
14;6,902213679953378;9,223963410002865
15;5,192138818438437;6,308268431179481
16;11,178834617158836;14,719583699932326
17;10,814907997922226;14,303949939197446
18;13,12811428719306;16,804954795557688
19;8,018823017108577;10,810730239598337
20;5,739363639269965;7,361911125665221
21;9,551338145881942;12,796734252286436
22;7,407506404269805;9,963332688902991
23;4,88361697684835;5,271034918969616
24;5,115339276515748;6,132483986033693
25;12,459935916186076;16,1157074910533
26;12,586300898669451;16,248564863669557
```

Figure 4.7: The Last Value CSV Table view representation.

Example

```
experiment {
    lastValueStats {
        series = [
            Resources.Total_Resources,
            Goods.Economic_Goods
        ];
    }
}
```

The corresponding component is represented in figure 4.8.


Last Value Statistics

This section displays the statistics summary collected in final time points.

```
Resources.Total_Resources
mean                      9,145071704459967
deviation                 4,325187364978447
minimum                   4,865129475228269
maximum                   22,447110921857973
count                     1000

Goods.Economic_Goods
mean                      11,485351584722222
deviation                 5,102404247087857
minimum                   5,125828743683213
maximum                   23,535121045723905
count                     1000
```

Figure 4.8: The Last Value Statistics Summary view representation.

Bibliography

[1] `Berkeley Madonna`. `http://www.berkeleymadonna.com`, 2024. Accessed: 20-July-2024.

[2] `Vensim` Software. `http://vensim.com`, 2024. Accessed: 20-July-2024.