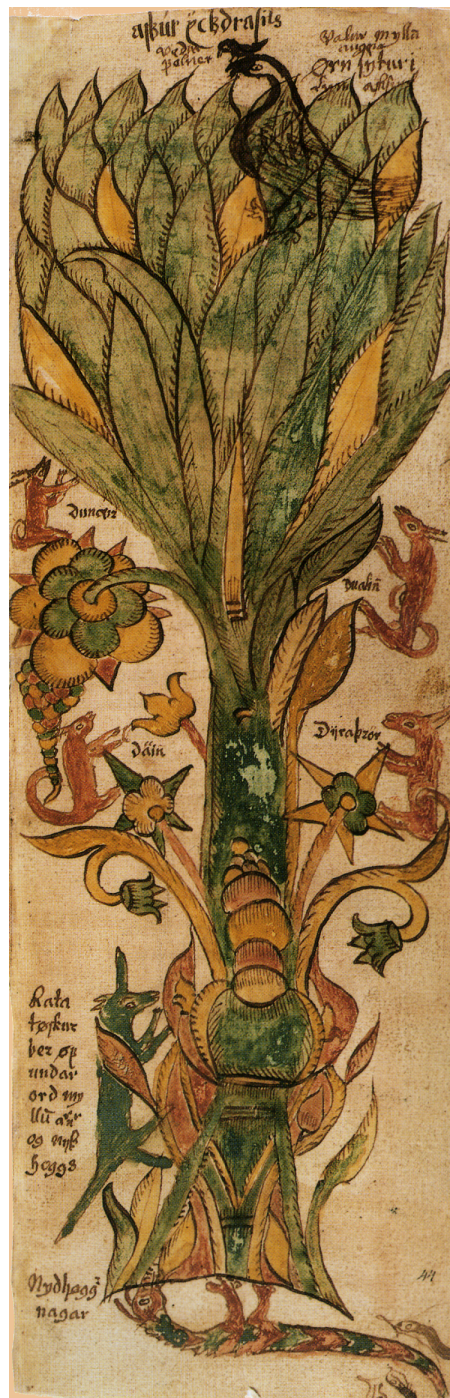


# YGGDRASIL Manual

Philip Müller, 13-928-304

July 13, 2019



**Front Image** Shows YGGDRASIL the source is [https://de.wikipedia.org/wiki/Yggdrasil#/media/File:AM\\_738\\_4to\\_Yggdrasil1.png](https://de.wikipedia.org/wiki/Yggdrasil#/media/File:AM_738_4to_Yggdrasil1.png).

# 1 Scope of This Document

The purpose of this document is to describes some of the fundamental aspects of the usage of YGGDRASIL. This manual is not complete, it only describes some fundamental usages of the library. However the source code is considered to be a good documentation, after all it's what YGGDRASIL itself is ~~look to when deciding what to do next!~~ However it is relatively big and not such a good starting point. This manual is considered a better starting point, but it will/can cover only the most basic usages.

## 2 Used Terminology and Definitions

In this section we will present the ~~used~~ terminology inside YGGDRASIL and this documentation.

**Global Data space** Also known as original data space. This is the data space, where the data was recorded. This is the domain that you passed to the constructor of the tree or was estimated from the data.

**Root Data Space** Also known as *rescaled* root data space. When the tree gets the samples, they can occupy almost any given volume of data space. Before the tree is generated, the samples are rescaled. The rescaling is such that the samples now reside in the unit hyper cube.

**Node Data Space** Nodes, especially leafs, need to know their size. So each node carries a hyper cube, that can be accessed with `getDomain()`. This cube describes which part of the space a node occupy. It is important that this cube is *relative* to the *root data space* and not to the global data space.

**Rescaled Root Data Space** It is important that there is a distinction between *rescaled node data space* and just *node data space*, but not in the case of the *root data space*, where both are synonyms.

We do rescaling for stability, so all samples that lies in a certain node are scaled such that they lie in a unit hyper cube. This is true for all leafs. ~~One says that the sample lies inside the rescaled node data space~~, but the leaf occupies the *node data space*<sup>1</sup>.

**Interval** In YGGDRASIL an interval is a right open *mathematical* interval. This means  $[a, b[$ , the end point is not included. If not specified otherwise, or given from the context, an interval means a right open interval.

**Leaf** A leaf in a a density element tree is a final node. This means that ~~the~~ both kind of tests, the GOF test and the independence tests, were accepted.

As a side note, in YGGDRASIL it is possible that ~~the~~ tests are not accepted but the leaf ~~was~~ not further split. The reason is that one dimension of the domain has fallen below the threshold length. However this happens only at depth above 100.

**True Split** A true split is a special kind of internal node. Such a split was directly created by the rejection of a test. It has an associated model and test instances, which are fitted and tested, but ~~the~~ test were rejected. See also the design specification for a further and more involved discussion.

**Indirect Split** An indirect split is not *directly* caused by the rejection of a test. It is always associated to a true split.

Indirect splits are created if a (tentative) leaf was split in more than one sub leafs. They are special in the sense that they have no fitted model associated to them. In a sense they are an effect of the requirement that we only have a binary tree.

**Gang Of Four** YGGDRASIL made use of several design pattern in its internal design. For the name ~~the~~ terminology ~~that is used in~~ the Gang of For book is used. The Wikipedia page is quite good.

---

<sup>1</sup>Note it is possible that some commensts in the code ~~does~~ not reflect this distinction correctly, the user has to take care in such situations.

### 3 Testing of the Code

The code was tested by writing several small programs that shows the correct working of their intended functionality. Also a scaling experiment was conducted, similar to the one that is presented in the paper. However no guarantee is given that the code will work correctly. I think it works correctly but all responsibility is rejected.

It is your responsibility alone to make sure that the output is correct. The best way is to redo the scaling experiments and extend them by new distributions or new incarnations of distributions that are already implemented.

### 4 Licence

The code that was written by Philip Müller is licenced under the GNU GPL version 3 or newer. This should be understood as an encouragement to anyone for improving this manual or the code base and share the improvements.

## 5 General Organization of the Code

The code is written in an object oriented fashion. This does not mean that we use virtual function dynamic dispatch everywhere. It means that general concepts of our thoughts maps to classes. The classes tries to do just one thing, but good. Instead of inheritance, composition was used to combine different functionality that belongs together, but putting them into one single class is not such a good idea.

As a small example, YGGDRASIL provides an interval class, that represents an interval. Everywhere in the code every interaction that uses an interval operation actually *uses* an interval object. This allows us to change the behaviour at *one single* location ~~and the effect will take effect everywhere~~.

As a matter of fact the code was designed to be self monitoring. This means that code is clustered with **asserts** that enforces preconditions. However they are only activated in ~~the~~ debug mode. For checks that are so important that they should be done and *enforced* all ~~the~~ time exceptions are used.

### 5.1 Folder Structure

The code is organized into different folders. ~~The folder separating~~ different aspect of the program. We have not separated the header and the code files, they simply reside in the same folder. Generally one header file declares one **class**, there are some rare exception, but then there are helper classes. Some function of ~~the~~ classes are implemented directly in the header file. Some other, most longer functions, are implemented in a separate **.cpp** file. If a class is large, then there are many **.cpp** files, which are split according to their functionality<sup>2</sup>.

#### 5.1.1 util

This folder contains all the utility code. For example the interval and the hyper cube are implemented there. This ~~is~~ code forms an important part of the internal structure of YGGDRASIL but is so general that it should not depend on any thing else.

However this folder also hosts code that is exclusively used by the statistical tests. For example the **class** that is used for creating the frequency table or the binning is implemented ~~here~~.

#### 5.1.2 interfaces

In *general* YGGDRASIL does not use dynamic dispatches, with some small exception where it makes sense<sup>3</sup>. The parametric model and the statistical test, goodness-of-fit (gof) and the independence test, are both implemented as an interface. This means that it is easy to change the underling implementation.

The interfaces offers a factory method, which allows an easy integration of new tests and models into the code base.

**New Implementation** If a new model or test should be integrated into the code base the user must implement the corresponding interface. ~~For that~~ a new class which inherent from the interface and all abstract functions must be implemented. The interface defines the specification which must be followed.

#### 5.1.3 core

This folder provides some fundamental **typedefs** that are used in YGGDRASIL. It also hosts the constant file, which define some constants that are used to control some aspects of the behaviour of YGGDRASIL.

If YGGDRASIL is extended it is a very good idea to read the files.

#### 5.1.4 samples

This folder contains some fundamental **classes** that deals with the storage of the sample. All of them are sophisticated wrappers around a vector<sup>4</sup>.

**Sample** This **class** is basically one single sample. It is implemented since in some contest one works with a sample. It is not very space efficient and should be only used when one single sample, or very few of them are needed to be processed.

---

<sup>2</sup>For example all constructors are gathered in one file ~~and~~ tree ~~travelling~~ function in a different one.

<sup>3</sup>At least from a engineering point of view.

<sup>4</sup>If we use the term vector in this document, we generally mean a **class** which conforms to the implementation requirements of the standard vector class. If the meaning is different it is mentioned or should be clear from the context.

**Dimensional Array** A dimensional array is the building block of YGGDRASIL. As we have stated ~~the~~ in the specification, the method uses independent dimension, so the dimensions are also stored independently. A dimensional array stores one single component of many samples<sup>5</sup>.

**Sample Collection** As the name implies this `class` stores a collection of many (complete) samples. It is implemented as a vector of dimensional array. It allows access to a single dimensions. It is also possible to access a single sample, but this is not so efficient.

This is the structure that is used to store the sample inside a node.

**Sample List** A sample list is just a `typedef` of a vector of type samples. It is a convenient data structure that allows fast access to a single sample, but not access to a single dimension. It is mostly used as interface. It is also not very space efficient, since it is a vector of vectors and has a lot of over head.

**Sample Array** This is a space efficient structure to store many sample, it is ~~the~~ recommend to use instead of the sample list. It is basically one single array ~~it~~ allows access to a sample.

#### 5.1.5 factory

This folder contains the factory method of the tests, the parametric model and the splitting strategy. It also contains the builder object which offers a nice way to simplify construction of a tree. The building object is basically a simple way to introduce named arguments to the constructor of the tree.

If new implementation are added the user should first ~~read~~ the files in this folder. This should ~~them~~ enable to integrate ~~the~~ new functionality.

#### 5.1.6 stat\_tests

This folder contains ~~all~~ the implementation of the statistical tests. Currently this are only the ones that uses the  $\chi^2$  tests.

#### 5.1.7 param\_models

This folder contains the implementation of the parametric models. Currently only the constant and the linear model are implemented.

#### 5.1.8 split\_scheme

This folder contains the split scheme.

#### 5.1.9 random

This folder is primarily needed for the testing infrastructure. It contains an interface, for the generation of random samples that obeys a certain distribution.

The interface also implements a factory method. The function is implemented in the `stdDistribution` file.

#### 5.1.10 verification

~~Here is the most of the code that is used for the conducting of a~~ scaling experiment. However it only offers the function to do this, ~~that~~ actual driver is located in the `prog` folder, see section 5.1.14.

**The Standard Scaling Experiment** YGGDRASIL has a standard scaling experiment. The scaling experiment involves fitting a tree for many different sizes. YGGDRASIL uses 20 different sizes, that are indexed from 0 up to 19. Each experiment is repeated several times, the standard configuration is 6 times. This is done to get meaningful averages. The process is repeated for each combination of model and splitter as well.

1. Generate a set of  $N$  random samples from the selected distribution. The number of samples is given as  $N = N(i) = 10^{2+\frac{1}{4}i}$ , where  $i$  is the size index.
2. Then a tree is generated, using the given configuration.
3. The tree is validated. This is done by checking if the tree is consistent.
4. We generate a set of  $N_q = N \cdot f$ , where  $f$  is the so called MISE factor, which has a value of 9.

---

<sup>5</sup>The component is the same for all samples.

5. We evaluate the tree at the  $N_q$  many sample points and record the PDF value there.
6. We compute the MISE. For that Monte Carlo approximation is used.

#### 5.1.11 tests

This folder hosts all the unit test functions. These are small functions that perform some super-simple computations and then check the result.

If you extend the functionality, please also offer unit tests.

#### 5.1.12 pygdrasil

This folder contains the code of PYGGDRASIL this is the wrapper that allows to use YGGDRASIL from within python. The interface is created with the help of the `pybind11` library, see section 11.1 for more information.

#### 5.1.13 pybind11

This folder contains the library that is used to build the interface. It is a submodule of the YGGDRASIL repository. It is set up to follow the stable branch of <https://github.com/pybind/pybind11>.

#### 5.1.14 prog

This folder contains some more involved programs. They will also be described later in more detail, see section 8 on page 11.

#### 5.1.15 tree\_gen

This folder contains the code that allows the sampling from the tree. It basically implements the second paper.

## 6 Implementation Notes

In this section we will present and list some important differences between the R and the C++-Implementation. Some of them were already discussed in the specification, so only a small note is provided here.

- The tree is now implemented as a real tree. This was done for providing the separation needed for insertion<sup>6</sup> of new samples and the parallelism.
- The bins are not determined by sorting. They are now computed from the theoretical distribution. Previously the bins were spaced such that they are uniformly according to the *empirical* quantiles. However now they are spaced uniformly according to the *theoretical* quantiles. This is actually the correct behaviour, according to the referenced paper.
- In R the number of samples was given by the number of samples after a unique was applied to a single dimension. A first effect is that the number of bins can differ in each dimensions. However, except for certain extreme cases, this should not matter much, since the numbers of bins grows very slowly.
- The application of unique also solved the problem of delta like distributions. The samples were simply removed. YGGDRASIL handles such cases quite differently. It continues to split the domain until a length of a domain falls below under a certain value. It then considers the domain as done, regardless of how bad the test is. Note that technically the rescaling would allow for infinity long split sequences, we still have to carry the size of the domain, relative to the unit cube. This means that we have to stop at a certain point, since this value will at some point reach zero. Note that technically the R code is also affected by this, however no checks are performed to detect such a situation<sup>7</sup>.
- YGGDRASIL only allows for 254 dimensions. You have to modify the tagged pointer if you want to increase the number.
- The tree that is generated by YGGDRASIL can not be copied. It is also not possible to save it.
- In Python `deepcopy` relies on Pickle support<sup>8</sup> however PYGGDRASIL does not implement pickle support. Some classes implement explicitly a copy constructor, this can be used like a copy constructor in C++.

---

<sup>6</sup>Which is not implemented.

<sup>7</sup>YGGDRASIL does this.

<sup>8</sup>Which is understandable.

- In the *R* code the split order in the case of a rejection of the independence test, which is composed of two splits, is determined by the p-value of the GOF test. YGGDRASIL does not use such an ordering step, the order is unspecific.
- The *R* code does not keep the samples after the tree is constructed, but YGGDRASIL keeps them. The reason is that they are needed for the incremental building and YGGDRASIL was designed to provide that feature at some point.
- The *R* code is able to rotate the sample first such that they are more suited for axes aligned bounding boxes. This is not implemented in YGGDRASIL.



## 7 Compiling

This section explains how to compile the library and programs. As build system CMAKE is used. Before you can compile the library you have to meet some fairly low requirements<sup>9</sup>.

### 7.1 Requirements

This section lists all the tools that are needed for compiling YGGDRASIL.

- You need a compiler that is able to support C++11. GCC supports this fully since version 4.8.1<sup>10</sup>.
- You need boost.
- You need Eigen 3.
- You will need HDF5. Currently it is not needed but it can be added any time.
- Optionally openMP.
- You need CMake, at least in version 3.13. It may be possible to lower the requirements, but there is no guarantee.
- You need `pybind11`, if you want to build the Python bindings. This dependency is a submodule of the YGGDRASIL project. So if Microsoft did not screw up and the project is still on github and you have cloned the project with the submodules, this dependency is no problem.

### 7.2 CMake Switches

The CMake file supports some switches that allows to change the behaviour at compile time. They can be used by `-D[NAME]=(ON|OFF)`, where name is one of the following switches.

**YGGDRASIL\_USE\_OPENMP** The default is **OFF**. This switch will enable the openMP support of YGGDRASIL. Note that the support is only very simple.

However it is possible to activate only specific tasks, see next options.

**YGGDRASIL\_OPENMP\_GOF** The default is **ON**. This flag it can be controlled if the GOF tests are performed in parallel. This means if the dimensions are tested one after another or in parallel.

**YGGDRASIL\_OPENMP\_INDEP\_PRE** The default is **ON**. This flag controls if the preprocessing of the dimension in the independence test is done in parallel or serial.

**YGGDRASIL\_OPENMP\_INDEP\_TESTS** The default is **ON**. It controls if the actual independence tests, the interaction between all dimensions, is done in parallel or serial.

**YGGDRASIL\_OPENMP\_FIT** The default is **ON**. With this flag it can be controlled if the fitting of the different dimensions of a model is done serial or in parallel.

**YGGDRASIL\_OPENMP\_PDF\_EVAL** The default is **ON**. This flag controls is the evaluation of pdf queries in the tree are done in parallel or not.

**YGGDRASIL\_NO\_MINIMAL\_SIZE** The default is **OFF**, setting this switch to **ON** is considered very dangerous. It will set the minimal length of the domain to zero, this means that the splitting is not stopped at a certain point. Setting the minimum length of a dimension will most likely result in the construction of an invalid domain and the termination of the code.

**YGGDRASIL\_COMPILE\_COMMANDS** Setting this flag to **ON** will instruct CMake to generate the file `compile_comand.json`. This file contains the compile commands. This flag is useful for inspecting the flags.

**CMAKE\_BUILD\_TYPE=MODE** This sets the compile mode. The default is **Debug**, this is not optimized code, but it is useful for debugging. Assertions are enabled.

For production it is recommended to use the **Release** mode. There the code is optimized and the asserts are disabled. However some checking is still performed.

---

<sup>9</sup>If your OS does not meet this requirements, consider installing a *real* operating system.

<sup>10</sup>I have version 8.

**YGGDRASIL\_WARN\_SIZE\_LIMIT=ON** When this option is set to **ON**, the code will output if a test was accepted because its size was too small. It will inform in which test it happens and which dimension was affected. Note this option only works in **Debug** mode.

**YGGDRASIL\_MAKE\_PYYGGDRASIL=ON** With this flag, which is by default set to **ON** **PYYGGDRASIL** is build. Note **PYYGGDRASIL** is not build the **pybind11** is not needed to be present.

### 7.3 Compiling the Library and Test Program

In this section it is explained how to build the library<sup>11</sup>.

1. First you have to go to the code folder. This is most likely in `${GIT_ROOT}/code/cpp`.
2. CMake should be configured such that it not allows to perform in source build. For that reason you first have to create a folder. This is traditionally named **build**, which is also ignored by **git**. After you have done that enter the folder.

The command to do that is: `mkdir build ; cd build`

3. Now you must call CMake. For that use the line

```
cmake .. [YOUR_FLAGS]
```

4. Now you can build the code. You can do this by issuing the following code:

```
make -j N
```

By giving the `-j` flag, you instruct make to use up to **N** cores. Note omitting **N** will start all, which is not recommended.

Now you have build the library. CMake will build a static library which reside in the build folder, its name should be `libYggdrasil.a`. This library can then be linked into other projects.

### 7.4 Compiling PYYGGDRASIL

This section explains how **PYYGGDRASIL** is compiled and integrated in your Python application. For that proceed as ~~it~~ is explained above ~~where~~ for the normal compilation, ~~see section 7.3~~. Of course you have to set the switch for building **PYYGGDRASIL** to **ON** which is the default.

After the build has finished, the build folder will contain several subfolders. One of them, which is only present if **PYYGGDRASIL** is build, will be called **pyyggdrasil**.

It will contain a shared module with a very technical name. On my system<sup>12</sup> the resulted object is named `pyyggdrasil.cpython-37m-x86_64-linux-gnu.so`. First part, up to first dot, should be the same on all platforms. The second part is platform dependent as you might have guessed.

---

<sup>11</sup>Not yet Python.

<sup>12</sup>A GNU+Linux system, with CPython.

## 8 The C++ Programs

This section briefly explains the usage of the important C++ programs. They are all implemented in the `progs` folder.

Assuming you have followed the building steps in section 7.3, then the `build` folder will contain a `progs` folder, that contains the generated executables. They can be called with the following syntax `./progs/NAME_OF_PROGRAM`, assuming you are inside the build folder.

### 8.1 Scaling Experiments

This program allows to performs the scaling experiments. It basically reproduces the experiment needed to redo some of the plots that are seen in the paper. If a new distribution or a new kind<sup>13</sup> is added to YGGDRASIL it must be adapted here. Note that the program is basically a wrapper function around the functions implemented in the `verification` folder, see section 5.1.10 on page 6.

It performs runs the experiments on all the different known distribution<sup>14</sup>.

It supports a number of arguments, the most important ones are. The arguments are parsed in UNIX manor. The switches for the distribution can also be preceded with a `no` to disable them.

`--save-dir=DIR` The results are saved in this folder. If the path exists it must be a folder, if the path does not exists it will be created.

`--dir=DIR` An alias to `--save-dir`.

`--none` Disable the testing of all distribution. This option is only useful when it is the first argument and specific distributions are turned on with later switches.

`--all` Activate the handling of all distributions. This switch is implicitly specified.

`--gauss` Enables the testing on Normally distributed samples. You can deactivate them by using the `-no-gauss` switch.

`--dirichlet` This switch enables the testing of samples that re distributed according to a Dirichlet distribution. Can also be turned of by specifying the `--no-dirichlet` switch.

`--outlier` This enables the testing of the outlier distribution.

`--uniform` This switch enables the testing for the uniform distribution.

`--gamma` This switch enables the testing of the gamma distribution.

`--beta` This enables the beta distribution.

#### 8.1.1 See Also

If you observe slow convergence when the median splitter is used see also section 13.2.1 on page 34 for further information.

### 8.2 Dump Loader

This program is provided for debugging obscure errors. The scaling experiments dumps its state, and this program can load it, such that a debugging is possible.

If you need this program, then read the source.

### 8.3 Plot Runner

This program was originally developed for generating the input for the plotting routine, but has evolved to a simple testing program. It supports a lot of flags and its output can give a rough estimate on the performance on the code.

It supports all the distribution flags that are also mentioned in the description of the scaling experiment program, see section 8.1. It only allows to run one single distribution at a time, so it does not support the `no` flags for disabling.

---

<sup>13</sup>Version of a distribution, see section 12.

<sup>14</sup>You have to specify them in the file directly.

### 8.3.1 Main Output

This program generates generally three output files that are needed/used by the plotting routines.

**sample\_x** This is a file which contains the sample locations. Each row is a single sample.

**pad\_ex** This is the file that contains the exact pdf values **of** at the sample locations. Each row contains one value, the ordering is the same as in the sample file.

**pdf\_est** This file contains the pdf that were estimated using the tree. Its ordering is the same as the ordering of the other pdf file.

### 8.3.2 Flags

The flags, additionally to the distribution flags witch are listed in section 8.1, the program supports the following flags.

**--fitN SIZE** The program generates SIZE many samples for generating the tree.

**--LinMod** The tree uses the linear model as parametric model.

**--ConstMod** The constant model is used for the tree.

**--Median** The median splitter is used, for determining the split location. This splitter is also known as “score” based splitter.

**--Size** This is the size based splitter, that cuts each domain in half.

**--kind** With the distribution switches you select the distribution you want to test. But which distribution you want to use from this family. This is selected with this switch, see section 12 for a list of the canonical distributions. Note, this number can also be negative.

**--print [FILE]** With this flag the generated tree will be outputted to the screen. This is for a visual inspection, but for greater trees it is not very useful. If the optional argument FILE is specified, the tree will not be outputted to the screen, but the output is written to FILE instead.

**--grid** This option affects how the query samples<sup>15</sup> are generated and also the output. This option has only an effect if the dimension of the underling distribution is one or two. If not specified the query samples are generated from the underling distribution.  
If the dimension is larger the switch is disabled automatically.  
See also section 8.3.3 and 8.3.4 for more information.

**--no-write** If this switch is given, then output, described in section 8.3.1, is not generated.

**--gPoint1 N** This argument controls the number of query points, for outputting and calculating the MISE. In grid mode this is the number of grid points used in the first dimension. In non grid mode the total number of query points is given by this number and the one specified by the **--gPoint2** switch.

**--gPoint2 N** This is the number of points for the second dimension. If not specified it will have the same value as the first dimension.

**--save-dir DIR** The output files are saved into that folder. Actually not correct, in this folder a subdirectory is created, where the final files are finally located.

**--exp-Name** This is the name of the experiment. If empty or not given, the canonical name is used.

### 8.3.3 Output in Grid Mode

When the grid mode is selected and applicable<sup>16</sup> then the **samples** are generated on a grid. This allows to perform numerical integration. The pdf of the theoretical distribution and the estimated one is computed. As well as the MISE is outputted.

Note that due to the restriction to a finite sample domain, the *theoretical* distribution must not necessary integrate to one. Since some part is striped. However the estimated distribution is constructed in a way that it should always integrate to one.

---

<sup>15</sup>The samples that are generated for computing the MISE.

<sup>16</sup>Meaning the underling distribution has a dimension of two or less.

### 8.3.4 Output in None-Grid Mode

In non grid mode, the samples are not generated on a grid, but are generated by the distribution object. The MISE is then calculated by an important sampling scheme.

There no integration is performed but the mass is calculated. The mass is technically a scaled mean value of the pdf value,  $\langle \text{pdf} \rangle$ . The value of the mass is pointless, however the mass of the estimated distribution should be similarly to the one of the exact distribution.

## 9 Plotting

YGGDRASIL provides some very rudimentary plotting support. They are done in python. You can find the code in `${GIT_ROOT}/code/plotting`.

### 9.1 Scaling Plot

The script for generating the scaling plots is implemented in the file `scalingPlotter.py`. It expects as argument the path to a folder which contains results from the scaling program, which was described in section 8.1 on page 11.

### 9.2 1D Plotter

The plot runner program, that was presented in section 8.3 on page 11, can be used to dump the estimated tree. The script `Plotter1D.py` can be used to visualize this. Currently only 1D plots are supported.

As argument it expects the path to a folder which contains the main output, see section 8.3.1 on page 12. The calling syntax is:

```
python3 ./Plotter1D.py [ops] FOLDER_TO_DUMP
```

It also supports some arguments:

`-x` The  $x$  axis will be plotted in logarithmic scale.

`-y` The  $y$  axis will be plotted in logarithmic scale.

`--x-start` Defines a range where the plotting should begin. Must be specified together with `--x-end`.

`--x-end` Set the end of the plotting range.

`--name` This is the name of the distribution. Must not contain spaces or other funny names, will be part of the filename.

`--savedir` DIR Specify the folder where the plot should be saved. If not specified the current directory (`.` in UNIX speak) is used.

## 10 The C++ Interface

YGGDRASIL is designed as a C++ library. As such it provides an interface that allows to use its functionality in other programs. The programs presented in section 8 on page 11 are implemented that way.

It is recommended to use the C++ interface for various reasons. One of them is that it allows a much more specific control and access to the representation of the tree.

Also a Python interface is provided, which is called PYYGGDRASIL. Since it is anticipated that YGGDRASIL is used primary through its Python interface, only PYYGGDRASIL will be documented in this manual. However PYYGGDRASIL was designed to mimic the C++ interface, so this document is also a good starting point even in the case the C++ interface will be used in the end. Some important deviations between the two interfaces are documented. However a cross check with the actual source is recommended.

### 10.1 Additional Documentation

The C++ code is documented with Doxygen. Thus an excellent documentation can be created by running Doxygen. It is advised to first read the Python section before diving into the Doxygen documentation.

### 10.2 Implementing new Models and Tests

It is quite easy to extend YGGDRASIL with new models and new tests. This section is not a complete guide how to do it. However for a skilled software developer the provided information should be enough to extend the program.

The tree does not know about models and tests. The only thing it knows are interfaces<sup>17</sup>. To extend YGGDRASIL with a new functionality the respective interface must be implemented. The interface is extensively commented. It defines contracts that the implementation must fulfil. In order to be able to inform YGGDRASIL about the new functionality, it must be integrated into the builder process. All functionality related to the building processes is implemented in the folder `factory`, see section 5.1.5 for more information. The code in that section is mostly self explanatory.

#### 10.2.1 Note On Python

Note that the C++ builder concept was designed for extensibility. The Python Builder concept is not, the wrapper has to be adapted for that.

## 11 Python Interface

This section explains how the Python interface works and how it can be used. As it was stated in the section about the C++ interface the Python interface mimics the C++ interface. It is object oriented and makes extensive use of the Duck-Typing system that Python has by providing lots of overloads. The interface is called PYYGGDRASIL.

### 11.1 Implementation

PYYGGDRASIL is implemented with the help of the `pybind11` library<sup>18</sup>. The code is located in the folder `pyyggdrasil`, see section 5.1.12 and 5.1.13 for more information.

#### 11.1.1 Design Principles

PYYGGDRASIL is designed to be object oriented. This means that there is no requirement to create an array in an obscure fashion fill it with values in a certain way and hope that you do not screw up and pass it as the right argument<sup>19</sup> to a function that takes several arguments of the same type, and with a complete nonsensical order that nobody can remember. Instead there is a `class` for each and every concept that you encounter in PYYGGDRASIL. The underlying type system<sup>20</sup> will enforce the correct order.

The second aspect is that PYYGGDRASIL follows the doctrine that *trust is good, but (exerting tight) control is better*. This is different from YGGDRASIL on the C++ level, where the creation of semi invalid object is easily possible and exerting control happens only when the object is actually used.

---

<sup>17</sup>See also section 5.1.2 on page 5 for more information.

<sup>18</sup>See <https://github.com/pybind>.

<sup>19</sup>This is more a problem in C++ than in Python, at least if you use named arguments.

<sup>20</sup>Keep in mind that you actually use C++ so you have a type system, that is not perfect, but you have one.

## 11.2 Documentation

First of all the oral history is your friend and best source of documentation. This manual provides a generally introduction to PYYGGDRASIL. It basically lists what is there and what is possible to do with it. It does not really explain things by providing cute little examples, after all it is written in C++ and this is a Spartan language<sup>21</sup>, it explain things by listing the classes and their intent and also some of the important methods.

pybind11 also provides an easy access to the integrated documentation system of Python. So you can simply write `help(NAME_OF_PYYGGDRASIL_CLASS)`, inside the interpreter after importing PYYGGDRASIL, to get a list of all functions of an object and what they do. It also works with the whole module, by writing `help(pyYggdrasil)` if you have not aliased the module upon importing it PYYGGDRASIL. So if starting up the interpreter and writing two lines then *any* documentation won't help you.

A second documentation are some small example that shows the basic usages of PYYGGDRASIL. They are implemented by means of several Jupyter notebooks, the reason is that they are quite interactive and can be better commented than a classical Python script.

Additionally there are also some smaller scripts, that ~~does~~ more non-trivial things. They are less well commented on the objects, but focuses more on the interaction of them.

As a last resort you can turn your attention to the *ultimate* documentation, the source code. After all it *is* the documentation that YGGDRASIL *itself* uses when deciding what to do next<sup>22</sup>.

## 11.3 Importing PYYGGDRASIL

This section is probably incomplete, since I am not a Python guru, help is welcomed.

Compiling the PYYGGDRASIL module, as it was described in section 7.4 on page 10 is not enough to use PYYGGDRASIL. You have to import it into your project. The object that is generated might be a binary file, but for the Interpreter it is just a regular module that you can load like any other project. You just have to write `import pyYggdrasil` and the module is loaded. As you can see there is no need to enter the long and complicated name of the object, just the first part is enough.

But this is still not enough. You have to put the object at some location where the Interpreter can ~~found~~ find it. The easiest way is to place a symlink to the shared module in the directory where your python code reside, aka. ~~form~~ `python MY_AWESOME_PROGRAM.py`, that points to the shared library. The user can find more information on this matter on <https://docs.python.org/3/tutorial/modules.html#the-module-search-path>.

## 11.4 Error Handling

YGGDRASIL enforces some consistency with exceptions. pybind11 will transparently convert an exception that is ~~thrown~~ in C++ into a Python exception. On ~~th~~ Python side they just ~~looks~~ like regular exceptions. If YGGDRASIL is compiled in debug mode, there are also assertions that enforce the correct behaviour. If an assert is violated `std::abort()` is called. This will lead to the termination of the program, which is the Python interpreter that is currently running.

## 11.5 Sample and Sample Containers

In this section the storage classes of PYYGGDRASIL are discussed. These are one building block of the YGGDRASIL.

### 11.5.1 Samples

As ~~it~~ was said before, YGGDRASIL provides a class for everything and so there is one for modelling a *single* sample. The sample is implemented in the Python object `pyYggdrasil.Sample`. This class is basically a wrapper around an array. It is not so space efficient, but offers a convenient and very direct way to manipulate a single sample, possible in a `for` loop.

**Creating A Sample** There are several methods to create a sample. One restriction is that the dimension of the sample must be greater than zero.

`pyYggdrasil.Sample(d)` This will create a sample of dimension `d`. All the components will be initialized to zero.

---

<sup>21</sup>Linus Torvald said this about C, but if you compare C++ to Python it can be also applied.

<sup>22</sup>This is adapted from *Life with UNIX*.

`pyYggdrasil.Sample(d, v)` This will create a sample of dimension `d`, and all the components will be initialized to the value `v`.  
Note that `v` must not be `nan` or `inf`.

`pyYggdrasil.Sample(PYTHON_LIST)` This will create a sample out of the provided Python list<sup>23</sup>.

`pyYggdrasil.Sample(pyYggdrasil.Sample_Instances)` This constructor will perform deep coping of the sample. The reason for this existence is that deep copying in Python needs pickle support.

**Other Function of the Samples** The sample provides some more functions. They are basically “natural” ones, that are functions that you would expect a sample to have, again `help(pyYggdrasil.Sample)` is your friend. Like getting its dimension, which is done by calling `nDims()` on the sample.

It also has a function which is called `isValid()`. It checks if the dimension is greater than zero or if its components are valid<sup>24</sup>.

The class mimics an array so it also supports the `operator[] (i)` syntax for both reading and writing a component. However it is not possible to set a component to `nan`.

**Generating An Invalid Sample** Under normal circumstances it is impossible to create an invalid sample. However in certain situation it could be needed to create one. For that the function `CREAT_INVALID_SAMPLE(d)` is provided<sup>25</sup>. This function will generate a sample that has `nan` as component and dimension `d`. The dimension can also be zero, in which case the sample is also invalid, because it does not have any component.

## 11.6 Sample Containers

Since working with `PYYGGDRASIL` most likely will involve working with *many* samples `PYYGGDRASIL` provides also classes to efficiently store many sample. There are three different container. There are that many container because different situations needs different solutions. Their existence is mostly due to the `C++` heritage of `PYYGGDRASIL`.

### 11.6.1 Universal Container Interface (UCI)

The *Universal Container Interface* or UCI for short, offers a set of methods that all containers implements. Since Python uses Duck Typing the container can be used interchangeably, and as long as functions respect the interface they can operate on all containers in the same uniform manner.

During its lifetime the size, numbers of samples, can change. However the dimension of the container<sup>26</sup> is fix during the lifetime of a container.

The interface is split into a base and extensions. Only the base is required to be implemented.

**Construction** This is part of the base. It deals with the construction of the container.

`ctr(d)` This will construct an empty container, meaning that the container has not any samples. The dimension of the container is set to `d`, as mentioned this can not be changed.

Note that this constructor does not implies that no memory is allocated.

`ctr(d, N)` This will construct a container of size `N`, its dimension will be set to `d`. The values of the samples are unspecific.

`ctr(src)` All container provides a copy constructor. This constructor will deep copy the container `src`. Note that the type of `src` must be the same as the container that should be constructed.

Because of its important the sample collection can be constructed also from the two other containers.

See also the third extension of the interface.

**Accessing** The Container basically models an array and a sample can be accessed with a zero based index. It is important that due to the nature of some container it is not possible to return a reference in all cases<sup>27</sup>. So the function operates on *copies*.

---

<sup>23</sup>For that binding the `pybind11::list` object is used.

<sup>24</sup>Not funny values.

<sup>25</sup>Note that `YGGDRASIL` and thus `PYYGGDRASIL` write create without *e*. This is intended and an homage.

<sup>26</sup>This is rather the dimension of the samples that can be stored.

<sup>27</sup>This could be circumvallated by implementing proxy classes, but it was considered to be not worth it.



**addNewSample(s)** This function is not directly involved with accessing samples. It adds the sample **s** to the container. It is appended at the end, it is like the **append** function of the list object. It can cause a reallocation of the underlying storages, however this is only a problem if references to samples exists<sup>28</sup>. It imposes the restriction that the sample has the same dimension than the container and that it is valid.

**setSampleTo(s, i)** This function writes to the container. It exchange the sample that was previously associated with index **i** with the sample **s** that was passed as argument. This will not cause a reallocation. It imposes the restriction that the sample has the same dimension than the container and that it is valid.

**getSample(i)** This function returns a *copy* of the sample, that is associated with index **i**.

**Capacity Management** These functions deals with the management of the size of the container. They impose some restrictions that aims to detect some kind of errors.

**resize(newSize)** This function changes the size of the container, to **newSize**. However this function is only allowed if the current size of the container is zero. If this function is called on a non-empty container, an error is generated.

**clear()** This function clears the container and sets its size to zero. It is guaranteed that the associated memory is deallocated by this function<sup>29</sup>.

**reserve(n)** This function performs preallocation of the container. It only has an effect if the size of the container is not set, zero. If called on a non empty container, it is considered a **nops**, especially no exception is generated.

**Query Functions** This functions allows to request some information about the container.

**nDims()** This function returns the dimension of the container.

**nSamples()** This function returns the number of samples that are present in the container.

**First Extension (Python)** This extension of the interface is only present in the Python interface. The reason is that the implementation of proxy classes for the **operator[]** is actually build in when using **pybind11**. It specifies two functions.

**\_\_getitem\_\_(i)** This function implements the *reading* bracket operator. It returns a *copy* of the sample that is stored at location **i** in the container. It is equivalent to the **getSample(i)** function.

**\_\_setitem\_\_(i, s)** This function implements the *writing* bracket operator. It sets the sample that is associated with index **i** to the provided sample **s**. It imposes the restriction that the sample has the same dimension than the container and that it is valid.

**Second Extension (C++ & Python)** This is the second extension to the UCI. It requires that the container provides at least reading access, which allows to return copies, to a single dimension. The sample collection is the only collection that naturally allow such kind of accesses, the other container have to construct such a container. It is thus not recommended to use this function on containers others than sample collection if not especially needed.

**getDimensionArray(d)** This function returns a dimension array of the dimension **d**. A dimensional array is an array that consists of all samples, in the same order as the container, but only contains a single dimension.

**Note on Python** In Python an **Eigen** type is returned. In every call on to this function, on *all* container types, will create a copy.

**Third Extension** This extension describes how containers can be constructed from matrices. On the C++ level **Eigen** matrices are used. The interface was designed according the recommendations of the **pybind11** manual, so interaction with NumPy is possible without much problem. It is implemented by all containers. The extension describes a constructor and a new function.

**ctr(mat, view)** This constructor reads in the matrix **mat**. How the matrix is interpreted is determined by the **view** argument, see below.

**getMatrix(view)** This function returns an **Eigen** matrix. The way the matrix is constructed is analogue to the interpretation of the matrix by the constructor and can be controlled by **view**.

---

<sup>28</sup>This is never the case in Python. And in C++ it only affects the sample list.

<sup>29</sup>In the C++ standard, this function does not guarantees that the memory is freed.

**The view Parameter** The `view` parameter controls how the matrix is interpreted. It is implemented as an `enum` that encodes where if a sample occupies a row or a column<sup>30</sup>. The `enum` is called `eMatrixViewSample` and can have two values<sup>31</sup>.

**Row** In this mode each row contains one samples. The number of columns is interpreted as the dimension of the samples and the number of rows is the number of samples.

**Col** In this mode one sample occupies one column. Thus the number of samples is equal the number of columns of the matrix and the dimensions is equal the rows.

Note on the C++ level this value is called `Dimension`.

**Matrix Order** There are two ways to store a matrix in memory row and column major order. The C++ code supports both, however only row major is supported by `PYGGDRASIL`. The motivation is that this is the default of NumPy.

Note that depending on the matrix order the efficiency to construct a container from a matrix depends on the `view` and the storage order. Below we have note which is the best choice for the `view`, these recommendations assume that the matrix are stored in row major order, which is the default in NumPy.

### 11.6.2 Sample List

The sample list is one of the container, that is provided. Its existence is mostly historical. It is available under the name `pyYggdrasil.SampleList`.

It is not very space efficient and should thus not be used. It is basically an array of samples. Thus imposes an significance over head on the memory system.

Its sole merit is on the C++ side. Since it is the only container that is able to return references to samples when the `operator[]` is used. However this makes it possible to add a sample with a different dimension than the container is possible. Thus great care has to be taken. It is thus recommended to use the named accessing function, that are specified by the UCI.

Note that this problem does not exists in Python, since there is an extra layer that performs a check on the dimension.

The performance of the construction of a list using a matrix is best if the `view` is set to `Row`.

### 11.6.3 Sample Array

This is a very efficient container and recommended to be used if the samples are not indented to be used for fitting a tree. It stores the samples in one single array. The samples are stored consecutive in memory.

It is available as `pyYggdrasil.SampleArray`.

The performance of the construction of a list using a matrix is best if the `view` is set to `Row`.

### 11.6.4 Sample Collection

The sample collection is also a very efficient way of storing the samples. It is not as efficient as the sample array, but the differences are negligible.

Instead of storing a single array, it stores each dimension in its own array.

This container is used internally by the tree. Thus it is possible to directly load the samples into the tree without the need of coping them first into a sample collection. However the passed collection will be empty afterwards.

This container is available as `pyYggdrasil.SampleCollection`. It can be constructed from all three container types by performing a deep copy.

Note that contrary to the other two container the best way to construct a sample collection from a matrix is to set the `view` parameter to `Col`. Meaning that each column contains one sample.

## 11.7 Utility Classes and Functionality

Here the utility classes are presented. These are classes that models some imported helper concept. Only the most important ones are present in `PYGGDRASIL`.

<sup>30</sup>Note that this is different from C++, where the `enum` is used to encode how a row should be interpreted. This divergence will not be changed.

<sup>31</sup>Note that actually only the `Row` value is recognised, all other values are considered `Col`.

### 11.7.1 Interval

The interval class, that is available as `pyYggdrasil.Interval`, implements the concept that was presented in section 2 on page 3. This means it models a right open interval. It is basically a pair of numbers that is enhanced with some meaningful functions.

It can be constructed by specifying the bounds. So by writing `pyYggdrasil.Interval(a, b)` an interval that spans the range  $a \leq x < b$ , is created. However an exception is generated if an invalid interval is generated.

The interval has the usual functions that you would expect, like `lower()/upper()` for obtaining the lower and upper<sup>32</sup> bound of the interval. Please use `help(pyYggdrasil.Interval)` to obtain a full list of the supported functions.

**A Valid Interval** The only function that is covered here in deep is the `isValid()` function of the interval. `PYGGDRASIL` imposes strong control over invalid interval. Although they are relatively common on the `C++` level and especially in the inner working of the tree, they are considered impossible to surface in `PYGGDRASIL`. Encountering one most likely indicates an internal error.

This section is here because it is very important to understand when an interval is invalid and when not. Since the validity of the interval limits the splitting depth of the tree<sup>33</sup>. The process to verify the validity of an interval is quite involved.

First of all a valid interval must have bounds that are not `nan`, but infinity is permitted. Further the lower bound must be *strictly* less than the upper bound. Then the length, which is defined as upper bound minus lower bound, must be *strictly* greater than zero. If the length happens to be infinity<sup>34</sup> then the interval is considered immediately valid<sup>35</sup>. Then it is checked if the length is zero, if so the interval is considered invalid. If this checks also succeeds it is tested if the inverse of the length,  $\frac{1}{l}$ , is a valid number, which is also greater than zero, seen from a finite precision point.

### 11.7.2 Hyper Cube

Since the method aims at settings where the problems involves more than one dimensions, it is clear that the interval is not enough. The extension of the interval to many dimensions is a hyper cube, which is available under `pyYggdrasil.HyperCube`. It is basically an array of intervals, one for each dimension where it describes the extend in the corresponding dimension. It also reimplements most of the functions that are provided by the interval again, but extended to several dimensions.

It is important to note, that the hyper cube models an immutable object. This means once it is created it is not possible to change the object itself. You can assign an other hyper cube to the object, but not modifying the underling structure. But under certain condition it is possible to do that.

**Creation of Hyper Cubes** There are two different ways creating a hyper cube. The first one is by simple coping an interval, this is done by the following invocation `pyYggdrasil.HyperCube(d, interval)`. This will create a hyper cube of dimension `d`, the `d` intervals that forms the cube, are copies of `interval`.

It is immediately clear that this mechanism is extremely limited. In order to construct more complex domains, we must use an exception in the immutability of the object.

A dimension can be changed, if and only if, said dimension is invalid. Thus in order to construct a more complex interval you must first create an invalid hyper cube, with the correct dimension. For that the function `pyYggdrasil.CREAT_INVALID_CUBE(d)` is used. This function returns an invalid cube of dimension `d`<sup>36</sup>. The intervals of the dimensions are invalid intervals, which have `nan` as boundary values.

After the invalid cube is created, the intervals can be exchanged, one by one. For that the member function `setInvalidDimension(d, interval)` is provided. This function exchanges the *invalid* interval of dimension `d` of the hyper cube, with the *valid* interval `interval`.

It is considered an error if `interval` is not valid. It is also an error if it is attempted to exchange a valid interval of the cube.

**Creation from a List** The hyper cube also supports the creation out of a list of intervals. As argument in the wrapper `std::vector<Interval>` is used. This means that every Python structure that `pybind11` can convert to that type is supported.

As far as it goes this includes lists and tuples.

---

<sup>32</sup>not included.

<sup>33</sup>Although this should be only a problem in obscure cases.

<sup>34</sup>This happens if at least one of the bound is infinity.

<sup>35</sup>This is a case that may be removed in the future.

<sup>36</sup>`d` can also be zero, but then the cube is useless for the process explained here.

**A Valid Hyper Cube** A valid hyper cube is an object, that meets several requirements. First of all all of its constituting intervals must be valid, in the sense as it was described in section 11.7.1 on page 19. Further the volume of the cube, which is nothing more than the product of the length of the intervals, is a valid number that is greater than zero. Second the inverse of the volume is again valid and strictly positive.

Note that the definition of the validity of an interval has an interesting consequences<sup>37</sup>. It can happen that the intervals are valid, individually, but their volume is so small, that we have problems there. So the condition that every *single* interval of a cube is valid, does not imply that the cube is also valid.

The cube provides two functions for validation. One is `isValid()`, this function performs the tests that are described above. The second function is `isValid(d)`, this only checks if dimension `d` has a valid interval. So iterating through the dimensions and testing if they are valid is not enough<sup>38</sup> to test if a cube is valid or not. For that the `isValid()` function<sup>39</sup> has to be used.

### 11.7.3 Depth Map

This is a very condensed description of the tree. It is basically a Python `dict` object and it is possible to convert it into one, the format is as described. It maps integers to integers, the key value of the map, represents a depth of a leaf. The depth of a leaf is the number of descend operations that is needed to reach the leaf. The associated value, is the number of leafs that where found that have that depth.

It supports other functions, please see the Python `help(pyYggdrasil.DepthMap)` output. Most important are the `getDepthMean()` which returns the mean depth of the leafs. Also important is `getDepthStd()` which returns the standard deviation of the depth.

### 11.7.4 Tree Facade

PYYGGDRASIL is a C++ library, that beliefs in data encapsulation, something that is not possible in Python. But since PYYGGDRASIL is technically C++ we can impose this.

YGGDRASIL or rather the tree object will not allow user code to access internal structures. This is done such that they are not messed up and remain in a valid state all the time<sup>40</sup>. However for some operations at least a controlled access to the internal data is required. This access is provided by a proxy object that is available to PYYGGDRASIL as `pyYggdrasil.NodeFacade`.

This object models a controlled access to the wrapped node by mimic a constant pointer to a constant node object. On the C++ level such a facade can technically point to any kind<sup>41</sup> of node. However in PYYGGDRASIL it is only possible to obtain references to leafs, encountering a non-leaf facade in Python indicates serious problems.

It is also possible to obtain a *copy* of the sample collection that is owned by the node. Depending on the situation they can be empty. It is not recommended to request the data, since it involves a copy. Also the user has to keep in mind that the data is rescaled and thus lies inside a unit cube.

What may be useful for the user is to obtain the domain of the node. The domain is the hyper cube that describes the fraction of space that is occupied by the leaf. This cube does not describe the position of the leaf in the original data space, but in the rescaled root data space.

**The Mass of a Node** This section is more intended for the C++ level. A node, regardless of its kind, has an associated mass. The mass is defined as the number of samples beneath that node. For a leaf this is just the number of samples that lies in its occupied space and is equivalent<sup>42</sup> with the number of samples that are stored inside the sample collection of the leaf.

For an inner node things are different, since these kind of nodes have an empty sample container. There it is the mass of all its children that are leafs.

### 11.7.5 Leaf Iterator

It is actually very obscure to provide this class to the user directly in Python, however there are situation where it could come in handy. The name of this class `pyYggdrasil.LeafIterator`, should be self explanatory. It allows the iteration of all leafs beneath a certain node. In PYYGGDRASIL only the tree is able to create one, see section 11.10 on page 21 for more details. Because of that it allows the iteration over all leafs. Note that in C++ offers a more fine grained access mechanism.

This class provides four functions. It can not be used inside a `for` loop, but the `while` loop has to be used to iterate over it.

---

<sup>37</sup>That is unlikely to happen, but is documented here.

<sup>38</sup>And fairly inefficient.

<sup>39</sup>The one without an argument.

<sup>40</sup>It is a pain to write such structure that then actually interact with other objects, however it is worth the time to actually do it.

<sup>41</sup>By this we mean true splits, indirect splits or leafs.

<sup>42</sup>This technically only true for the dirty mass, but since inserting is not supported, the dirty and the normal mass are the same.

**access()** This function returns a reference like object to the leaf the iterator currently points to. The type of the returned object is of type `pyYggdrasil.NodeFacade`, that was already discussed in section 11.7.4 on page 20.

**isEnd()** This function returns a bool. The value is `True` if the end of the range was reached. This has several implications. First of all it can not be incremented more. The second implication is that the iterator does not point to any leaf. Trying to access it will result in undefined behaviour. A `False` indicates that the node can be safely incremented.

**getDepth()** This is the depth of the current leaf. See section 11.7.3 on page 20 for more information.

**nextLeaf()** This function increments the iterator and the head of the iterator will be advanced to the next leaf. Note that the order in which the leafs are visited is not specified and not guaranteed to be static during the lifetime of the tree. It only guaranteed that all leafs will be visited<sup>43</sup>.

Tring to increment an iterator which is at the end, **isEnd()** returns `True` results in undefined behaviour.

## 11.8 Tree Builder

Functions with many arguments are annoying. In Python you have named arguments, but they are only able to cover a certain aspect of the problem. The builder pattern<sup>44</sup> is an extreme convenient way of introducing named arguments to C++, it also provides an extreme convenient and clean solution to our problem.

It is a very simple object which represents the arguments of the constructor. It is implementing such that the functions returns references to the object itself, thus allowing method chaining.

With that object you can set the parameter of your tree.

**Setting the Model Type** Currently there are two models implemented in YGGDRASIL. The linear and the constant model. In order to set model, the following functions are supported `useLinearModel()` and `useConsatntModel()`, which evident meaning.

However it is also possible to use the constructor to set the model. The constructor takes an optional argument. This argument is an `enum` with the name `pyYggdrasil.eParModel`, it defines member constants that are used to encode the model.

**Setting the Significance Level** In order for performing the statistical tests a significance level has to be specified. As default the value is 0.01 used. The level can be adjusted by calling `setGOFLevel(sigLevel)` which sets the significance level of the goodness-of-fit test to `sigLevel`.

The level for the independence test can be adjusted by `setIndepLevel(sigLevel)`.

**Split Strategy** The scheme for the splitting can be set by the following two functions. `useMedianSplitter()` for using the median splitting scheme, also known as score based splitting, this is also the default. The size splitting can be activated by `useSizeSplitting()`.

**Reading the Builder** It is also possible to query the currently set options. For that similar named functions are provided, consult `help()` for a full list.

## 11.9 Note on Extending

The builder class is one of the class that has to be adjusted if the functionality is extended. The C++ level of this class was designed with that in mind. There are more `enums` defined, that are used to encode the test type. This flexibility was not carried over to Python.

## 11.10 Tree

In this section the tree is explained, in Python the tree is known by `pyYggdrasil.DETree`.

---

<sup>43</sup>This guarantee is only valid if the insertion of samples is not implemented or used.

<sup>44</sup>The name is a bit misleading, and it could be confused with the factory pattern, which is also used.

### 11.10.1 Creating a Tree

Creating the tree is rather easy, the constructor has only three arguments. The first step is to create a building object, that was discussed in section 11.8 on page 21.

The tree can be constructed from all three containers that were discussed in section 11.6 on page 16. However in the case of the sample collection a more efficient construction is used. The three arguments are.

**samples** These are the samples that should be loaded into the tree. As it was mentioned all three containers are supported, it is also planned to support NumPy Arrays.

**domain** This is the data space that is used, it can be invalid. If **domain** is valid then it is required and enforced, that the samples lies inside the given domain. If **domain** is invalid the domain is estimated from the given data. Notice that the definition of the interval is also enforced, thus the upper bound is slightly larger than the largest element.

**builder** This is the builder object that was created.

**load** This is an optional argument, that is only present if **samples** is of type `pyYggdrasil.SampleCollection`. By default it is **False**. If it set to **True** then the data of the collection is *moved into* the tree. The argument is then empty.

The constructor will call the fitting process on its own. When the constructor returns the tree is fitted.

### 11.10.2 Steps in the Construction of the Tree

In this section the steps that are done in the constructor are outlined and explained.

1. The first step depends if the passed domain is valid or not.  
In the case of a valid domain, tree checks if the passed samples really lies inside the domain, that was given.  
In the case of an invalid domain, the data is scanned to find a domain. For that the largest and the smallest element in each dimension is searched for. The smallest element is used for the lower bound. Because of the interval definition the largest element can not be used as upper but. Instead the next larger representable number is used<sup>45</sup>.  
The final domain is then stored as *global data space*.
2. The data is rescaled such that it lies in the *rescaled root data space*, which is the unit cube.
3. Then the model on that node is fitted.
4. Then the gof and independence test is performed<sup>46</sup>. If all tests are accepted the leaf is accepted and the process ends.
5. In the case that a test is not accepted the node will be split. For that a split position is searched for, how this is done depends on the chosen strategy.
6. The node is split at that location<sup>47</sup>. We assume that the split happens in dimension  $d$  and at position  $x_s$ . The samples for which we have  $x^{(d)} < x_s$  are sent to the *left* child the other samples are sent to the *right* child.  
The samples are rescaled such that they, again, lies inside the *rescaled node data space*, which is nothing else than a unit cube. Also the domain of the parent node is split in two new *node data spaces* that together completely fill the *node data space* of their parent.  
Then the two childes are tested as well.

### 11.10.3 Operation on Trees

In these section we explain some of the functions that are provided by the tree, not this excludes functions related to the density estimation which are treated in section 11.10.4 on page 23. Again `help()` is your friend.

---

<sup>45</sup>For that `std::nextafter` is used.

<sup>46</sup>The gof test is done first and the independence test is only done if the gof test is accepted.

<sup>47</sup>This implicitly means a dimension and a position.

**Accessing** The tree implements the `__iter__` function. Thus allows the iteration over all leafs in the tree. This is done by using the functionality of the `pyYggdrasil.LeafIterator` that was discussed in section 11.7.5. The user also have to keep in mind, that the operation happens in a read only fashion on `pyYggdrasil.NodeFacade` objects, see section 11.7.4. This happens in a hidden fashion to the user. By calling the `getLeafIterator()` function the user can obtain a leaf iterator and manipulate it directly.

**Get the Data Domain** If the user did not pass a domain, meaning that the domain was estimated from the data, the user does not know what the domain is. The function `getGlobalDataSpace()` returns a copy of the data domain. This is the unscaled original data domain, where the samples lived in.

**Test Integrity** The tree offers some functions that can be used to test the integrity of the constructed tree. Some of them only performs quick checks, such as `isValid()`, others like `checkIntegrity()` or `isFullySplittedTree()` performs some deep and comparable expensive checks.

#### 11.10.4 Density Queries

The whole point of the tree is to be able to have an estimation of the probability density function. For that functions are provided. Again they operate on sample containers, but also a single sample can be estimated.

There are two kinds of functions to query the tree pdf. The difference between them is their behaviour if a sample is encountered, which lies outside the data space.

**It is an Error** The default behaviour is, that the tree consider this as an error. The argument for that is the following. If the user has passed a data domain to the tree, he has some prior knowledge about the generating process. So a sample outside the domain violates this knowledge and should not go on unnoticed. In the case the domain was estimated from the data, saying something that lies *outside* the data is very critical.

**It is not an Error** In some situation it could be justified to allow such queries. Then the pdf value of these samples is set to zero.

**Functions** The tree provides three functions. They are overloaded for the containers and one single samples.

`evaluateSamplesAt(x, beQuiet)` This functions evaluates the pdf of the samples that are stored inside the container  $x$ <sup>48</sup>. The parameter `beQuiet` is a bool, which defaults to `False`. This controls the behaviour of the tree if a sample is encountered that lies outside the data domain. If `beQuiet` is `False` then an error is generated. In the case of `True` the pdf of such samples are set to zero and no error is generated.

`evaluateSamplesAt_err(x)` This function is like the first one, but with `beQuiet` fixed to `False`.

`evaluateSamplesAt_noErr(x)` This function is like the first one, but with `beQuiet` fixed to `True`.

**Return Type** In the case of a single sample a float is returned. In the case of many samples an `Eigen` vector is returned. `pybind11` will convert this vector transparently into a NumPy array without coping<sup>49</sup>.

Note that this is different for the C++ case. There the functions return a `yggdrasil_PDFValueArray_t`, which currently are an alias of `std::vector`. However also the versions which returns an `Eigen` type are available.

### 11.11 PYR

For testing and demonstration purposes `PYYGGDRASIL` provides facilities to generate samples from certain parametric distributions. They reside in the submodule `PYYGGDRASIL.RANDOM`, you can get a complete help by typing `help(pyYggdrasil.random)`.

---

<sup>48</sup>Note that even in the case only a single sample is queried, the function is named that way.

<sup>49</sup>At least the manual claims that.

### 11.11.1 Note on Performance

The distribution are implemented by using standard methods that are provided by **Eigen** and the **C++** standard library. The non trivial ones where tested<sup>50</sup>. The test consists of verifying if the mean and covariance of the generated samples are the same as the theoretic values.

But it is recommended to use *real* implementations, written by people who knows what they are doing. The advantages of the distributions provided by YGGDRASIL itself are, that they provide a very tight integration into the general frame work. So they store the samples directly in the sample containers.

A second note is, that the generators, especially the base class was not designed for speed. They were designed to add new distribution with the least possible amount of code. This means that a concrete distribution must implement, beside its constructor only 5 functions, of which 3 are more or less trivial<sup>51</sup>.

### 11.11.2 Implementation

Here we will shortly describes the implementation of the base class. All distributions inherent from the base class, in C++ this class is known as `::yggdrasil::yggdrasil_randomDistribution_i` and in Python as `pyYR.RNDistribution`. The class is abstract, so it is impossible to create instances. However the base implements the main sampling functions. The derived class only has to implement two hooks, beside some other utility functions, see section 11.11.5 for more details.

`wh_pdf(s)` This function computes the probability of the sample `s`.

`wh_generateSample(g, &s)` This function generates a sample using the generator `g`. Note that the `&` is used to indicate, that the sample is modified and no object is returned.

All 17 other sampling functions are implemented by these two functions. Thus allowing the implementation of a new distribution in a short amount of time.

### 11.11.3 The “Sampling Domain”

The sampling domain is a concept that was introduced to solve a problem in the scaling experiment program. It is a very simple concept, that seems strange in some situations but can come in handy.

The sampling domain is the domain where samples *could* exists. It is important to note that there is no requirement that the probability must be greater than zero everywhere in the domain. In fact the probability can be zero everywhere in that domain<sup>52</sup>.

The functionality for reducing the support of certain distributions is particular interesting if the support is  $\mathbb{R}^n$ , but the probability is rather small after a certain value. This allows to artificially shrink the domain. The canonical distributions are all restricted to a certain domain. However these domain was chosen very large, such that no important features are missing<sup>53</sup>.

There is also an important point to pointed out. Lets assume that the distribution is restricted in a *meaningful* way. Then the base object will sample the distribution until a sample is generated that lies inside the sampling domain<sup>54</sup>. It is recommended that the deriving class does this on its own, this will lead to better performance.

A second point is, that if a sample, that lies outside the sampling domain, the pdf value is set to zero automatically. The actual probability for that sample at that point does not matter at all, only the fact that the sample lies outside the domain is important. This is done by the base object itself, so concrete distribution can not influence that<sup>55</sup>.

A last remark is, that the pdf value must not be adapted. A pdf function must integrating to one. But when restricting the support to a smaller set than the actual support, then this is not true any more. This inconsistency is ignored.

**Setting the Domain** There is only one way to set the sampling domain. This is at construction. All constructors takes an optional last argument, which is called `domain`<sup>56</sup>.

If the domain is omitted a distribution specific default value is used. In the case of infinite support, no restriction is applied. In the case of a finite support, the sampling domain is set to the smallest set<sup>57</sup> that contains the analytic support, that can be encoded by a single hyper cube.

<sup>50</sup>As far at is goes, the uniform distribution was not tested, since it is just a wrapper around the standard function.

<sup>51</sup>The interface could be removed even more, but it evolved.

<sup>52</sup>Note that this distribution is very useless, since no sample will ever be generated.

<sup>53</sup>At least we hope that.

<sup>54</sup>There is a protection if no progress was made.

<sup>55</sup>The only possibility to do that is to override all pdf functions.

<sup>56</sup>The name is only useful in Python.

<sup>57</sup>YGGDRASIL only supports axis aligned domains.



**An Unbounded Domain** It is important that for several reasons, an unbounded domain is not encoded by a domain that has `inf` as bounds<sup>58</sup>. A sampling domain that should be unbounded, must be the invalid domain.

#### 11.11.4 Explicit Random Number Generator Needed

The distribution objects that are provided by Python and especially SciPy do not need a (pseudo) random number generator passed to the distribution. They use some global object that is provided *somewhere*. We consider this, global variables, as ugly and dangerous. Thus all functions that needs a source of randomness, will need a generator passed to them as an argument. The name of that argument is always `g`. For that `PYYGGDRASIL` provides the object `pyYggdrasil.Random.pRNG`. This is a very thin wrapper around a `std::mt19937_64` object, which is the C++ implementation of the Mersenne Twister algorithm.

#### 11.11.5 Functions of the Distributions

The base of all distributions, `pyYR.RNDistribution`, implements the main functions directly, as it was explained before. Generally all functions are overloaded for a single sample as well as all sample containers.

**pdf(s) -- single sample** This function calculates the probability for a single sample. It returns a floating point value<sup>59</sup>.

**pdf(s) -- sample containers** This function calculates the pdf values for all samples that are stored inside the sample container `s`<sup>60</sup>. The function returns an vector of type `Eigen::VectorXd`, `pybind11` can transparently convert this type into a NumPy array without the need of coping it. This is different from C++ where a `PDFValueArray_t` is returned, which is currently an alias of `std::vector`. However the `Eigen` versions are also available on C++.

**generateSamples(g)** This function generates a single sample form the underling distribution.

**generateSamplesCONTAINER(g, N)** This function generates N samples. The samples are stored and returned in a sample container.

The type of the container must be selected by the function name. You must substitute `CONTAINER` by `List`, `Array` or `Collection`, the meaning should be clear.

**generateSamples{CONTAINER}PDF(g, N)** This function is similar to the one above. It will also generate N samples and returns an appropriate container, that can be selected by substituting `CONTAINER` as described above.

However this functions return actually a pair. The first member of that pair is the requested sample container, filled with the samples. The second member of that pair, is a vector, also an `Eigen` compatible type, that can be converted to a NumPy array. This vector contains the probability of the samples that where generated.

**getMean()** This function returns the theoretical mean of the distribution. As a type the function returns an `Eigen` vector.

This function has to be implemented by the deriving class.

**getCoVar()** This function returns the theoretical covariance of the distribution. It returns an `Eigen` matrix.

This function has to be implemented by the deriving class.

**getSamplingDomain()** This function returns a *copy* of the sampling domain of the distribution.

This function has to be implemented by the deriving class.

**isUnbounded()** This function returns `True` if the sampling is not restricted by a domain. This also implies that the domain is invalid.

By default this function uses the `getSamplingDomain()` function to obtain the domain and then testing if it is invalid. This is rather costly and it is recommended to override this function. The base class is designed to minimize calls to both of them.

**Note on the Sampling Functions** The sampling functions made a unusual guarantee. All samples that are generated will have a probability greater than zero. This sounds rather strange, since a sample with probability zero, will never be generated. However this guarantee means *finite* precision, where cancellation can occur.

As a side note the generation of a pdf value that is `nan` is considered an error. However also an infinite pdf value is considered an error.

<sup>58</sup>By a quirk in the code this might actually work, but it is not the supported way. Doing it is undefined behaviour.

<sup>59</sup>In C++ this is `::yggdrasil::Real_t` which is an alias of `long double`.

<sup>60</sup>The argument is named `s` in both case single sample or sample container.

### 11.11.6 Concrete Distributions

Here we list the concrete distributions that are implemented. They all supports two constructor, one with sample domain and one without. As it was explained in section 11.11.3 on page 24, if the domain is not specified *meaningful* defaults are applied. In the following we will only list the constructor with the domain argument.

The concrete implementation also implements some additional functions. Primarily they are used to access the parameter of the distribution. See `help()` for more information.

When we write `Eigen` type or something similar, then this is equivalent to a corresponding NumPy type.

**Gauss Distribution** This is a normal distribution in  $n$  dimensions. In Python this class is named `pyYggdrasil.Random.Gauss`. The constructor reads:

`Gauss(mu, Sigma, domain)`

**mu** This is the mean of the distribution. In C++ this is an `Eigen` vector type. It must have length  $n$ .

**Sigma** This is the covariance matrix, this is the generalization of the variance to many dimension. It must be an  $n \times n$  `Eigen` matrix.

**domain** This is the sampling domain. If the domain is omitted the invalid distribution is used. This means no restriction is applied.

**Dirichlet Distribution** The class `pyYggdrasil.Random.Dirichlet` models the Dirichlet distribution. For the Dirichlet distribution there are many conventions used. We follow the one that is also used in the paper and in Mathematica, this differs from Wikipedia.

The  $k$  dimensional Dirichlet distribution, is described by  $k + 1$  parameters  $\vec{\alpha}$ , for which we have  $\alpha_i > 0$ . From the sample vector  $\vec{x}$  we require  $x_i > 0$  for all  $i = 1, \dots, k$  and  $\sum_{i=1}^k x_i < 1$ . Its probability density function is given by

$$\frac{\Gamma\left(\sum_{i=1}^{k+1} \alpha_i\right)}{\prod_{i=1}^{k+1} \Gamma(\alpha_i)} \cdot \left(1 - \sum_{i=1}^k x_i\right)^{\alpha_{k+1}-1} \cdot \prod_{i=1}^k x_i^{\alpha_i-1}$$

It has the following constructor:

`Dirichlet(alpha, domain)`

The arguments are:

**alpha** An `Eigen` vector of length  $k + 1$ . These are the parameters that are used to describe the distribution.

**domain** This is a hyper cube of dimension  $k$ . If the domain is omitted the unit hyper cube is used.

**Uniform Distribution** YGGDRASIL also provides a multidimensional uniform distribution, thy Python name is `pyYggdrasil.Random.Uniform`. The dimension of the distribution are independent from each other. It provides the following constructor:

`Uniform(suppDom, domain)`

The arguments are:

**suppDom** This is an hyper cube of dimension  $k$ . It describes where samples are generated.

**domain** For this distribution the sample domain is a bit off. Note that there are no restriction on the domain. If none is specified `domain` is equal `suppDom`.

**Uniform Distribution** This is the outlier distribution, in Python `pyYggdrasil.Random.Outlier`. It is basically a bimodal Gaussian distribution where both modes have the same mean. In a short hand form this distribution reads as:

$$\alpha \mathcal{N}(\mu, \sigma_1^2) + (1 - \alpha) \mathcal{N}(\mu, \sigma_2^2)$$

The constructor of this is:

`Outlier(alpha, mean, sigma1, sigma2, domain)`

**alpha** This is the strength of the first mode. It must be in the range  $0 < \alpha < 1$ .

**mean** This is the mean of *both* modes.

**sigma1** This is the *standard deviation* of the first mode. Note that this differs from the Gauss distribution where the variance is used.

**sigma2** This is the *standard deviation* of the second mode.

**domain** This is the sampling domain. Note that even this is a one dimensional distribution, the domain must be a hyper cube object, but of dimension one. As default an invalid cube is used.

**Bimodal Uniform Distribution** This is a bimodal uniform distribution, `pyYggdrasil.Random.BiUniform`. In the paper the term spiky uniform is used. It can be written in the following way

$$\alpha \text{Uni}(a_1, b_1) + (1 - \alpha) \text{Uni}(a_2, b_2) \quad (1)$$

The constructor reads as:

`BiUniform(suppDom1, suppDom2, alpha, domain)`

With the arguments

**suppDom1** This the support range of the first mode,  $\text{Uni}(a_1, b_1)$ . It is of type `pyYggdrasil.Interval`.

**suppDom2** This the support range of the first mode,  $\text{Uni}(a_2, b_2)$ . It is of type `pyYggdrasil.Interval`.

**alpha** This is the coupling constant of the mode, or the strength of the first mode.

**domain** This is the support domain. For consistency it must be a hyper cube of dimension one. If no domain is used, an invalid cube is used.

**Gamma Distribution** This is the gamma distribution, it is implemented by `pyYggdrasil.Random.Gamma`. Since there are more than one convention that is in use, we will give the PDF explicitly.

$$\Gamma(\alpha, \beta) \sim \frac{1}{\Gamma(\alpha)\beta^\alpha} x^{\alpha-1} e^{-\frac{x}{\beta}}$$

The constructor is given as:

`Gamma(Alpha, Beta, domain)`

**Alpha** This is the shape parameter of the distribution. It must be greater than zero.

**Beta** This is the rate parameter of the distribution. It must be greater than zero.

**domain** This is the sampling domain of the distribution. It must be a hyper cube of dimension one. If it is invalid no restriction is applied.

**Beta Distribution** This is the beta distribution, `pyYggdrasil.Random.Beta`. The pdf is given as:

$$\text{Beta}(\alpha, \beta) \sim \frac{x^{\alpha-1} \cdot (1-x)^{\beta-1}}{B(\alpha, \beta)}$$

$$B(\alpha, \beta) := \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha + \beta)}$$

Its constructor reads as:

`Beta(Alpha, Beta, domain)`

The parameters are:

**Alpha** This is the shape parameter, must be greater than zero.

**Beta** This is also called shape parameter<sup>61</sup>, it must also be greater than zero.

**domain** This must be a hyper cube. If the domain is not given the unit interval is used.

---

<sup>61</sup>According to Wikipedia.

**Multimodal Distribution** This is an application of the composite pattern. It is not a distribution by itself, it is the (linear) combination of several distributions. Assume you have  $N$  different distribution that you want to combine with each other. The weight of each mode,  $P_k$ , is given by  $N - 1$  weights,  $\omega_i$ , with  $\sum_{i=1}^{N-1} \omega_i < 1$ . The last weight is implicitly given as  $1 - \sum_{i=1}^{N-1} \omega_i$ . Then the new distribution can be written as:

$$P \sim \sum_{i=1}^{N-1} \omega_i P_i + \left(1 - \sum_{i=1}^{N-1} \omega_i\right) P_N$$

Due to some circumstances this distribution does not exposes any constructors to Python. The only way to construct multimodal distributions, is to use the provided helper functions `createMultiModal(...)`. Use `help(pyYggdrasil.Random.createMultiModal)` to get a list of the functions. They will create a distribution of type `pyYggdrasil.Random.MultiModal`.

A small note on the domain. If the domain is omitted it will be tried to estimate it form the passed distributions, by merging the corresponding domains. If at least one of them is invalid, the resulting domain will be invalid as well and the distribution will be unbounded.

## 11.12 Tree Sampler

Beside the `RANDOM` sub module of `PYYGDRASIL`, does not only contain distributions, it also contains a distribution that takes a tree as underling distribution. This object is called `pyYggdrasil.Random.TreeSampler`. This object is similar to the distributions that are presented in section 11.11 on page 23, but is a bit different.

Because of some implementation details the sampler is not able to compute the pdfs of stand alone samples. The sampler can compute the probability of sample that was sampled by it as a by product, but it is not possible that a sample is passed to it and its probability is calculated<sup>62</sup>. Thus the `pdf(s)` functions that are described in section 11.11.5 on page 25 are not supported, as well as the `getMean()` and `getCoVar()` functions that are also not supported. The rest of the listed functions is implemented.

### 11.12.1 Conditions

As with any other main concept that surfaces in `YGGDRASIL` there is a class that models it. In `PYYGDRASIL` this class is available as `pyYggdrasil.Random.MultiCondition`.

It is build around the class `pyYggdrasil.Random.SingleCondition` that is used to model a condition in a single dimension. It is basically a wrapper around an integer and a double. The integer is the dimension in which the condition should be applied<sup>63</sup>. The double represents the value samples should have.

A `MultiCondition` can be seen as a list of `SingleConditions`. It is also an important note that *every* `MultiCondition` belongs to a sample space and thus needs its dimension. The class supports many way of constructing it.

**ctr(nDims)** This construct an empty `MultiCondition` this means no conditions that are applied. As we have said every `MultiCondition` is associated to an underlying sample space, and thus the dimension of this space must be passed to it to. In this case the sample is set to `nDims`.

**ctr(nDims, cVec)** This is a constructor that associates the condition to a sample space of dimension `nDims`. The conditions that are applied are extracted from `cVec`, this is a list<sup>64</sup>. The order inside the list is not important.

**ctr(nDims, eMap)** This constructor binds the condition to a sample space of dimension `nDims`. The conditions are extracted from the `dict` object. The dictionary is interpreted as a map from integers<sup>65</sup> to doubles<sup>66</sup>. The keys are interpreted as the dimension and the associated value is seen as condition that should be applied.

**ctr(src)** Also a copy construct is provided that performs a deep coping of the condition.

<sup>62</sup>Technically it is possible, but it is extremely inefficient, so it is not implemented.

<sup>63</sup>`YGGDRASIL` and `PYYGDRASIL` use zero based indexing, this means the first dimension 0 and not 1.

<sup>64</sup>A `std::vector` is used, this means a python list and tuples can be passed to it. Under the condition that all elements are of type `SingleCondition`.

<sup>65</sup>The keys.

<sup>66</sup>The associated values.

### 11.12.2 Constructing a Sampler

Constructing a sampler is quite forward. It must be provided with a tree and condition that should be applied, there are several constructor that allows to bypass the construction of an explicit condition object, see `help()` for more. If no condition is supplied then an unconditioned sampler is generated. It is important that regardless of the form the conditions are supplied they are interpreted on the original data space. All rescaling that is needed is done by YGGDRASIL under the hood.

The step of the constructing a sampler are roughly as follows. The sampler iterates over all leafs of the tree, for that the leaf iterator, that was already presented in section 11.7.5 on page 20 is used. Then each leaf is processed.

1. First it is tested if the leaf satisfy the conditions. This is done by testing if all conditions lies inside the domain<sup>67</sup> that is occupied by the leaf.
2. If this is the case the leaf joins the selected leafs, we use the symbol  $\mathcal{L}$  to denote them. Note that the leaf is partially copied into a proxy object<sup>68</sup>. This proxy stores a copy of the model and the domain and some other auxiliary object, that allows it to mimic a node for the parametric model. This coping allows the decoupling of the sampler and the tree. This means that the sampler can then exists on its own<sup>69</sup>.
3. Then the probability weight of the leaf is calculated. According to the paper<sup>70</sup> this weight is given by

$$\mathcal{M}_k := \frac{1}{p(\vec{x}^c)} \cdot \frac{n(C_k)}{n_t} \cdot \prod_{i=q+1}^d p[x_i \mid \vec{\theta}_i(C_k)]$$

The components  $q + 1$  up to  $d$  were the one where a condition was applied on. This is done out of convenience and bears no real meaning. YGGDRASIL allows that any dimension can be conditioned on. The first factor of  $\mathcal{M}_k$ ,  $\frac{1}{p(\vec{x}^c)}$  does not depend on the leaf and is thus the same for all leafs. For the computation of the weights it is ignored. However this factor is quite important for calculating conditioned probability.

The second factor is a normalization constant for the *probability density functions of the leafs*.  $n(C_k)$  is the number of samples that are inside leaf  $k$ .  $n_t$  is the total number of samples in the *whole* tree.

The last factor is a product of probabilities. This factor is only present in the case of conditions, in the case of an unconditioned sampler, it is just one. Actually the probabilities should be expressed on the global data space. However due to some internal design aspects of YGGDRASIL the probability on the *root data space* is used. This is not a problem since the transformation factor that must be applied is a multiplicative constant that is the same for all leafs and will cancel out.

**Calculating of  $\frac{1}{p(\vec{x}^c)}$**  In this section we will explain how the factor is computed and used. This factor is needed for obtaining the probabilities in the conditioned case.

The calculation of this factor happens after the weights, remember when we compute the weights we omit some constant multiplicative terms. They are not important for the weights, since only their relative weights are needed, but they are important for calculating this factor.

The first step is that they are summed up.

Then we have to include the factors that we have omitted. The only factor that we have to include is the transformation factor that transforms the probability from the root data space to the global space. It is important that this factor is not the inverse volume of the global data space. It is the inverse product of the length of all the dimensions that we conditioning on.

### 11.12.3 Generate Samples

As it was explained in the beginning of section 11.12 on 28 the sampler mimics the functionality of a random distribution, with some limitations. Here we will explain how the sampling process works under the hood. It is a two step process, first the leaf is selected and then a sample is drawn from that leaf.

1. The probability that a leaf is selected is proportional to its probability mass  $\mathcal{M}_k$  that was discussed in section 11.12.2 on page 29. This is implemented by using `std::discrete_distribution` that allows to sample from this.

<sup>67</sup>For that a variation of the `isInside` function of the `HyperCube` is used, that is not available in `pyYGGDRASIL`.

<sup>68</sup>The type of the proxy is `ygInternal_leafProxy_t`, but is only available on the C++ level.

<sup>69</sup>Howver there is this limitation about pdf queries.

<sup>70</sup>See equation (6).

2. We then generate a sample on the selected leaf. For that a function of the parametric model interface is used. As argument the function gets the domain and the conditions, that where transformed to the *rescaled node data space*. Also a source of randomness must be passed. This is a sample with components that where sampled uniformly on the unit interval. The intent is that the inversion method can be applied, but it is not required to do that.
3. It is then tested if the probability of the generated sample is slightly negative or zero. This can happen because of the finite arithmetic of the doubles. If this happens the sample is rejected and the process is repeated.

#### 11.12.4 Note On Returned Probability

Earlier versions of YGGDRASIL always computed the *unconditioned* probability. At some point this was changed, however some parts of the documentation might not reflect that change.

So the functions, those who actually return probability values, return conditioned ones.

## 12 Canonical Distributions

YGGDRASIL uses some canonical distribution. They are characterized by a name, the name of the distribution<sup>71</sup>, and a number. The number corresponds in a sense to the dimension, if positive, when the number is negative, it has some special meaning.

### 12.1 Gauss Distribution

This are the Normal or Gauss distributions. Here we will list the known ones.

**Kind: -2** Here we have a two dimensional distribution. This distribution comes from the paper and is there introduced in section 3.2.1.

$$\vec{\mu} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \quad \Sigma = \begin{pmatrix} 4 & -2.28 \\ -2.28 & 1.14 \end{pmatrix}$$

The sampling domain is restricted to the domain  $[-8, 8]^2$ .  
If conditions are applied we get

$$\Sigma|_{x_1=0} = \begin{pmatrix} 0 & 0 \\ 0 & 0.1404 \end{pmatrix} \quad \Sigma|_{x_1=0} = \begin{pmatrix} 0.39 & 0 \\ 0 & 0 \end{pmatrix}$$

**Kind: -22** Here we have a two dimensional distribution. This distribution is based on kind -2.

$$\vec{\mu} = \begin{pmatrix} 0.5 \\ -0.5 \end{pmatrix} \quad \Sigma = \begin{pmatrix} 4 & -2.28 \\ -2.28 & 1.14 \end{pmatrix}$$

The sampling domain is restricted to the domain  $[-8, 8]^2$ .  
If conditions are applied we get

$$\Sigma|_{x_1=0.5} = \begin{pmatrix} 0 & 0 \\ 0 & 0.1404 \end{pmatrix} \quad \Sigma|_{x_2=-0.5} = \begin{pmatrix} 0.39 & 0 \\ 0 & 0 \end{pmatrix} \quad \Sigma|_{x_1=0} = \begin{pmatrix} 0.221625 & 0 \\ 0 & 0 \end{pmatrix}$$

**Kind: 2** This is also a two dimensional gauss distribution. The distribution does not come from the paper. It has the following parameters

$$\vec{\mu} = \begin{pmatrix} 1.4 \\ -4.0 \end{pmatrix} \quad \Sigma = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

The sampling domain is restricted to the domain  $[-10, 10]^2$ .

**Kind: 3** This distribution comes from the paper about the random generator. It is a 3 dimensional gauss distribution, with the following properties

$$\vec{\mu} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \quad \Sigma = \begin{pmatrix} 0.35 & 0.25 & 0.5 \\ 0.25 & 0.4 & 0.6 \\ 0.5 & 0.6 & 1 \end{pmatrix}$$

The sampling domain is restricted to the domain  $[-10, 10]^2$ .  
Under certain restricting we have

$$\Sigma|_{x_2=0} = \begin{pmatrix} 0.19375 & 0 & 0.125 \\ 0 & 0 & 0 \\ 0.125 & 0 & 0.1 \end{pmatrix} \quad \Sigma|_{x_2=0, x_3=0} = \begin{pmatrix} 0.0375 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

**Kind: 4** This is a four dimensional Gauss distribution. It is also from the paper, where it is introduced in section 3.4.1.

The sampling domain is restricted to  $[-15, 15]^4$ .

---

<sup>71</sup>In the code they are modelled by the enum `eDistribType`.

**Kind: 7** This is a seven dimensional Gauss distribution. It is also from the paper, where it is introduced in section 3.4.1. The sampling domain is restricted to  $[-15, 15]^7$ .

## 12.2 Dirichlet Distribution

For the Dirichlet distribution there are two different convention are in use. The first is used by Wikipedia, the second is used by the paper. YGGDRASIL follows the convention of the paper<sup>72</sup>. All of the canonical distribution are restricted to the unit hyper cube.

**Kind: 2** This is a distribution from the paper, it is introduced in section 3.2.3. Its parameters are

$$\vec{\alpha} = \begin{pmatrix} 0.9 \\ 1.5 \\ 3 \end{pmatrix}$$

**Kind: 4** This is a four dimensional Dirichlet distribution. It is also introduced in the paper in section 3.4.2. Its parameter are

$$\vec{\alpha} = (6.13 \quad 9.29 \quad 10.6 \quad 8.24 \quad 3.91)$$

**Kind: 7** This is a seven dimensional Dirichlet distribution. It is also from the paper, where it is introduced in section 3.4.2.

## 12.3 Outlier Distribution

This is basically a super position of two Gaussian The Distribution is characterized by

$$\text{Out}(\alpha, \mu, \sigma_1, \sigma_2) = \alpha \mathcal{N}(\mu, \sigma_1^2) + (1 - \alpha) \mathcal{N}(\mu, \sigma_2^2)$$

**Kind: 0** This distribution is introduced in section 3.1.2 of the paper. It has the following parameters.

$$\alpha = \frac{1}{10} \quad \mu = 0 \quad \sigma_1 = 1 \quad \sigma_2 = \frac{1}{10}$$

## 12.4 Uniform Distribution

This is the uniform distribution or related.

**Kind:  $>0$  &  $\leq 100$**  In this is a multi dimensional uniform distribution. The different dimensions are independent. In each dimension one has the following distribution

$$X_i \sim \text{Uni}(0.25, 0.75)$$

*The sampling domain is restricted to the unit hyper cube of the corresponding dimension.*

See also section 13.2.1 on page 34 if you observe slow convergence for this distributions.

**Kind:  $>1000$**  This is also a multidimensional uniform distribution. The different dimensions are independent from each other. In each dimension one has the following distribution

$$X_i \sim \text{Uni}(0.25, 0.75)$$

The number of dimensions is calculated by **kind** – 1000.

The main differences between the other uniform distribution is that here the sampling domain not restricted to the unit hyper cube, but the domain where the probability is greater than zero.

**Kind: -1** This is a distribution which is known as “spiky uniform” and was also used in the paper section 3.1.4. The sample domain is restricted to the unit interval.

---

<sup>72</sup>In an earlier version YGGDRASIL followed Wikipedia, but this was changed.



**Kind: -2** This is a different version of the spiky uniform distribution. With the notation used in section 3.1.4, the distribution can be characterized by:

$$\frac{1}{10} \text{Uni}(0.25, 0.5) + \frac{9}{10} \text{Uni}(0.7, 0.8)$$

The sampling domain is restricted to the unit interval.

## 12.5 Gamma Distribution

This is also a classical distribution. The distribution is a one dimensional distribution. It is parametrized by two parameters  $\alpha$  the shape and  $\beta$  called the rate<sup>73</sup>.

**Kind: -1** This is a distribution from the paper, see section 3.1.6. Its parameters are

$$\alpha = \frac{2}{3} \quad \beta = 50$$

The sampling domain is restricted to the interval  $[0.001, 300[$ .

**Kind: -2** This is essentially the same as kind  $-1$ . The reason for its existence is, that kind  $-1$  does not pass the statistical test, since its sampling range is restricted too much. So this one provides a larger sampling domain.

**Kind: -3** This is a different gamma distribution. The parameters are

$$\alpha = 1 \quad \beta = 2$$

The sampling domain is restricted to the interval  $[0.001, 300[$ .

## 12.6 Beta Distribution

This is the beta distribution. Controlled by two parameters  $\alpha$  and  $\beta$ , which are both called shape<sup>74</sup>. The distributions are only defined in the interval  $]0, 1[$ , so we have restricted all distribution to that domain. They are mostly taken from Wikipedia.

**Kind: 1** This is the Beta distribution taken from the paper, see section 3.1.5. The parameters are

$$\alpha = 1.05 \quad \beta = 0.8$$

**Kind: -1** This is a distribution from Wikipedia, the parameters are:

$$\alpha = 0.5 \quad \beta = 0.5$$

**Kind: -2** This is a distribution from Wikipedia, the parameters are:

$$\alpha = 2 \quad \beta = 5$$

**Kind: -3** This is a distribution from Wikipedia, the parameters are:

$$\alpha = 1 \quad \beta = 3$$

---

<sup>73</sup>Notice that there is a second convention, we use the convention used by the C++ standard.

<sup>74</sup>According to Wikipedia.

## 13 Known Bugs and Strange Behaviour

This section lists all known bugs, but also strange behaviour that can be observed.

### 13.1 Bugs

Currently no known bugs.

### 13.2 Strange or Unusual Behaviour

#### 13.2.1 Scaling Test Shows Slow Convergence if Median Splitter is Used

It can happen that the scaling test shows very bad convergence for certain kind of distributions, especially the uniform kind of the canonical distribution, with a kind parameter less than 1000, when the median splitter is used. However if the constant size splitter<sup>75</sup> is used the convergence is superior.

This effect is known and is not a bug, but an effect of the domain. The effect can be best understood in a one dimensional case. The main cause of this effect is that the “zero probability parts” of the domain, can not be eliminated, since always sample are “partially there”.

To fix estimate the domain from the data, this is done by passing an invalid hypercube to the constructor of the tree. A second approach is to shrink the zero probability range.

#### 13.2.2 Probability of the Tree Sampler

Previously the probability that was returned by the tree sampler was *always* unconditional, even in the case of a conditioned tree sampler. The first “fix” was to document it as a strange behaviour. However it was later found that this was not very good solution, so the behaviour was changed. It is possible that at some location of the manual, code or the auxiliary material this change was not made, since the location was forgotten. It was tried to change all location but there is no guarantee that it happened.

#### 13.2.3 Iterating in Python

In Python a class can implement the special function `__iter__` which allows the iteration over its members. Technically the majority of the classes in PYYGGDRASIL would support such a feature and does so but only in C++. The reason is that the behaviour in Python is rather unintuitive and wrong as far as I can tell. Part of the problem is the fact that in Python there are no references like in C++. Assigning something to an iterator had no effect and did not produced an error.

But a lot of classes that are provided by PYYGGDRASIL implements the iteration function. In the beginning they did not implement the function, but it was possible to iterate over them and no error occurred. I do not really understand it why this was possible. So it was decided that the easiest fix it, was to implement the function and generate an exception when called.

---

<sup>75</sup>which cut the dimension in half, geometrically.