StochBB v 1.0 – Stochastic Building Blocks Manual

Hannes Matuschek

Deparment of Mathematics,
Focus area *Dynamics of complex systems*,
University of Potdsam, Germany

Abstract

StochBB is a sofware framework and a graphical user interface program that allows of an efficient analysis of complex systems of dependent random variables. For example, such networks are frequently used to describe cognitive processes and are usually analyzed by means of Monte Carlo type stochastic simulations. Although very flexible, the stochastic simulation approach requires a large number of samples to be drawn, for obtaining reliable statistics of the response variable of interest. In contrast to stochastic simulation, **StochBB** derives the marginal distributions of dependent random variables analytically and resorts to numerical solutions whenever the analytic approach fails. To achieve this, **StochBB** implements a rudimentary computer algebra system for systems of dependent random variables. As an example, I study the properties of a simple response-latency model.

1 Introduction

Frequently, complex systems are described in terms of stochastic processes, as the underlying deterministic process is too complex to be modeled exactly or as the process is indeed random. It is not always the random process itself that is of interest, but a derived quantity. For example, the distribution of waiting times until the process reaches a certain state.

In the field of cognitive psychology, random processes are frequently used to describe each processing stage in a chain of stages that leads to a response. The state of each random process itself is usually not measurable but the total response time of all processing stages involved. Although each processing stage may be modeled as a random process, the waiting-time of a single stage is just a random variable[1]and the complete system is

[1] This requires that the random process is *reset* for each sample. This is usually the case if the waiting time of the processing stage is determined by the time a random process starting at a specified state

therefore a system of dependent random variables (e.g., the EZ-Reader model, Reichle et al., 2003).

StochBB is able to describe and analyze complex systems of dependent random variables by combining simple ones (representing single stages with a known waiting-time distribution) to a complex system. For example, consider the following independent random variables

$$X_1 \sim \Gamma(10, 100)\,, X_2 \sim \Gamma(20, 50) \text{ and } X_3 \sim \mathrm{Exp}(0.01)\,,$$

which are described completely by their distribution. In terms of *processing stages*, this means that the time, the processing stage X_1 needs to complete is gamma-distributed with shape $k = 10$ and scale $\theta = 100$. Analogously, the stages X_2 and X_3 are defined by their own waiting-time distribution.

From these basic building blocks, a more complex system can be assembled by combining them. Using the example above, one may assume sequential processing. That is, a chain of the stages $X_1, \dots, X_3$. This simple chain then describes the successive processing of information entering the first stage described by X_1. Once the first processing stage finishes, its result gets forwarded to the second stage described by X_2 and finally to the last stage described by X_3. The waiting time of the complete chain is again a random variable that is the sum of all random variables, or expressed mathematically

$$Y = X_1 + X_2 + X_3\,.$$

SochBB determines the probability density function (PDF) or cumulative probability function (CDF) of the waiting-time distribution of Y analytically (as far as possible) or resorts to a numeric method if the analytic approach fails (see Thomas et al., 2012, 2013, for examples of numerical approximations outperforming the stochastic simulation approach). Moreover it provides an efficient and correct sampler for the system of random variables.

In the following section, I outline the representation of random variables and the reductions performed on a network. In section 3, I describe the architecture of the framework followed by a in-detail description of the graphical user interface (GUI) application that allows to assemble and analyze networks of dependent random variables easily. In section 5, I showcase the usage of the GUI application for an analysis of a simple response latency model. Finally, in section 6, I describe how parameters of the distributions of random variables can be estimated from data.

reaches a certain end state. Then, the waiting times of the processing stage are independent samples from a simple random variable with some distribution.

2 Representation and reductions of random variables

Continuing the example above, please note that the sum of random variables commutes. Hence the random variable Y remains the same if defined as $Y = X_1 + X_3 + X_2$ instead of $Y = X_1 + X_2 + X_3$. Moreover, the distribution of the sum of X_1 and X_3 can be determined analytically as $X' = (X_1 + X_3) \sim \Gamma(11, 100)$. Hence the random variable Y can now be expressed as $Y = X_2 + X'$, and only a single numeric convolution is necessary to obtain the PDF of the random variable Y. **SochBB** implements several reductions, exploiting mathematical identities of random variables and their distributions. To this end, it allows to obtain their PDFs and CDFs efficiently. In this section, all basic building blocks that are currently implemented and their reductions are presented and discussed in some detail.

2.1 Affine transformations of random variables

An affine transformation of the random variable X has the form $Y = a\,X + b$, where $a \neq 0$ and b are real values. Although affine transformations of random variables are not frequently used directly in a system of random variables, they may appear as a result of other reductions of the system. The PDF and CDF of the random variable Y defined above, are then

$$f_Y(y) = \frac{1}{a} f_X\left(\frac{y-b}{a}\right) \text{ and } F_Y(y) = F_X\left(\frac{y-b}{a}\right) .$$

Of course, an affine transformation of an affine transformed random variable X is also a simple affine transformation of the random variable. Hence the following reduction is implemented

$$c\,(a\,X + b) + d \longrightarrow (a\,c)\,X + (c\,b + d) .$$

2.2 Sums of random variables

Sums of random variables have been introduced briefly above and may represent a chain of processing stages being triggered sequentially. The sum itself is a derived random variable that depends on all mutually independent variables being summed up. The PDF of the sum Y is then the convolution of all PDFs of the summed variables. That is

$$Y = \sum_{i=0}^{N} X_i \text{ where } X_i \sim f_i(x) \text{ mutually independent}$$
$$Y \sim f_1(x) * \cdots * f_N(x) .$$

The direct numerical convolution of the underlying distributions can be computationally expensive if the number of PDFs is large. Assuming that all distributions are well supported on a common interval, however, allows for a fast numerical convolution by means of FFT convolution (e.g., Press et al., 2007). This requires that the grid is chosen such that all densities being convoluted as well as the result are well supported on the chosen interval and the grid must be fine enough to capture the details of all distributions.

Like any numerical approach, the FFT convolution is only an approximation. Hence **SochBB** tries to perform convolutions analytically before resorting to the numerical approach.

First, all sums of random variables are flattened. That is,

$$\begin{array}{l} Y_1 = X_1 + X_2 \\ Y_2 = Y_1 + X_3 \end{array} \longrightarrow \begin{array}{l} Y_1 = X_1 + X_2 \\ Y_2 = X_1 + X_2 + X_3 \end{array} ,$$

and all common terms are collected

$$X_1 + X_2 + X_1 \longrightarrow 2\,X_1 + X_2\,.$$

Then, the distribution of the sum is derived. Here the following reductions are performed.

$$\begin{aligned} \delta(x-x_0) * f(x) &\longrightarrow f(x-x_0)\,, \\ \phi(x;\mu_1,\sigma_1) * \phi(x;\mu_2,\sigma_2) &\longrightarrow \phi(x;\mu_1+\mu_2,\sqrt{\sigma_1^2+\sigma_2^2})\,, \\ \Gamma(x;k_1,\theta) * \Gamma(x;k_2,\theta) &\longrightarrow \Gamma(x;k_1+k_2,\theta)\,, \end{aligned}$$

where $\delta(\cdot - x_0)$ is the delta distribution located at x_0, $\phi(\cdot;\mu,\sigma)$ the normal distribution with mean μ and standard deviation σ and $\Gamma(\cdot;k,\theta)$ the gamma distribution with shape k and scale θ.

2.3 Minimum and maximum of random variables

The minimum or maximum of some given random variables, e.g. $Y = \min\{X_1, X_2\}$ or $Y = \max\{X_1, X_2\}$ are themselves random variables. These derived random variables may express the waiting time of a processing stage that consists of several independent processing stages being performed in parallel (in contrast to sequential processing described by sums of random variables). If the complete system finishes once the first of the parallel processing stages finishes, the resulting waiting time is the minimum of the underlying random variables. If the system finishes once all underlying processing stages are finished, the resulting waiting time is the maximum.

If the two independent random variables X_1 and X_2 are distributed according to the (cumulative) distribution functions $F_1(x)$ and $F_2(x)$, the maximum of these two random variables $Y = \max\{X_1, X_2\}$ is distributed according to the probability function $F_Y(y) = F_1(y)\cdot F_2(y)$. Consequently, its PDF is $f_Y(y) = f_1(y)\cdot F_2(y) + f_2(y)\cdot F_1(y)$, where $f_1(\cdot)$

and $f_2(\cdot)$ are the PDFs of the random variables X_1 and X_2. Likewise, the CDF and PDF of the minimum can be expressed as $F_Y(y) = 1 - (1 - F_1(y)) \cdot (1 - F_2(y))$ and therefore $f_Y(y) = f_1(y)(1 - F_2(y)) + f_2(y)(1 - F_1(y))$.

For applying these equations to derive the CDFs and PDFs of the minimum and maximum random variables, the underlying random variables must be independent. To achieve independence where possible, the following reductions are performed first. In a first step, the maximum and minimum structures were flattened, for example

$$\begin{array}{l} Y_1 = \max\{X_1, X_2\} \\ Y_2 = \max\{Y_1, X_3\} \end{array} \longrightarrow \begin{array}{l} Y_1 = \max\{X_1, X_2\} \\ Y_2 = \max\{X_1, X_2, X_3\} \end{array},$$

then, possible common terms are collected like

$$\max\{X_1 + X_2, X_3 + X_2\} \longrightarrow \max\{X_1, X_3\} + X_2 \,.$$

The latter transformation does not *ensure* independence of random variables, but decreases the complexity of setting-up a complex system of random variables by resolving simple-structured dependencies between random variables analytically.

2.4 Mixture of random variables

A mixture is the weighted sum of random variables

$$Y = \frac{w_1 X_1 + \cdots + w_N X_N}{w_1 + \cdots + w_N} \,,$$

where w_i are positive weights. The result Y is also a random variable with the PDF

$$f_Y(y) = \frac{w_1 f_1(x) + \cdots + w_N f_N(x)}{w_1 + \cdots + w_N} \,,$$

where $f_i(\cdot)$ is the PDF of the i-th random variable X_i. The CDF of the mixture is obtained analogously. A mixture can be used to describe a random path-selection in a system of processing stages.

2.5 Conditional random variables

The *conditional* random variable selects one of two random variables (e.g. Y_1 or Y_2) depending on the condition $X_1 < X_2$. That is

$$Z = \begin{cases} Y_1 & \text{if } X_1 < X_2 \\ Y_2 & \text{else} \,, \end{cases}$$

where X_1, X_2, Y_1 and X_1, X_2, Y_2 are mutually independent random variables. The variables Y_1 and Y_2 do not need to be mutually independent. Likewise for the *maximum* and *minimum* of random variable introduced above, common terms of random variables are removed first. That is

$$Z = \begin{cases} Y_1 + C_Y & \text{if } X_1 + C_X < X_2 + C_X \\ Y_2 + C_Y & \text{else}\,, \end{cases} \longrightarrow Z = C_Y + \begin{cases} Y_1 & \text{if } X_1 < X_2 \\ Y_2 & \text{else}\,, \end{cases} .$$

Given that the possible result variables Y_1 and Y_2 are independent from the condition, the result variable is then a simple mixture where the weight is given by the probability of $X_1 < X_2$. Hence the density of the result variable Z is

$$\begin{aligned} f_Z(z) &= Pr[X_1 < X_2]\, f_{Y_1}(z) + (1 - Pr[X_1 < X_2])\; f_{Y_2}(z) \\ &= f_{Y_1}(z) \int_{-\infty}^{\infty} F_{X_1}(x)\, f_{X_2}(x)\, dx + f_{Y_2}(z) \int_{-\infty}^{\infty} F_{X_2}(x)\, f_{X_1}(x)\, dx\,. \end{aligned}$$

2.6 Conditional sum of random variables

The only non-textbook example of a derived random variable, is the conditional sum of random variables. It can be defined as

$$Z = \begin{cases} X_1 + Y_1 & \text{if } X_1 < X_2 \\ X_2 + Y_2 & \text{else}\,, \end{cases}$$

where X_1, X_2, Y_1 and X_1, X_2, Y_2 are mutually independent. Y_1 and Y_2 may be dependent random variables. Although being similar to the conditional random variable, there is an important difference: Both possible outcomes ($X_1 + Y_1$ and $X_2 + Y_2$) are not independent from the condition ($X_1 + Y_1$ depends trivially on X_1). Therefore, the simple conditional random variable cannot be used here. However, like with the conditional random variable, common terms are removed first. That is

$$Z = \begin{cases} X_1 + Y_1 + C & \text{if } X_1 + C < X_2 + C \\ X_2 + Y_2 + C & \text{else} \end{cases} \longrightarrow Z = C + \begin{cases} X_1 + Y_1 & \text{if } X_1 < X_2 \\ X_2 + Y_2 & \text{else}\,. \end{cases}$$

The conditional sum of random variables can be used to describe two independent parallel processing stages where the fastest stage will trigger another stage. For example, if X_1 *wins*, it triggers Y_1 and if X_2 *wins*, it triggers Y_2. In contrast to the conditional sum, the simple conditional random variable does not trigger a third stage but *selects* the value of a third stage. Therefore, the actual waiting time of the *winning* stage does not have an influence on the selected one and may lead to non-causal waiting times if X_1 and X_2 are larger than Y_1 and Y_2 (i.e., the response is faster than the processes selecting the response). The conditional sum of random variables always maintains causality.

The conditional sum is certainly not common and to my knowledge not covered in text books. Hence the density for it must be obtained first. Given that the cases ($X_1 < X_2$ and $X_2 < X_1$) are mutually exclusive, the density of Z, $f_Z(z)$ can be expressed as a simple sum. Precisely

$$f_Z(z) = \iiint_{-\infty}^{\infty} f(x_1, x_2, y_1 | z = x_1 + y_1, x_1 < x_2)\, dx_1\, dx_2\, dy_1 \\ + \iiint_{-\infty}^{\infty} f(x_1, x_2, y_2 | z = x_2 + y_2, x_2 < x_1)\, dx_1\, dx_2\, dy_2\,.$$

Given that X_1, X_2 and Y_1 are mutually independent, the first integral can be reduced to

$$\begin{aligned}
&\iiint_{-\infty}^{\infty} f(x_1, x_2, y_1 | z = x_1 + y_1, x_1 < x_2)\, dx_1\, dx_2\, dy_1 \\
&\quad = \iiint_{-\infty}^{\infty} f_{X_1}(x_1)\, f_{X_2}(x_2)\, f_{Y_1}(y_1)\, \delta(z - x_1 - y_1)\, H(x_2 - x_1)\, dx_1\, dx_2\, dy_1 \\
&\quad = \iint_{-\infty}^{\infty} f_{X_1}(x_1)\, f_{X_2}(x_2)\, f_{Y_1}(z - x_1)\, H(x_2 - x_1)\, dx_1\, dx_2 \\
&\quad = \int_{-\infty}^{\infty} f_{X_1}(x_1)\, f_{Y_1}(z - x_1) \int_{-\infty}^{\infty} f_{X_2}(x_2)\, H(x_2 - x_1)\, dx_2\, dx_1 \\
&\quad = \int_{-\infty}^{\infty} f_{X_1}(x_1)\, f_{Y_1}(z - x_1) \int_{x_1}^{\infty} f_{X_2}(x_2)\, dx_2\, dx_1 \\
&\quad = \int_{-\infty}^{\infty} f_{X_1}(x_1)\, \left(1 - F_{X_2}(x_1)\right)\, f_{Y_1}(z - x_1)\, dx_1\,.
\end{aligned}$$

Analogously, the second integral can be reduced to

$$\iiint_{-\infty}^{\infty} f(z, x_1, x_2, y_2 | z = x_2 + y_2, x_2 < x_1)\, dx_1\, dx_2\, dy_2 \\ = \int_{-\infty}^{\infty} f_{X_2}(x_2)\, \left(1 - F_{X_1}(x_2)\right)\, f_{Y_2}(z - x_2)\, dx_2\,.$$

The final density of Z is then given by

$$\begin{aligned}
f_Z(z) &= \int_{-\infty}^{\infty} f_{X_1}(x_1)\, \left(1 - F_{X_2}(x_1)\right)\, f_{Y_1}(z - x_1)\, dx_1 \\
&\quad + \int_{-\infty}^{\infty} f_{X_2}(x_2)\, \left(1 - F_{X_1}(x_2)\right)\, f_{Y_2}(z - x_2)\, dx_2\,, \\
&= \left[f_{X_1}\, (1 - F_{X_2})\right] * f_{Y_1} + \left[f_{X_2}\, (1 - F_{X_1})\right] * f_{Y_2}\,,
\end{aligned}$$

and it can be evaluated using the same *trick* used for chains of random variables.

2.7 Compound random variables

The most complex derived random-variable type is the compound random variable. That is a random variable X, distributed according to a parametric distribution $X \sim f_{X|A}(x; A)$ with parameter A. A, however, is itself a random variable with its own distribution $A \sim g(a)$. The distribution of the compound random variable X is then obtained by marginalizing the parameter of the PDF $f_{X|A}(\cdot; a)$ as

$$f_X(x) = \int f_{X|A}(x; a)\, g(a)\, da\,,$$

and the CDF is obtained analogously as

$$F_X(x) = \int F_{X|A}(x; a)\, g(a)\, da\,.$$

SochBB determines the PDF and CDF of X by performing the following reductions

$$\begin{aligned} &X \sim \delta(x - Y), Y \sim f(y) \longrightarrow X \sim f(x)\,,\\ &X \sim \mathcal{N}(Y, \sigma^2), Y \sim f(y) \longrightarrow X \sim \mathcal{N}(0, \sigma^2) * f(y)\,, \end{aligned}$$

or solves the integral numerically.

3 Software architecture

In this section, I briefly describe the software architecture of the complete **StochBB** framework as shown in Fig. 1.

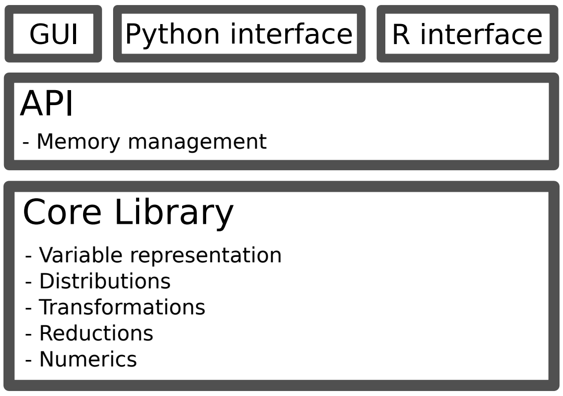


Fig. 1: Overview of the **StochBB** software architecture.

The foundation of the complete framework is formed by the **StochBB** core library. This C++ library implements the core functionality of the system. Prominent elements are classes representing atomic random-variables, parametric distributions as well as derived variables (e.g., sums of random variables) and their distributions (e.g., convolution densities). Together, these classes form the representation of a complex system of dependent random variables. Analytic solutions for some marginal distributions are obtained by reduction and transformation operations performed on the system representation. These operations are also part of the core library. Finally a set of numerical algorithms (e.g., numerical convolution and integrals) where implemented for obtaining numerical approximations of densities which cannot be derived analytically.

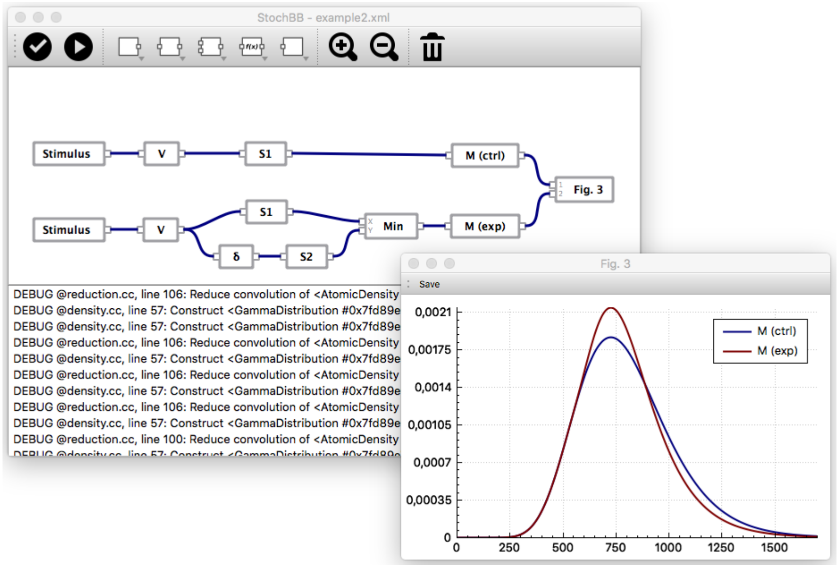
Using the core library directly would rather be complicated as C++ does not provide any means of automatic memory management. For a convenient usage of the core library an application programming interface (API) is provided for the core library that encapsulates all objects of the core library into container classes. These container classes then implement an automatic memory management by means of a *mark-and-sweep* garbage collector (e.g., Aho et al., 2007). This API (outlined in Appendix A) is then used to provide interfaces to the Python and R programming language as well as implementing a graphical user interface (GUI) application described in Section 4 below.

4 Graphical user interface application

Within this section, I briefly overview the graphical user interface (GUI) application that allows to assemble and analyze complex networks of random variables graphically, without the need to specify such networks declaratory using the R, Python or even the C++ API.

While the semantic of the **StochBB** API is closely related to the internal representation in terms of random variables, the GUI uses a more abstract representation. For example, models of cognitive processes are usually expressed in terms of a network of interacting *processing stages*, each associated with a waiting time distribution. For the sake of simplicity, when analyzing such networks, the semantic of the network representation within the GUI is more related to *processing stages* rather than random variables.

All these *processing stages* share the same form. They have at least one input socket that is the trigger of the stage and one output socket, that is the response. By chaining these stages, that is, connecting the output socket of one stage with the trigger socket of another stage, an equivalent of a sum of the waiting-time random variables of the chained stages is created. Although this representation is quiet uncommon in the field of statistics, it is very common in applied fields dealing with networks of random variables, like modeling of cognitive processes.



```
DEBUG @reduction.cc, line 106: Reduce convolution of <AtomicDensity
DEBUG @density.cc, line 57: Construct <GammaDistribution #0x7fd89e
DEBUG @density.cc, line 57: Construct <GammaDistribution #0x7fd89e
DEBUG @reduction.cc, line 106: Reduce convolution of <AtomicDensity
DEBUG @density.cc, line 57: Construct <GammaDistribution #0x7fd89e
DEBUG @reduction.cc, line 106: Reduce convolution of <AtomicDensity
DEBUG @density.cc, line 57: Construct <GammaDistribution #0x7fd89e
DEBUG @reduction.cc, line 106: Reduce convolution of <AtomicDensity
DEBUG @density.cc, line 57: Construct <GammaDistribution #0x7fd89e
DEBUG @reduction.cc, line 100: Reduce convolution of <AtomicDensity
DEBUG @density.cc, line 57: Construct <GammaDistribution #0x7fd89e
```

Fig. 2: Overview of the GUI of **StochBB**, the toolbar at the top of the main window (background), network edit field in the middle of the main window, the log view at the bottom of the main window (can be hidden) and a separate plot window at the foreground.

4.1 General overview

The main window of the GUI (see Figure 2) consists of two parts:

1. The tool bar at the top of the main window. Here, frequently used actions are collected like verifying the network, performing the analysis, adding or removing items. All these actions are also available at the main menu (not shown in Fig. 2).
2. The network view at the center of the main window. Here the network of *stages* can be viewed and edited. This view is described in more detail below.

Additionally, there is a log window at the bottom. By default, this view is hidden and can be enabled at the menu under *View* → *Show log*. It displays various messages from the core library emitted during the verification and analysis of the network.

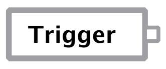
4.2 Editing a network

Within the network view, the network is shown and can be assembled or modified. New items (e.g., stages) can be added to the network by selecting them from either the tool bar or from the menu under *Edit* (see Figure 3 for some examples). These items can then be moved around in the network view by simply dragging them. An item can be removed by first selecting it with a single click and choosing *Edit* → *Remove* from the main menu or the *Remove* button in the tool bar.

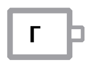
All network items have at least two properties that can be edited by double-clicking the item. Its label, shown inside the box representing the item in the network view, and a descriptive text (not shown) that allows to document the item. Many nodes, particularly those representing *stages* with some parametric waiting-time distribution, have additional parameters that specify the waiting-time distribution.

All networks items have so-called sockets (small rectangles at the left or right side of the item, compare Fig. 3). These sockets can be used to connect the items to form a network. All sockets on the left side of an item are input sockets and all on the right are outputs. Although a single output socket might be connected to several input sockets, every input can only be connected to single output socket. Like with network items, a connection can be removed by selecting it with a single click and choosing *Edit* → *Remove* from the main menu or the *Remove* button in the tool bar.

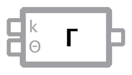
All items can be divided into five groups: *Random variables*, *random stages*, *join stages*, *transformations* and *plots*. *Random variable* items (e.g., Figs. 3a-3c) represent simple random variables with some parametric distribution, which usually (except for compound random variables, e.g., Fig. 3c) have no input slots. One special item is the trigger or stimulus item (see Fig. 3a). Although being a simple random variable with some



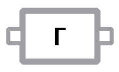
(a) Trigger/Stimulus item.



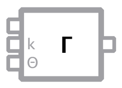
(b) Gamma distr. random variable.



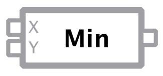
(c) Compound Gamma distr. random variable.



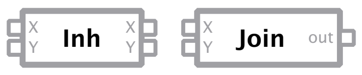
(d) Gamma distr. stage.



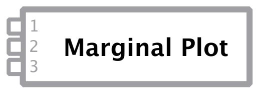
(e) Compound Gamma distr. stage.



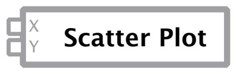
(f) Minimum stage.



(g) Inhibition stage.



(h) Marginal plot.



(i) Scatter plot.

Fig. 3: Some examples of stages available in the GUI application.

δ distribution, that is a constant, it is ment to specify an event at a fixed time-point. There can be an arbitrary number of trigger items in every network (e.g. a stimulus at $T = 0$ and a second manipulation of the experiment at some later time point).

All items of the *random stage* group (e.g., Figs. 3d, 3e) have a trigger input socket. Although being similar to the *random variable* items, they might be considered as *processing stages* with some parametric waiting-time distribution. That is, its output is a random variable being the sum of its input and a random variable distributed according to some parametric distribution. Like for the simple *random variable* items, there are also *random stage* items with a compound waiting-time distribution (e.g., Fig. 3e).

As mentioned above, an input socket may only be connected to a single output socket, although a single output may be connected to several inputs. This is due to the fact, that the function for joining several *signal paths* must be specified explicitly. *Join stages* (see Figs. 3f, 3g) are these functions that can be used to join signal paths according to some function.

For example the *minimum* stage (Fig. 3f) represents the minimum of the inputs X and Y. Or, in terms of processing stages, it gets triggered once either the stage connected to the X input or the stage connected to the Y input completed. A special *join stage* is the *inhibition* stage (Fig. 3g). It represents the *conditional sum* random variable introduced above and is the only stage represented by two items. The first item labeled *inh* forwards the first event and inhibits the second. For example, it triggers only the stages connected to the X output socket if the stage connected to the X input completes first. The second item labeled *join* then joins the two mutually exclusive signal paths into one. The *conditional* random variable introduced above, is not included as a *random stage* item as it would break causality.

Finally, the *plot* items (see Figs. 3h, 3i) allow for evaluating or sampling from the network. When added to the network, the *Marginal Plot* item (Fig. 3h) has no inputs at all. This item has a *graphs* property that defines the number of graphs and consequently the number of inputs for this item. After this property has been set to the desired value, the corresponding number of inputs will appear at the item. In contrast, the *Scatter Plot* item has always two inputs. They specify the two random variables to sample from for a scatter plot.

4.3 Verifying the network and running an analysis

Performing an analysis consists of two steps. In a first step, a network of random variables is derived from the stage-network representation used by the GUI application. This step will fail if any assumption (e.g., independence assumptions) made by the derived random variables is not met. This step will fail too, if there is a cyclic dependency between stages or an unconnected input socket. Once the network of random variables is derived, it is ensured that the network is consistent. Hence running an analysis and verifying the network share this first step.

In a second step, the derived network of random variables is actually analyzed. That is, the marginal distributions of the random variables being plotted are obtained and evaluated on the desired intervals. For the *Scatter plots* or *KDE plots*, a sampler gets instantiated to obtain samples from the random variables of interest. Finally, the plots are created and shown in separate plot windows.

5 An example: The divergence point

This example shows a simple *toy model* for some cognitive process that is able to produce a so called *divergence point* (e.g., Reingold et al., 2012). A divergence point of two response-latency distributions is the earliest time point at which the two distributions differ. Obviously, there exists no divergence point, if the two distributions are analytic (e.g., the convolution of Gamma distributions, Mathai (1982)) and there is some controversy whether a cognitive model that is formed by simple *processing stages* exists, which has a divergence point (e.g., Gomez et al., 2016).

Within this example, I not only show how a non-analytic response-latency distribution may arise using very common processing-stage models (i.e., these stages are also used in EZ-Reader model, Reichle et al. (2003)) that allows for a divergence point, but actually has a divergence point[2].

[2] At least one distribution being non-analytic is a necessary but not a sufficient condition for the

This very simple model consists of 4 Gamma-distributed stages V, S_1, S_2 and M, where the second stage S_2 is delayed by 500 ms. The model is

$$\text{control:}\ R_c = V + S_1 + M \tag{1}$$

$$\text{experimental:}\ R_e = V + \min(S_1, 500 + S_2) + M\,, \tag{2}$$

where $V \sim \Gamma(5, 30)$, $S_1 \sim \Gamma(10, 50)$ $S_2 \sim \Gamma(1, 200)$ and $M \sim \Gamma(1, 150)$.

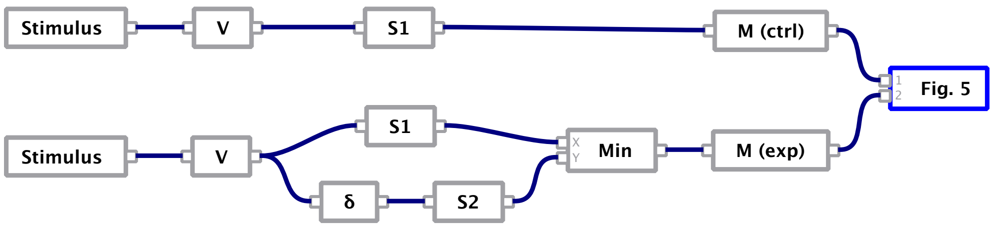


Fig. 4: Network of the second example model as visualized by **StochBB**.

This model (also shown in Figure 4) can be read as: Under control condition, the stimulus triggers a common *visual* stage V which then triggers a second stage S_1 that itself immediately triggers the response stage M. Under experimental condition, again the stimulus triggers the common stage V which then triggers S_1. Additionally to the stage S_1, V also triggers the delayed stage S_2 in parallel to S_1. The response stage M is then triggered by either S_1 or S_2, depending on which stage completes first. Consequently, the response latency under experimental condition is $R_e = V + \min(S_1, S_2 + 500) + M$.

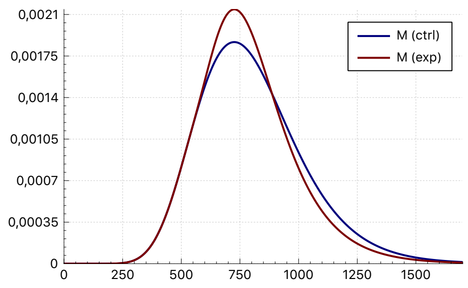
Figure 5 shows the plots generated by **StochBB**. The blue lines show the PDF of the response latency under control condition. Being a convolution of 3 Gamma distributions, it is an analytic function on the interval $(0, \infty)$ (Mathai, 1982). The red lines show the PDF of the response latency under experimental condition. As the distribution of S_2 is not analytic on the complete interval $(0, \infty)$ (discontinuity at $T = 500$ ms), a divergence point may exist at $T = 500$.

Due to the *minimum* stage and the fact that $P(S_2 < 500) = 0$ (delayed by 500 ms), it is ensured that the two response latencies are identical on the interval $[0, 500)$ ms. Hence, this simple model is able to produce a divergence point as both response-latency distributions are identical on the interval $[0, 500)$ and diverge thereafter.

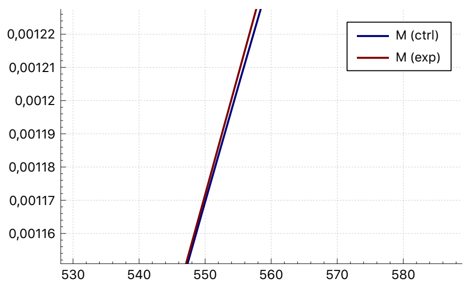
However, Fig. 5b shows that this divergence point cannot be estimated reliably from a finite data set as the two response latency distributions diverge very slowly[3]. Consequently, an estimate for this point (e.g. by means of estimators proposed in Reingold et al., 2012) based on a finite sample will always be heavily biased towards larger values.

existence of a divergence point.

[3] In fact, the distribution of the response latency under experimental condition is an example of a C^∞-function, that is a smooth function, that is not analytic ($\notin C^\omega$).



(a) PDFs of the response latencies under control condition (blue lines) and experimental condition (red lines). The *true* divergence point of the two response-latency distributions is 500 ms, however, the distributions appear to diverge much later (about 600 ms).



(b) The same PDFs but zoomed in. Here it get visible that the *true* PD is much earlier that visible in the right plot. In fact the *true* DP is not visible under any zoom level, as the PDFs diverge very slowly.

Fig. 5: Comparison of the response latency distributions as obtained from the mode shown in Fig. 4.

Moreover, if the experimental manipulation also affects the parameters of the $V, S1$ or M stage, the two response distributions will be different everywhere on the interval $(0,\infty)$ and consequently no divergence point exists. For a finite data set, however, an estimator (like Reingold et al., 2012) will always report a finite estimate as it is simply impossible to test for the existence of a divergence point on a finite data set.

6 Dealing with data

Given that **StochBB** is able to provide good approximations for the marginal PDFs of some (response) random variable, it is also able to fit a system of dependent random variables to data. For this, the **StochBB** API provides two functions. One is the `kolmogorov` function which returns the Kolmogorov-Smirnov statistic (KS-statistic, e.g., Kolmogorov, 1933; Smirnov, 1948; Marsaglia et al., 2003), the other function is the `logLikelihood` function. Both take the random variable, the number of bins and some data vector as arguments. These functions then uses an approximation of the PDF (for the log likelihood) or CDF (KS-statistic) evaluated on a regular grid spanning at least the data range for computing the corresponding value.

They can then be used to fit a network of random variables to some given observations. For example, consider the network of random variables introduced in the divergence point example above, describing the cognitive process under the *experimental condition*. One may try to re-estimate the $\theta = 120$ parameter of the stage S2. First, a relatively large sample is drawn ($N = 30,000$). Then one may implement a cost function of that parameter. Here, the negative of the log likelihood, given the samples is used. Finally the model is fitted to the the data by minimizing the cost function using a numerical optimization algorithm.

An example R code could be

```
library(stochbb)

# The "stages"
d  <- 500
V  <- gamma(5,30)
S1 <- gamma(10,50)
S2 <- gamma(1,120)
M  <- gamma(1,150)

# Response under "experimental condition"
Re <- V %+% minimum(S1, S2 %+% d) %+% M

# Sample Re
Nsam <- 30000;
sam <- new(ExactSampler, c(Re))
res <- array(0, c(Nsam,1))
sam$sample(res)

costFunc <- function(theta) {
  S2 <- gamma(1,theta[1]);
  R <- V %+% minimum(S1, affine(S2, 1, d)) %+% M
  return( -logLikelihood(R, 10000, res[,1]) )
}

# Find estimate
opt <- optim(c(100), costFunc, method="L-BFGS-B")
```

This code is then able to find an estimate of the θ parameter for the S2 stage by means of maximum likelihood (see Figure 6).

Moreover, the `logLikelihood` can then also be used to profile the likelihood (compare Figure 6) given the data, to obtain confidence intervals for the parameter estimate. There the true value of $\theta = 120$ is shown as a solid vertical line and the estimate as the dashed vertical line.

7 Summary

With this article I introduced **StochBB**, a framework that allows to analyze systems of dependent random variables efficiently without resorting to stochastic simulation. It uses analytic solutions for the marginal distributions where possible and resorts to numerical approximations if necessary. It provides a surprising good precision, one that would require a large amount of samples using stochastic simulation to achieve same

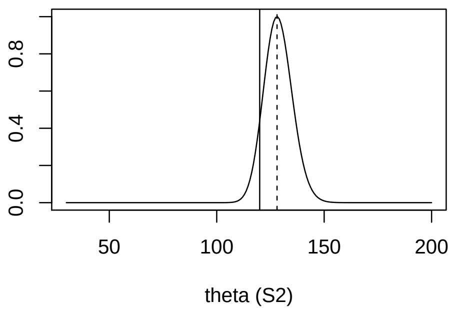


Fig. 6: An example of a likelihood profile for some θ estimate (vertival dashed line) using 30,000 samples drawn from a network of random variables, with $\theta = 120$ (vertical solid line) for the $S1$ stage.

precision. To this end, it is possible to fit networks of random variables to data by means of maximum likelihood or minimizing the KS-statistic.

Additionally, **StochBB** provides a GUI, which eases assembling of complex networks that are mend to describe random models like those typical for cognitive psychology by providing an abstract semantics related to *stochastic processing stages* rather than *random variables*. The abstract network representation of the GUI application eases the analyses of complex networks such that undergraduate students in applied fields are able to analyze and simulate such networks without the need to *translate* them to the *random variable* representation. To this end, the **SochBB** is a suitable tool to complement undergraduate lectures.

The software (including the core library, interfaces to *Python* and *R* as well as the GUI application) is available under the General Public License version 3 at https://github.com/stochbb.

References

Aho, A. V., Sethi, R., and Ullman, J. D. (2007). *Compilers, Principles, Techniques*. Addison Wesley.

Gomez, P., Breithaupt, J., Perea, M., and Rouder, J. (2016). Are divergence point analyses suitable for response time data? preprint.

Kolmogorov, A. N. (1933). *Sulla determinazione empirica di una legge di distribuzione*.

Marsaglia, G., Tsang, W. W., and Wang, J. (2003). Evaluating Kolmogorov's Distribution. *Journal of Statistical Software*, 8(18):1–4.

Mathai, A. M. (1982). Storage capacity of a dam with gamma type inputs. *Annals of the Institute of Statistical Mathematics*, 34(1):591–597.

Matuschek, H. (2016). StochBB – Stochastic Building Blocks: Application Programming Interface. http://stochbb.github.io/libstochbb.

Press, W. H., Flannery, B. P., and Teukolsky, S. A. (2007). *Numerical recipes*, volume 547. Cambridge Univ Press.

Reichle, E. D., Rayner, K., and Pollatsek, A. (2003). The E-Z reader model of eye-movement control in reading: comparisons to other models. *The Behavioral and brain sciences*, 26:445–526.

Reingold, E. M., Reichle, E. D., Glaholt, M. G., and Sheridan, H. (2012). Direct lexical control of eye movements in reading: Evidence from a survival analysis of fixation durations. *Cognitive psychology*, 64(1):177–206.

Smirnov, N. (1948). Table for Estimating the Goodness of Fit of Empirical Distributions. *The Annals of Mathematical Statistics*, 19(2):279–281.

Thomas, P., Matuschek, H., and Grima, R. (2012). intrinsic Noise Analyzer: A Software Package for the Exploration of Stochastic Biochemical Kinetics Using the System Size Expansion. *PLoS ONE*, 7(6):e38518.

Thomas, P., Matuschek, H., and Grima, R. (2013). How reliable is the linear noise approximation of gene regulatory networks? *BMC Genomics*, 14(Suppl 4:S5):1–15.

A Application programming interface

The application programming interface (API, see also Matuschek, 2016) allows to assemble complex systems of random variables programmatically in C++. All API classes are derived from the `Container` class which is an essential part of the memory management system used by **StochBB**. Usually a C++ programmer needs to keep track of all objects still in use and is responsible to free unneeded objects to avoid memory leaks. This can be a difficult task when dealing with complex structured objects cross-referencing each other. To ease the usage of **StochBB**, a *mark and sweep* garbage collector is implemented which keeps track of all objects being directly or indirectly reachable and freeing all unreachable objects. For this memory management system to work, it is necessary to treat all container objects like values although they represent references to objects allocated on the heap.

There are also Python and R packages (called `stochbb` too) that provide access to the classes and functions of the C++ API. This allows for a convenient construction of systems of random variables while maintaining the speed of the C++ implementation. These APIs are almost identical to the C++ one, except that R and `numpy` arrays are used instead of Eigen matrices and vectors.

The central class of **StochBB** is `Var`, representing a random variable. This could be a simple random variable having a specified distribution (`AtomicVar`) or a random variable that is derived from others like `Sum`, `Minimum`, `Maximum` or `Mixture`. All random variables (atomic and derived) have probability density functions attached. They can be accessed using the `Var::density` method which returns a `Density` object.

All `Density` objects have two methods, `Density::eval` evaluating the probability density function and `Density::evalCDF` evaluating the cumulative density or probability function. Assembling a system of random variables and evaluate their PDFs or CDFs is straight forward. Sampling, however, is not that trivial and is described below in some detail.

A.1 Assembling a system of random variables

In a first step, one may define a new gamma-distributed random variable with shape $k = 10$ and scale $\theta = 100$ as

```
#include <stochbb/api.h>
using namespace stochbb;

// [...]

Var X1 = gamma(10, 100);
```

Its PDF can then be evaluated as on a regular grid in $[0, 1000)$ with 1000 grid points as

```
// [...]

Eigen::VectorXd pdf(1000);
X1.density().eval(0, 1000, pdf);
```

The result of the evaluation is stored into the vector `pdf`. There are only very few basic or atomic random-variable types defined in **StochBB**:

Constructor	Parameters	Processing stage description
`stochbb::delta`	`delay`	A constant delay or a stage with a fixed waiting time.
`stochbb::unif`	`a`, `b`	A stage with a uniform-distributed waiting time.
`stochbb::norm`	`mu`, `sigma`	A stage with a normal-distributed *waiting time*.
`stochbb::gamma`	`k`, `theta`	A stage with a gamma-distributed waiting time.
`stochbb::invgamma`	`alpha`, `beta`	A stage with an inverse gamma-distributed waiting time.
`stochbb::weibull`	`k`, `lambda`	A stage with a Weibull-distributed waiting time.
`stochbb::studt`	`nu`	A stage with a Student's t-distributed *waiting time*.

More complex processing stages can be derived by combining these atomic random variables or as special cases of them. For example, the exponential distribution $\mathrm{Exp}(\lambda)$ is equivalent to a Gamma distribution with $k = 1$ and $\theta = \lambda^{-1}$.

A.1.1 Affine transformation of random variables

An affine transformation of a random variable X is of the form $a\,X + b$, where $a \neq 0$ and b are real values. An affine transformation can be obtained using the overloaded * and + operators or using the `stochbb::affine` function. For example, the code

```
#include <stochbb/api.h>
using namespace stochbb;

// [...]

Var X = gamma(10, 100);
Var Y = 3*X + 1;
```

constructs a random variable `Y` being an affine transformed of the random variable `X`.

A.1.2 Sums of random variables

Beside the affine transform of random variables, the most basic derived random variable is a `Sum`. This type represents the *chaining* of processing stages. Such a chain can be constructed using the overloaded `+` operator or the `stochbb::sum` function. For example

```
#include <stochbb/api.hh>
using namespace stochbb;

// [...]

Var X1 = gamma(10,100);
Var X2 = gamma(20, 50);
Var Y = X1 + X2;
```

A.1.3 Minimum and Maximum of random variables

Another simple derived random variable is the `Maximum` or `Minimum` class. As the names suggest, they represent the maximum or minimum of a set of random variables. They can be created using the overloaded standard library function `std::min` and `std::max` or the `stochbb::minimum` and `stochbb::maximum` functions. The latter take a vector of random variables.

```
#include <stochbb/api.hh>
using namespace stochbb;

// [...]

Var X1 = gamma(10, 100);
Var X2 = gamma(20, 50);
Var Y = std::max(X1, X2);
```

A.1.4 Conditional random variables

Conditional random variables as described above, can be created using the function `stochbb::cond` as following.

```
#include <stochbb/api.hh>
using namespace stochbb;

// [...]

Var X1 = gamma(10, 100);
Var X2 = gamma(20, 50);
Var Y1 = normal(0, 1)
Var Y2 = normal(0, 2);
Var Z = cond(X1,X2, Y1,Y2);
```

A.1.5 Conditional sums of random variables

Likewise the simple conditional random variable, conditional sums of random variables can be constructed using the `stochbb::condsum` function as

```
#include <stochbb/api.hh>
using namespace stochbb;

// [...]

Var X1 = gamma(10, 100);
Var X2 = gamma(20, 50);
Var Y1 = normal(0, 1)
Var Y2 = normal(0, 2);
Var Z = condsum(X1,X2, Y1,Y2);
```

A.1.6 Mixtures of random variables

Similar to the `Minimum` or `Maximum`, a mixture of random variables can be constructed using the `stochbb::mixture` function. This function takes a at least two variables and their associated weight. Such a mixture can be considered as a random process which randomly selects the outcome of a set of other random processes, where the probability of selecting a specific process is given by the relative weight assigned to each process.

```
#include <stochbb/api.hh>
using namespace stochbb;

// [...]

Var X1 = gamma(10,100);
Var X2 = gamma(20, 50);
Var Y = mixture(1,X1, 2,X2);
```

In the example above, the random process Y will select the outcome of X_1 with a probability of $\frac{1}{3}$ and the outcome of X_2 with probability $\frac{2}{3}$.

A.1.7 Compound random variables

An important class of derived random processes are compound processes. There, the parameters of the distribution of a random variable are themselves random variables. That is

$$X \sim f(x|A)\,, \quad A \sim g(a|\theta)\,,$$

where the random variable X is distributed as $f(x|A)$, parametrized by A, where A itself is a random variable distributed as $g(a|\theta)$, parametrized by θ.

Compound random variables are created using the same factory function like the atomic random variable types. In contrast to the atomic random variables, the factory functions take random variables as parameters instead of constant values.

```
#include <stochbb/api.hh>
using namespace stochbb;
```

```
// [...]

Var mu = gamma(10,100);
Var cnorm = norm(mu, 10);
```

instantiates a compound-normal distributed random variable, where the mean is gamma distributed while the standard deviation is fixed.

A.2 Sampling several dependent random variables

As mentioned above, sampling efficiently from a complex system of random variables is not trivial. First of all, it must be ensured that all atomic random variables are sampled only once. Otherwise, two dependent random variables may be sampled as independent. Moreover, the samples of derived random variables should be cached for efficiency.

StochBB provides a separate class that implements a proper sampler for a system of random variables, the `ExactSampler` class. This class allows to sample from several possibly dependent random variables simultaneously. Upon construction, the set of random variables to sample from, is specified. A sample from these random variables can then be obtained by the `ExactSampler::sample` method.

```
#include <stochbb/api.hh>
using namespace stochbb;

// [...]

Var X1 = gamma(10,100);
Var X2 = gamma(20, 50);
Var Y = std::min(X1, X2);
// Construct sampler
ExactSampler sampler(X1, X2, Y);
// Get 1000 samples
Eigen::MatrixXd samples(3, 1000);
sampler.sample(samples);
```

The `ExactSampler::sample` method takes a reference to a `Eigen::Matrix` where each column represents the random variable given to the constructor and each row an independent sample from the system.

For very large systems, sampling may get slow. Particularly if one is only interested in the marginal distribution of single random variables. For these cases, an approximate sampler for single random variables is provided, the `MarginalSampler`. This sampler uses an approximation of the inverse of the cumulative distribution function of a random variable to draw samples.

```
#include <stochbb/api.hh>
using namespace stochbb;
```

```
// [...]

Var X1 = gamma(10,100);
Var X2 = gamma(20, 50);
Var Y = std::min(X1, X2);
// Sample from Y on [0,500] in 1000 steps
MarginalSampler sampler(Y, 0, 500, 1000);
// Get 1000 samples
Eigen::VectorXd samples(1000);
sampler.sample(samples);
```

In this example, a `MarginalSampler` is constructed for the random variable `Y`. Using an approximation of its CDF on the interval $[0, 500)$ using 1000 steps. Then, the `MarginalSampler::sample` method is used to obtain 1000 independent samples.

A.3 Handling data

Being able to evaluate the CDFs and PDFs of random variables accurately, **StochBB** can be used to fit a system of random variables to observations by either maximizing the (log) likelihood or minimizing the Kolmogorov-Smirnov statistic (KS; e.g., Marsaglia et al., 2003). For these tasks, **StochBB** provides two functions. One called `kolmogorov`, evaluating the KS statistic and the other is called `logLikelihood` evaluating the log likelihood of some given data for a specific random variable. Both functions share the same interface (continuing the example above)

```
// [...]

// evaluate the KS statistic of the data samples for RV Y
kolmogorov(Y, 0, 500, 1000, samples);
// or evaluate the log likelihood
logLikelihood(Y, 0, 500, 1000, samples);
```

The first argument specifies the random variable, here `Y`. the next two arguments specify the range on which the CDF or PDF is evaluated, here $[0, 500)$. The fourth argument specifies the number of bins to use for the evaluation (here 1000) and finally the fifth argument specifies the data vector. Please note that both functions may log a warning message if some given samples lay outside of the specified interval. Both functions will then simply ignore these values.