korali

CSELab

Dec 08, 2022

USING KORALI

High-performance framework for uncertainty quantification, optimization and reinforcement learning.

Korali is a high-performance framework for Bayesian UQ, optimization, and reinforcement learning. Korali's multi-language interface allows the execution of any type of computational model, either sequential or distributed (MPI), C++ or Python, and even pre-compiled/legacy applications. Korali's execution engine enables scalable sampling on large-scale HPC systems.

Korali provides a simple interface that allows users to easily describe statistical / deep learning problems and choose the algorithms to solve them. The framework can easily be extended to describe new problems or test new experimental algorithms on existing problems.

For more information, read: S. Martin, D. Waelchli, G. Arampatzis, A. E. Economides and P. Karnakov, P. Koumoutsakos, "Korali: Efficient and Scalable Software Framework for Bayesian Uncertainty Quantification and Stochastic Optimization". arXiv 2005.13457. Zurich, Switzerland, March 2021. [PDF].

Usage

Run with Docker: `docker run -it cselab/korali:latest`

Documentation: https://korali.readthedocs.io/

Website: https://www.cse-lab.ethz.ch/korali/

Contact us

The Korali Project is developed and maintained by

- Sergio Miguel Martin, martiser at ethz.ch
- Daniel Waelchli, wadaniel at ethz.ch
- Georgios Arampatzis, garampat at ethz.ch
- Pascal Weber, webepasc at ethz.ch

Frequent contributors: Fabian Wermelinger, Lucas Amoudrouz, Ivica Kicic

CHAPTER

ONE

USER MANUAL

1.1 Installation

1.1.1 Docker

The easiest way to use Korali is to launch it's pre-built Docker container which provides Korali with all its dependencies already installed and configured. To launch the docker container, run:

```
docker run -it cselab/korali:latest
```

1.1.2 Manual Installation

Korali has been thoroughly tested on Linux (Ubuntu, Fedora) systems. Although it is possible to compile and run Korali on MacOS, we strongly recommend users to use the Docker image instead. Korali is not yet supported on Windows systems.

Below, we list the system requirements and steps to install Korali:

System Requirements

- **C++ Compiler** Korali requires a C++ compiler that supports the C++17 standard (*-std=c++17*) to build. **Hint:** Check the following link to verify whether your compiler supports C++17. Korali's installer will check the **$CXX** environment variable to determine the default C++ compiler. You can change the value of this variable to define a custom C++ compiler.
- **Git Client** You need Git to clone (download) our code before installation.
- **meson** To generate the installation configuration.
- **ninja** To build Korali.
- **Python >=3.8** Korali requires a version of Python higher than 3.8 to be installed in the system. Korali's installer will check the *python3* command. The path to this command should be present in the $PATH environment variable. *Hint:* Make sure Python3 is correctly installed or its module loaded before configuring Korali.

Installation Steps

1. Download Korali

   Download Korali with the following command:

   ```
   git clone https://github.com/cselab/korali.git
   ```

2. Setup Installation

   To set up the compilation and installation, run:

   ```
   cd korali
   meson setup build --buildtype=release --prefix=PREFIX
   ```

where `PREFIX` is the absolute path where Korali will be installed. For example, use `$HOME/.local/` to install it in the same folder where `pip` installs packages (this can be verified with `python3 -m sysconfig | grep userbase`). Optionally you can add optional parameters (adding support for MPI, OneDNN, and CUDNN,..). A full list of installation options can be found in meson_options.txt. For more information, see *Optional Requirements* below.

3. Build and Install

   To build and install Korali, run:

   ```
   meson install -C build
   ```

To uninstall Korali, run `cd build && ninja uninstall` or manually delete the folder containing the `korali` package.

5. Setup environment

   The `LD_LIBRARY_PATH` and `PYTHONPATH` environment variables need to be correctly setup for the linker to find the correct libraries at the moment of runtime. They can be setup by using

   ```
   export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:PREFIX/lib64
   export PYTHONPATH=${PYTHONPATH}:PREFIX/lib/python3.8/site-packages/
   ```

Troubleshooting

If you are experiencing problems installing or running Korali, please check the following hints:

- Check Korali's *system requirements* to verify that your system has all the required software components.
- Check the build status to see if Korali is currently building correctly.
- If the problem persists, please submit a new issue report on our Github repository detailing the problem, your system information, and the steps to replicate it and we will promptly address it.
- For further questions, feel free to contact us.

Cray systems (Piz Daint)

Cray systems use a propietary build system that may cause conflicts with the default meson configuration when using MPI. To fix this, the following steps are recommended:

1) Specify the *cc* and *CC* commands as default C and C++ compilers, respectively:

```
CC=cc CXX=CC meson setup build --buildtype=release --prefix=PREFIX
```

2) It is possible tat the the default installation of *mpi4py* possibly uses a different MPI implementation than Korali, preventing multi-rank runs. To fix it, configure MPI compilers and reinstall *mpi4py* and Korali.

```
# Create wrappers `mpicc` and `mpic++` around Cray compilers `cc` and `CC`,␣
↪respectively.
# Warning: this will overwrite any `mpicc` and `mpic++` in your `~/bin` folder!
mkdir -p $HOME/bin
echo -e '#!/bin/bash'"\n"'cc "$@"' > $HOME/bin/mpicc
echo -e '#!/bin/bash'"\n"'CC "$@"' > $HOME/bin/mpic++
chmod +x $HOME/bin/mpicc $HOME/bin/mpic++

# Load Python module (you can add this to your `~/.bashrc`).
module load cray-python

# Clear cache and reinstall mpi4py locally
python -m pip cache remove mpi4py
python3 -m pip install --user mpi4py --ignore-installed -v
```

Optional Requirements

- **oneDNN** Korali uses the OneAPI Deep Neural Network Library for deep learning modules, and is disabled by default. You can enable it by adding the `-Donednn=true` option on the meson configuration line. To recommended configuration for oneDNN is:

```
  wget https://github.com/oneapi-src/oneDNN/archive/refs/tags/v2.7.tar.gz -O oneDNN-
↪v2.7.tar.gz; \
  tar -xzvf oneDNN-v2.7.tar.gz; \
  mkdir -p "oneDNN-2.7/build"; \
  cd "oneDNN-2.7/build"; \
  CXXFLAGS=-O3 cmake .. \
   -DCMAKE_INSTALL_PREFIX=$HOME/.local \
   -DONEDNN_BUILD_EXAMPLES=OFF \
   -DONEDNN_BUILD_TESTS=OFF \
   -DONEDNN_ENABLE_CONCURRENT_EXEC=ON \
   -DONEDNN_ARCH_OPT_FLAGS='-march=native -mtune=native' \
   -DBUILD_SHARED_LIBS=true; make -j8; make install

- **CMake**
    Korali requires that you have `CMake <https://cmake.org/>`_ version 3.0 or higher␣
↪installed in your system if you need it to install certain external libraries␣
↪automatically.

- **MPI**
    One way to enable support distributed conduits and computational models is to␣
↪configure Korali to compile with an MPI compiler. The installer will check the *
↪$MPICXX* environment variable to determine a valid MPI C++ compiler.
```

(continues on next page)

(continued from previous page)

```
- **MPI4Py**
    If you need to run Python-based MPI application as computational models in Korali,
↪ you will need to install the MPI4py python module, and install Korali via meson␣
↪using the `-Dmpi4py=true` option.
```

1.2 Korali Basics

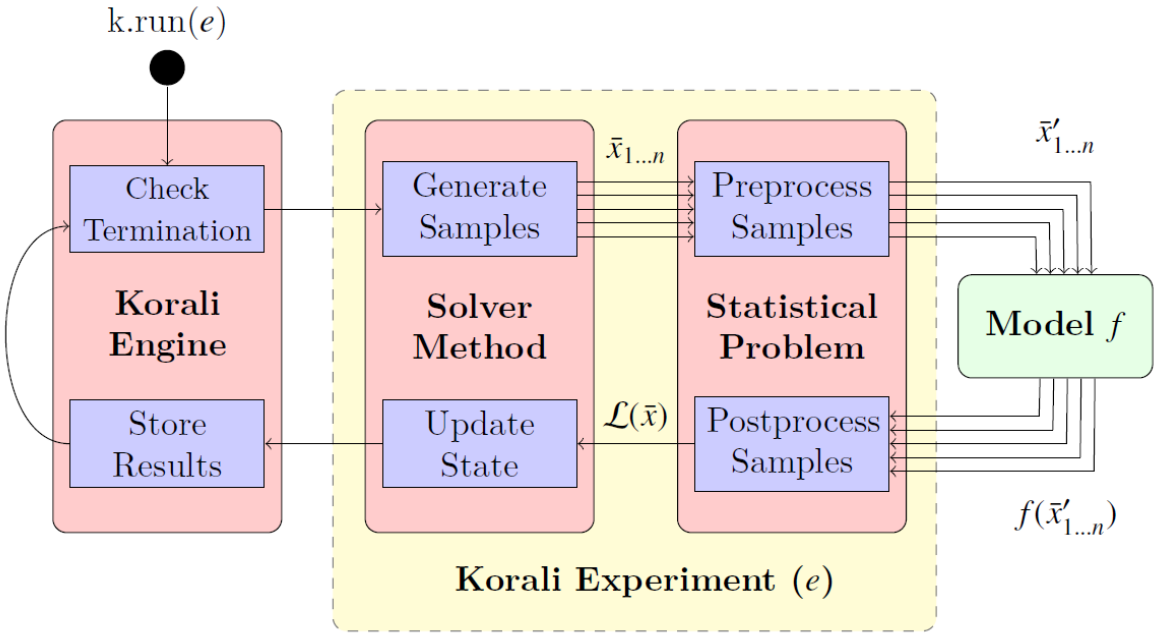
This document describes the basic concepts that make a Korali application.

1.2.1 Creating a Korali Experiment

Korali provides a range of optimization and uncertainty quantification tools. To use these tools, a user needs to create a *Korali Experiment*. The following python code snippet shows how to load Korali's library and instantiate a new experiment:

```
import korali
e = korali.Experiment()
```

This creates a new experiment object which can be then configured by specifying a Korali *problem* and *solver* module combination, as shown below.



The solver represents a specific algorithm or method find the solution of the specified problem. Most problem types require the evaluation of a *computational model* (function) to evaluate *samples*, whose results are returned to the Korali engine.

The cycle of sample creation and evaluation is called a *generation*. Korali will run as many generations required to reach a certain *termination criterion*

Experiments contain *general settings* which affect the way in which Korali runs the experiment and outputs its results.

1.2.2 Korali Problems

A Korali Experiment is configured to run a specific problem type by specifying its *problem type* setting, for example:

```
# Example: Setting the experiment to run an optimization problem
e["Problem"]["Type"] = "Optimization"
```

Sets the experiment to solve an *optimization* problem. The complete list of problem types can be found *here*.

Problem Configuration

Each problem type has its own set of properties necessary to be fully described. These settings can be specified by using the following syntax:

```
# Example: Configuring settings 1 and 2, of string and real types, respectively.
e["Problem"]["Setting 1"] = "String"
e["Problem"]["Setting 2"] = 1.0
```

Depending on the property, their values could be numeric, text strings, functions, or arrays thereof. They can also contain sub-properties which require their own set of properties, as shown below:

```
# Example: Configuring setting 3, which contains sub-properties to be defined.
e["Problem"]["Setting 3"]["Sub-Type"] = "myType"
e["Problem"]["Setting 3"]["Parameter 1"] = 0.0
e["Problem"]["Setting 3"]["Parameter 2"] = 1.0
```

To find the full list of properties for each problem type, look for "Configuration Settings" in the problem's configuration page. Here is, for example, the *configuration settings* for Optimization/Stochastic.

1.2.3 Choosing a Solver Method

The next step is to choose which solver algorithm should be used to obtain the results required by the problem. This can be done by specifying the *solver type* setting.

Solver-Problem Compatibility

Although the complete list of solver types can be found *here*, each solver can only solve a specific set of problem types. To find which solver methods can be used for a specific problem type, look for "Compatible Solvers" in the problem's configuration page. Here is, for example, the *compatible solvers list* for Optimization/Stochastic.

To continue our example above, we will choose to use the *DEA*, which is a compatible solver for the Optimization/Stochastic problem type.

```
e["Solver"]["Type"] = "DEA"
```

It is possible, however, to choose another solver to solve a given problem, simply by changing the solver method choice. For example, if now we wanted to solve the problem using *CMAES* instead, we simply change the field:

```
e["Solver"]["Type"] = "Optimizer/CMAES"
```

Solver Configuration

Korali solvers, just like problems, also contain their own set of settings to configure. For example, *CMAES* requires defining a *population size*, the number of samples to run per iteration.:

```
e["Solver"]["Population Size"] = 32
```

Termination Criteria

A Korali solver will run until at least one of its *termination criteria* is met. Termination criteria are entirely user-defined, and can be modified just like any other parameter, for example:

```
e["Solver"]["Termination Criteria"]["Min Value Difference Threshold"] = 0.0001
e["Solver"]["Termination Criteria"]["Max Generations"] = 1000
```

Will run iterations of the CMAES algorithm until the difference in objective value (optimization) is less than 0.0001, meaning it has reached convergence within an accepted tolerance **OR** until it has reached a total of 1000 generations (iterations). The list of termination criteria for each solver can be found in the "Termination Criteria" section of their documentation. Here is, for example, the *termination criteria list* for CMAES.

Configuration Defaults

Not all the properties or termination criteria of a solver method need to be explicitly defined. Instead, every solver provides a set of defaults values which should work fine in the majority of cases. To see which defaults have been defined for a given method, look for the "Default Configuration" section in their configuration page. Here is, for example, the *default configuration* for CMAES.

1.2.4 Variables

Most problem types require the description of the parameter-space that represents physical or mathematical phenomenon to analyze. To describe the parameter-space a user needs to define one or more *Korali Variable*. Variables are created by simply adding their name into the experiment:

```
# Example: Defining two variables for my problem.
e["Variables"][0]["Name"] = "Thermal Conductivity"
e["Variables"][1]["Name"] = "Heat Source Position"
```

Variable Configuration

Variable definitions require additional parameters depending on which problem and solver types have been selected. These parameters are explained in detail in each solver/problem documentation page.

For example, the following *variable settings* are mandatory for the CMAES solver, and these *variable settings* are mandatory for the optimization problem.

In the code snippet below, we show how the configuration for each variable is specified:

```
# Example: Defining two variables for my problem and their DEA lower/upper␣
↪exploration bounds.
e["Variables"][0]["Name"] = "Thermal Conductivity"
e["Variables"][0]["Lower Bound"] = 0.0
e["Variables"][0]["Upper Bound"] = 1.0
```

(continues on next page)

(continued from previous page)

```
e["Variables"][1]["Name"] = "Heat Source Position"
e["Variables"][1]["Lower Bound"] = -10.0
e["Variables"][1]["Upper Bound"] = +10.0
```

Variable Defaults

Korali problem or solver can specify defaults for their variable settings. To see which variable defaults (if any) have been defined for a given method, look for the "Default Configuration" section in their configuration page. Here is, for example, the *variable defaults* for CMAES.

1.2.5 Korali Samples and Models

Most problems require defining a *model* to be optimized/sampled from (among other purposes). A model in Korali is specified as a python function which contains a mathematical formula or an entire computational simulation of a given phenomenon. Model functions accept a single parameter representing a Korali *sample*. A sample is an input/output object that contains a determinate value for each of the variables defined in the experiment. The syntax to access these values is shown below:

```
# Defining a model function for my experiment
def myModel(k):
thermalConductivity = k["Parameters"][0]
heatSourcePosition  = k["Parameters"][1]
```

The sample (*k*) contains an array of values (*Parameters*) that hold the value of each variable, in the order as they were defined in the experiment.

Model Output

Different problem types require the output of different results from the model. For example, Optimization/Stochastic requires as output the value of the function at the given point (*F(x)*), as shown below:

```
# Defining a model function for my experiment that returns F(x)
def myModel(sample):
thermalConductivity = sample["Parameters"][0]
heatSourcePosition  = sample["Parameters"][1]
distanceFromSource = 1.0 - heatSourcePosition
sample["F(x)"] = thermalConductivity * distanceFromSource * distanceFromSource
```

Users can also save custom quantities of interest for each samples. These quantities are not used by Korali, but they can be later retrieved from the result files to provide additional data for post-processing.

```
# Defining a model function for my experiment that returns F(x) and quantities of␣
↪interest
def myModel(sample):
thermalConductivity = sample["Parameters"][0]
heatSourcePosition  = sample["Parameters"][1]
distanceFromSource = 1.0 - heatSourcePosition
sample["Distance From Source"] = distanceFromSource
sample["F(x)"] = thermalConductivity * distanceFromSource * distanceFromSource
```

Model functions can also be represented as lambda functions:

```
# Defining a lambda model function for my experiment that returns F(x)
myModel = lambda sample : sample["F(x)"] = sample["Parameters"][0] * sample[
↪"Parameters"][1]
```

Using the Model

To assign the model to the experiment, the user passes it as parameter to the corresponding setting. For example, for the Optimization/Stochastic problem, we need to define its *Objective Function*, as follows:

```
# Setting model to optimize
e["Problem"]["Type"] = "Optimization/Stochastic"
e["Problem"]["Objective Function"] = myModel
```

1.2.6 Distributions

Some problem type or solvers require the specification of probability distributions. To create distribution, use the following syntax to specify them by name, type, and properties:

```
# Example: Defining two variables for my problem.
e["Distributions"][0]["Name"] = "My Distribution 1"
e["Distributions"][0]["Type"] = "Univariate/Uniform"
e["Distributions"][0]["Minimum"] = -10.0
e["Distributions"][0]["Maximum"] = +10.0

e["Distributions"][1]["Name"] = "My Distribution 2"
e["Distributions"][1]["Type"] = "Univariate/Normal"
e["Distributions"][1]["Mean"] = 0.0
e["Distributions"][1]["Sigma"] = 5.0
```

A complete list of distribution types and their configuration can be found *here*.

Linking Distribution to Variable

Some problems type (e.g., *Bayesian*) require that variables define a *prior distribution*. This requires linking a variable to a specific distribution, which can be done by name referencing, for example:

```
# Example: Linking a variable with its prior distribution
e["Variables"][0]["Name"] = "Thermal Conductivity"
e["Variables"][0]["Prior Distribution"] = "My Distribution 1"
```

It is possible also to assign the same distribution to different variables:

```
# Example: Using the same distribution for multiple variables
e["Variables"][0]["Name"] = "Thermal Conductivity"
e["Variables"][0]["Prior Distribution"] = "My Distribution 1"

e["Variables"][1]["Name"] = "Heat Source Position"
e["Variables"][1]["Prior Distribution"] = "My Distribution 1"
```

Conditional Properties

Some problem types (e.g., *Hierarchical Bayesian*) require the definition of *conditional priors*, distributions for which properties are given by the value of a variable, for example:

```
# Defining conditional prior distributions for a hierarchical Bayesian problem

e["Variables"][0]["Name"] = "Psi 1"
e["Variables"][1]["Name"] = "Psi 2"

e["Distributions"][0]["Name"] = "Conditional 0"
e["Distributions"][0]["Type"] = "Univariate/Normal"
e["Distributions"][0]["Mean"] = "Psi 1"
e["Distributions"][0]["Standard Deviation"] = "Psi 2"

e["Problem"]["Conditional Priors"] = [ "Conditional 0" ]
```

1.2.7 Running Korali

After the experiment has been fully configured, the user needs to instantiate a *Korali Engine* object:

```
k = korali.Engine()
```

The engine contains all necessary execution logic to run the experiment and produce the results.

Running Experiments

To run a given experiment, simply use the engine's *run()* function, passing the experiment as argument.

```
k.run(e)
```

It is not necessary to instantiate multiple Korali engines if the application needs to run multiple experiment; it suffices to call the *run* function as many times as necessary:

```
k.run(e0)
k.run(e1)
k.run(e2)
```

Similarly, it is possible to launch multiple experiments simultaneously:

```
k.run( [e0, e1, e2] )
```

In this case, Korali will not return until all three experiments have finished.

Running your Korali Application

To run an python application containing a Korali experiment, simply run:

```
python3 ./myKoraliApp arguments
```

1.2.8 Accessing Results

When called, the *run* will not return until one of the experiment's termination criteria has been met. After return, the experiment will contain a *Results* section, from which the user can retrieve the desired results.

Each solver type prescribes a different set of results that it produces. To see which results are produced (if any) by a given method, look for the "Results" section in their configuration page. Here is, for example, the *results*.

To access the results, use the following syntax:

```
bestSample = e["Results"]["Best Sample"]
print('Found best sample at:')
print('Thermal Conductivity = ' + str(bestSample["Parameters"][0])
print('Heat Source Position = ' + str(bestSample["Parameters"][1])
print('Evaluation: ' + bestSample["F(x)"])
```

Result Files

After every generation, Korali stores the entire state of the framework (including results) to a results directory. The default path is given in *experiment defaults*.

To set a different results folder for a given experiment (recommended when you run multiple experiments), use the following syntax:

```
# Setting a different results folder for my experiment
e["File Output"]["Path"] = "./myResultsFolder"
```

If you would like to reduce the frequency of state files output or outright disable it, use the follwing syntax:

```
# Saving results to a file every 5 generations, instead of 1
e0["File Output"]["Enabled"] = True
e0["File Output"]["Frequency"] = 5
e0["File Output"]["Path"] = "/home/user/my.custom.path"

# Disable the output for this other experiment
e1["File Output"]["Enabled"] = False
```

To preserve the all input/output parameters for every sample generated in Korali, you need to enable it by:

```
# Saving the details of all samples generated during execution and their results␣
↪(this file can become really large)
e["Store Sample Information"] = True
```

This option is by default disabled, since storing all samples may require large file sizes.

Console Verbosity

If you'd like to reduce or increase the amount of information that Korali outputs to console when running, you can use the following syntax:

To set a different results folder for a given experiment (recommended when you run multiple experiments), use the following syntax:

```
# Do not print anything to console.
e["Console Output"]["Verbosity"] = "Silent"
```

(continues on next page)

(continued from previous page)

```
# Only print important progress notifications to console
e["Console Output"]["Verbosity"] = "Minimal"

# Print all possible information available.
e["Console Output"]["Verbosity"] = "Detailed"
```

To reduce the output frequency, use the following:

```
# Print partial results only every 5 generations
e["Console Output"]["Frequency"] = 5
```

Plotting Results

To generate a plot with the results of your experiment, check the documentation for the *Korali Plotter* and *Korali Reinforcement Learning View* tools.

1.3 Parallel / Distributed Execution

The Korali engine is tailor-made for execution in parallel and distributed systems, especially large-scale supercomputers.

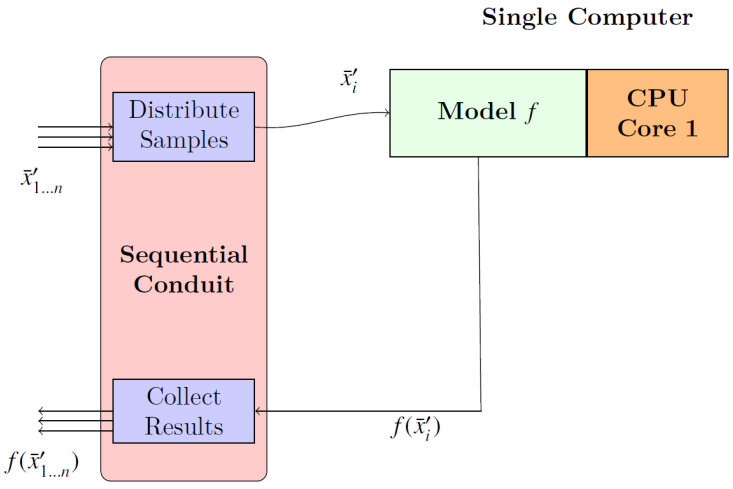
Since the possible array of particular needs and systems can be diverse, Korali exposes multiple *Execution Conduits*. An execution conduit handles the distribution of sample evaluations from/to the parallel system. Each conduit provides its own set of advantages and has its own set of configuration settings.

The conduit is a property of the Korali engine, which means that all experiments that a given engine runs will use the same parallelism strategy.

In this document, we discuss different parallelism/distributed execution scenarios and show which conduit fits better for the case.

1.3.1 Sequential Execution (No parallelism)

The default behavior for Korali is to use the *Sequential Conduit* conduit, which runs a single sample evaluation at any given moment using a single CPU core.



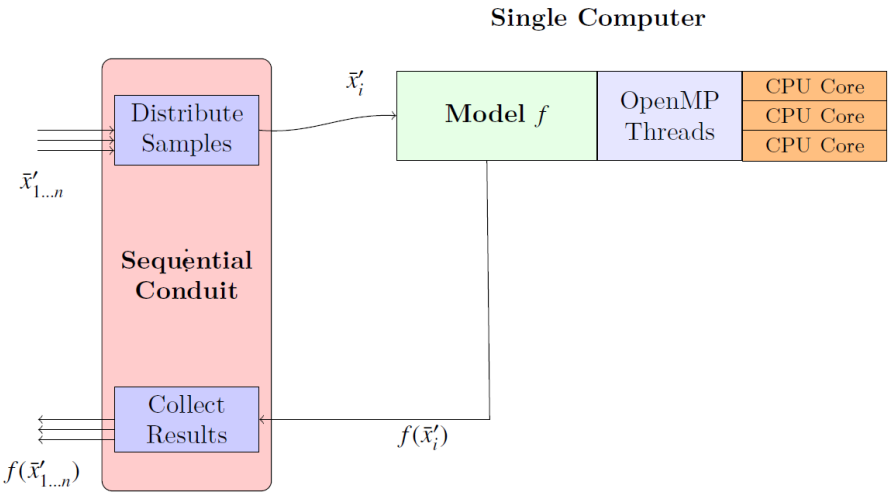
It is not necessary to specify any additional configuration for the sequential conduit to run.

This scenario assumes that the model function is also sequential.

1.3.2 Local Parallelism

Sequential Sampling - Parallel Model

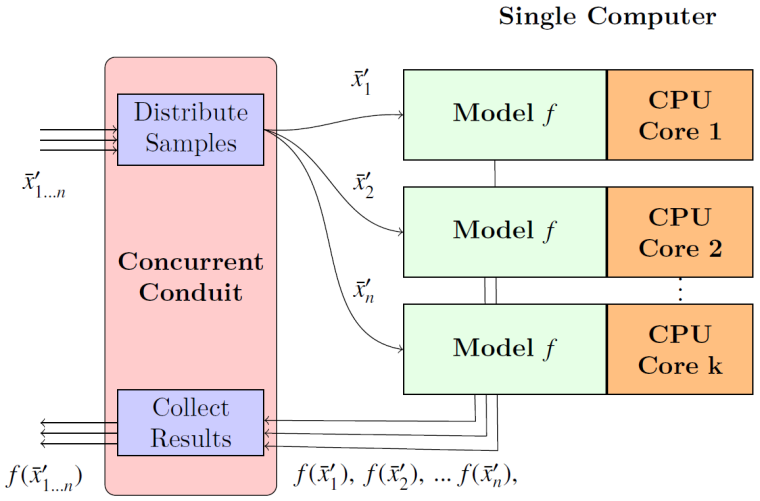
This scenario is similar to the one above, except that the model function uses parallelism to compute.



In this case, the model function may be using threads (e.g., OpenMP, Pthreads), or multi-processing (Fork/Join). This setup is perfectly compatible with Korali.

Parallel Sampling - Sequential Model

In this scenario, we use local parallelism (single computer) to run many simultaneous instances of a sequential model.



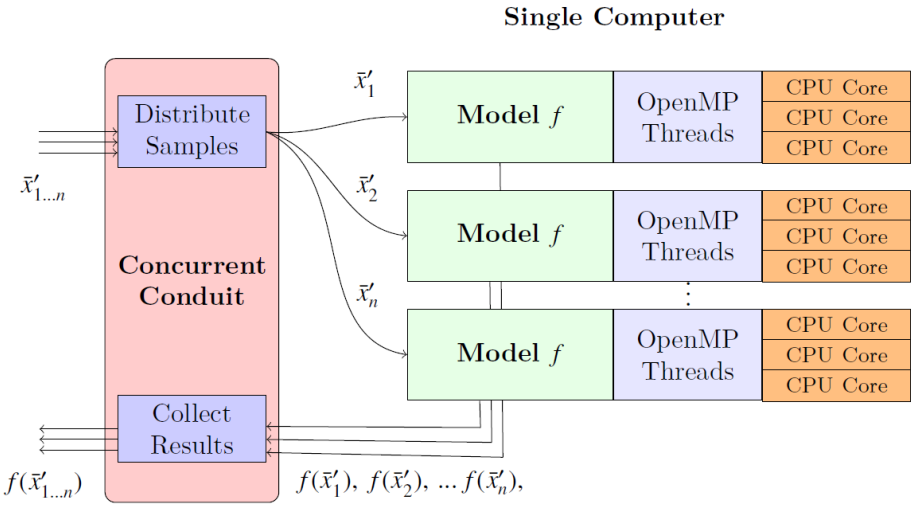
To enable this, we use the *Concurrent Conduit*, which uses multi-processing (Fork-Join model) to create many instances of the Korali process: one for the main engine, and the others for workers whose only task is to evaluate samples and return their results.

```
k["Conduit"]["Type"] = "Concurrent"
k["Conduit"]["Concurrent Jobs"] = 16
```

In this case, Korali will create 16 worker processes (see: concurrent jobs setting), using 16 CPU nodes to run the model.

Parallel Sampling - Parallel Model

In this scenario, we use local parallelism (single computer) to run many simultaneous instances of a parallel model.



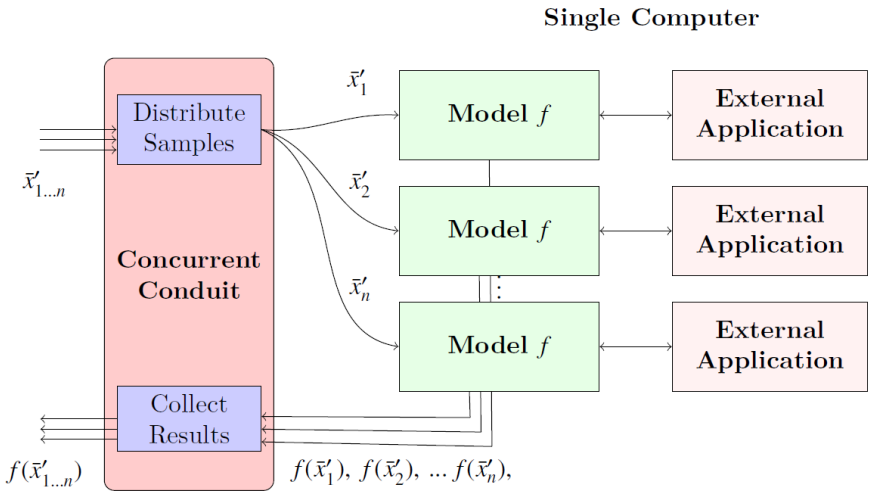
This scenario is similar to the one above, except that the model function uses parallelism to compute. In this case, the user needs to be careful not to oversubscribe the CPU with too many threads/processes.

In this case, we recommend to maximize sample-based parallelism, specifying as many concurrent jobs as possible, as opposed to the model's threads per execution.

Examples of the use of this conduit can be found in *Concurrent Execution*.

Parallel Sampling - Pre-Compiled Model

The *Concurrent Conduit* allows for the parallel execution of pre-compiled/binary files/legacy codes.



1.3.3 Distributed Parallelism

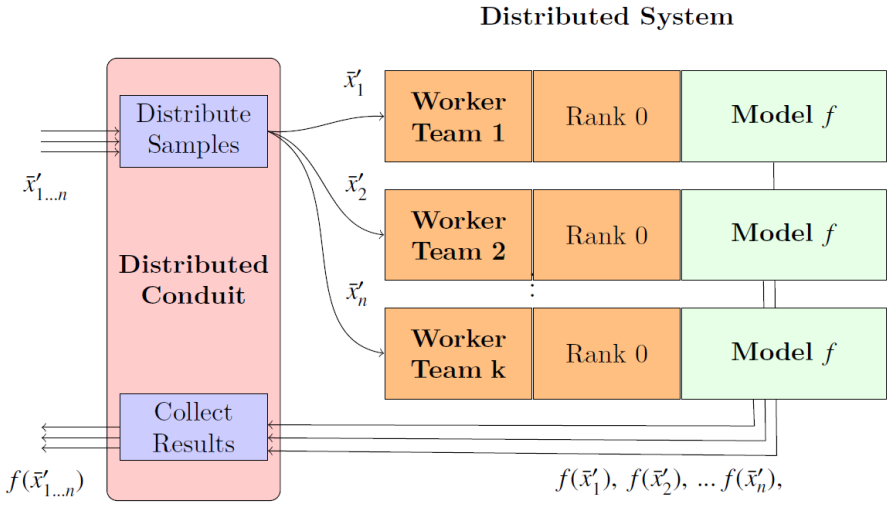
Here we discuss scenarios were parallelism extends to multiple computers using distributed computing models through MPI.

For an example on how to create MPI/Python Korali applications, see: *MPI/Python Example*).

For an example on how to create MPI/C++ Korali applications, see: *MPI/C++ Example*).

Distributed Sampling - Sequential Model

In this scenario, we use distributed parallelism (many computers) to run many simultaneous instances of a sequential model.



To enable this, we use the *Distributed Conduit*, which uses MPI as communication backend to create many instances of Korali workers distributed among the system.

The following code snippet shows how to set the distributed conduit to run a sequential model, also specifying the MPI communicator to use:

```
k.setMPIComm(MPI_COMM_WORLD)
k["Conduit"]["Type"] = "Distributed";
k["Conduit"]["Ranks Per Worker"] = 1;
```

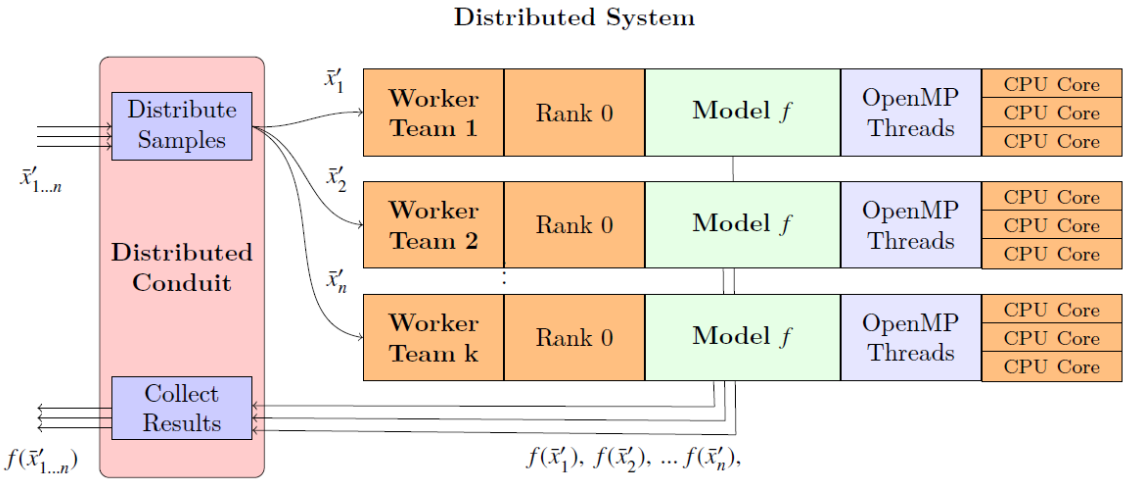
And run it using `mpirun` or similar launch command, for example:

```
mpirun -n 257 ./myKoraliExperiment.py
```

This example will run 256 Korali workers (257 - 1 for the main Korali engine), each one running the model function using a single process to compute.

Distributed Sampling - Parallel Model

This scenario is similar to the one above, except that the model function uses thread-parallelism (e.g., OpenMP) or GPUs (e.g, via CUDA) to compute.



In this case, it is recommended that the user runs one Korali worker per node/NUMA domain, and then the model function uses threading to employ all the cores/GPU therein.

```
k::setMPIComm(MPI_COMM_WORLD);
k["Conduit"]["Type"] = "Distributed";
k["Conduit"]["Ranks Per Worker"] = 1;
```

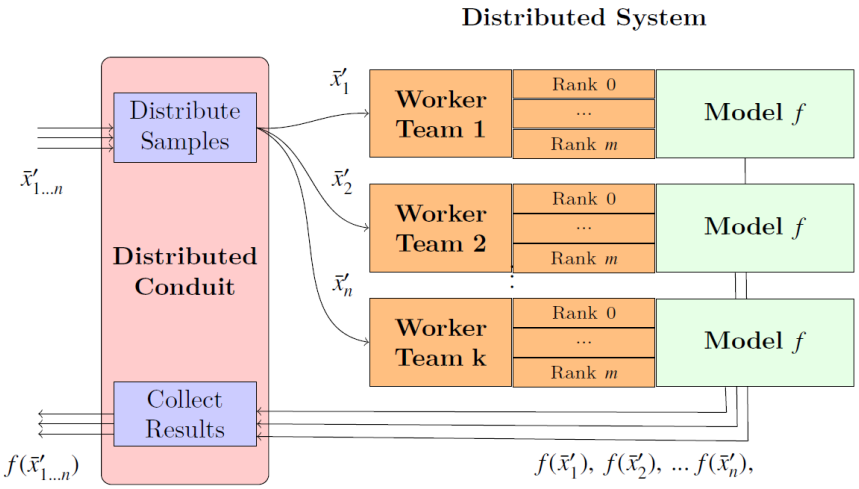
And run it using `mpirun` or similar launch command, for example:

```
mpirun -n 17 --ranks-per-node=1 ./myKoraliExperiment.py
```

Where the run will employ 17 nodes, one for the engine, and 16 for the workers.

Distributed Sampling - Distributed (MPI) Model

This scenario is similar to the one above, except that the model function uses MPI as distributed parallelism library.



This is the general case for the *Distributed Conduit*, in which worker can contain more than one rank. For example,

```
k.setMPIComm(MPI_COMM_WORLD);
k["Conduit"]["Type"] = "Distributed";
k["Conduit"]["Ranks Per Worker"] = 4;
```

The model function should expect an MPI Communicator object and operate upon it as in the following example:

```
void myMPIModel(korali::Sample &sample)
{
 MPI_Comm comm = *(MPI_Comm*) korali::getWorkerMPIComm();

 double x = sample["Variables"]["X"];
 double q = compute_partial_result(x);

 double result;
 MPI_Allreduce(&q, &result, ..., MPIComm);
 sample["F(x)"] = result;
}
```

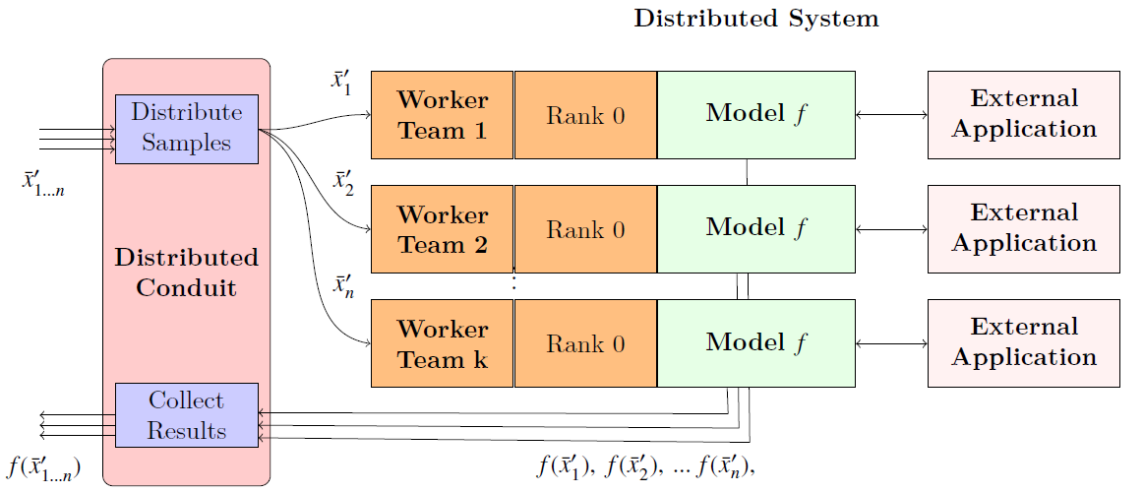
And run it using `mpirun` or similar launch command, for example:

```
mpirun -n 257 ./myKoraliExperiment
```

Where the run will employ 257 cores, one for the engine. With the reamining 256 ranks, it will create 64 workers of 4 ranks each.

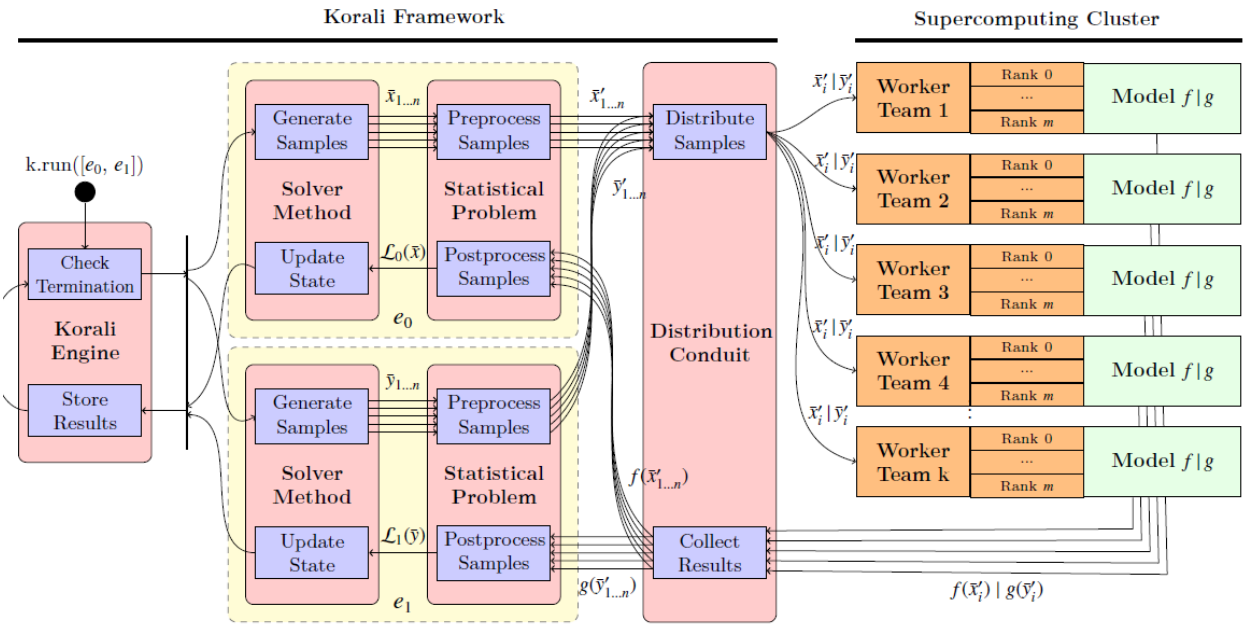
Non-Intrusive Distributed Sampling

This is the case in which we run an external application in a distributed system.



1.3.4 Distributed Multi-Experiment Runs

Korali conduits are capable of running multiple, independent experiments simultaneously. Furthermore, these experiments neet not be similar in their configuration, as they can specify diverse problem types and solver methods. The purpose for enabling multi-experiment runs is to increase the pool of pending samples, maximizing the occupation of Korali workers.



1.3.5 Obtaining Profiling Information

The Korali engine can be configured to store profiling information that allows the post-mortem reconstruction of the execution timelines for each worker. This allows users to measure the efficiency of their parallelism strategy.

To enable profiling information, use this syntax:

```
k["Enable Profiling"] = True
```

Visit Korali's *profiler tool* documentation page for details on how to visualize profiling information.

1.4 Checkpoint / Resume

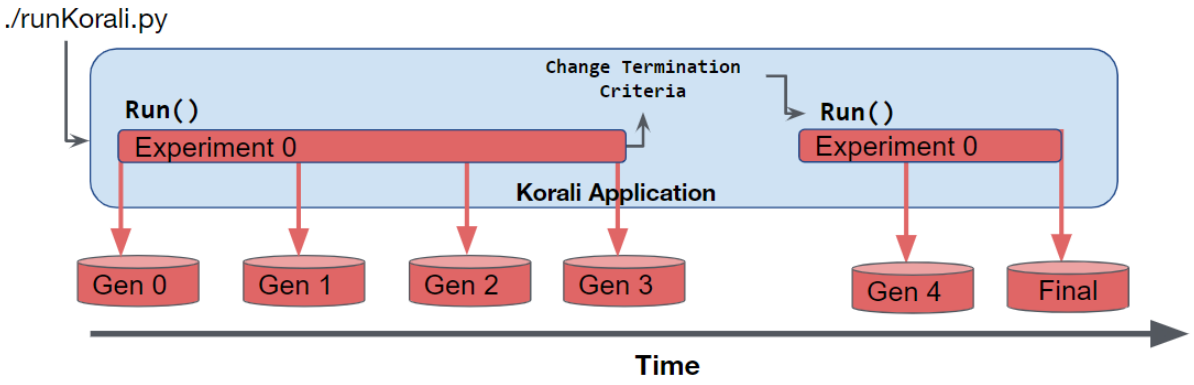
Korali experiments can either run to completion or in part. In the latter case, the user can then operate over its partial results or change termination criteria and continue its execution later.

To ensure fault-tolerance for all experiments and systems, by default Korali also stores the entire state into its result files (checkpoints), regardless of the solver/problem/conduit modules selected. Execution can be later resumed from any of its result files without loss of information.

Below, we discuss three possible use case scenarios for Korali's checkpoint/resume capability.

1.4.1 Running an experiment partially

In this scenario, we have a Korali application which runs a stochastic optimization problem (using CMAES) for a few generations, checks partial results, and then continues with the rest of the generations.

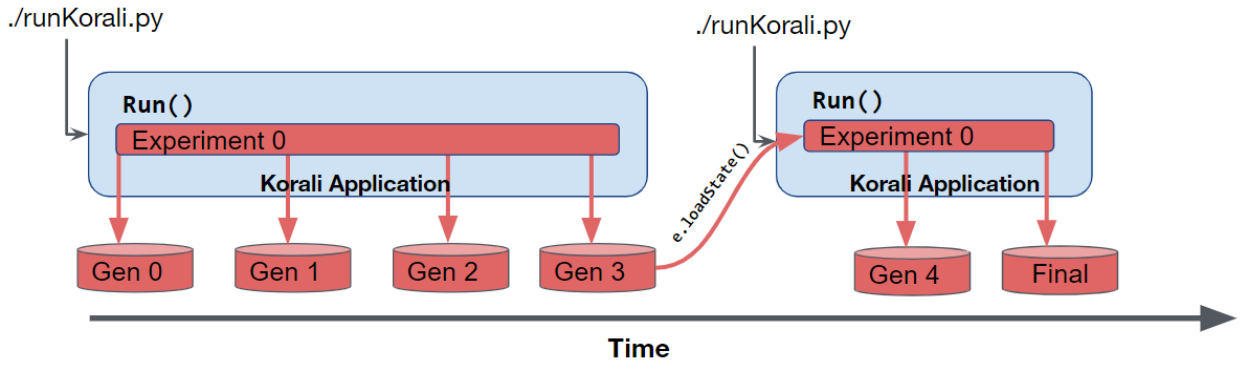


To do this, we simply define an initial set of termination criteria, and run the experiment:

```
import korali

# Starting Engine
k = korali.Engine()

# Configuring experiment
e = korali.Experiment()
e["Problem"]["Type"] = "Optimization/Stochastic"
e["Problem"]["Objective Function"] = myModel
e["Solver"]["Type"] = "Optimizer/CMAES"

# ...  More experiment configuration ...

# Limiting maximum number of generations to 3
e["Solver"]["Termination Criteria"]["Max Generation"] = 3

# Running 3 generations
k.run(e)

# Evaluating a domain-specific criterion as to whether to continue or not
if (e["Results"]["Best Sample"]["F(x)"] < 1.0):

 # Extending max generation number
 e["Solver"]["Termination Criteria"]["Max Generation"] = 100

 # Running again, starting from generation 4.
 k.run(e)
```

1.4.2 Resuming an experiment from a save-state file

In this scenario, we continue the execution of an unfinished experiment which had started prior to the execution of the current application and saved checkpoints of its progress.



This use case is common in large-scale executions when a model evaluation error may occur, or simply a an allocation job has run out of allocated time.

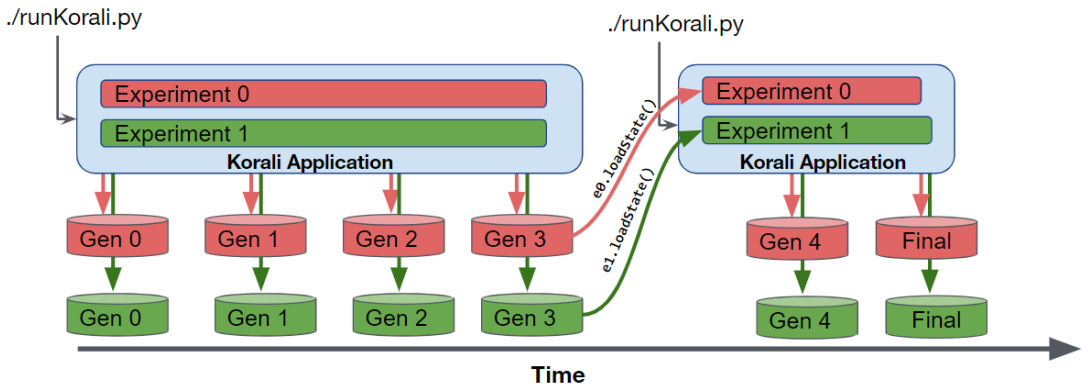
To load the state an unfinished experiment, we use the `e.loadState()` function.

To do this, we simply define an initial set of termination criteria, and run the experiment:

```
import korali

# Starting Engine
k = korali.Engine()

# Creating experiment
e = korali.Experiment()

# Setting the path from which to load checkpoints and to write new ones
e["File Output"]["Path"] = "resultsPath"

# Loading previous results, if they exist.
found = e.loadState()

# Important: Re-specify model functions, because they are not stored in checkpoints
e["Problem"]["Objective Function"] = myModel

# We continue execution
if (found == True):
  k.run(e)
```

1.4.3 Resuming multiple experiments simultaneously

In this scenario, we continue the execution of multiple unfinished experiments which had started prior to the execution of the current application and saved checkpoints of their progress.



This use case is common in large-scale executions when a model evaluation error may occur, or simply a an allocation job has run out of allocated time.

In this case, we use the `e.loadState()` function for all experiments.

```
import korali

# Starting Engine
k = korali.Engine()

# Creating experiments
e0 = korali.Experiment()
e1 = korali.Experiment()

# Setting the path from which to load checkpoints and to write new ones
e0["File Output"]["Path"] = "resultsPath0"
e1["File Output"]["Path"] = "resultsPath1"

# Loading previous results, if they exist.
e0.loadState()
e1.loadState()

# Important: Re-specify model functions, because they are not stored in checkpoints
e0["Problem"]["Objective Function"] = myModel0
e1["Problem"]["Objective Function"] = myModel1

# We continue execution
k.run([e0, e1])
```

1.5 Tools

1.5.1 Korali Plotter

Usage

Syntax: `python3 -m korali.plot [--dir=RESULTS_DIR] [--test]`

Where:

- `--dir` specifies the source path of Korali results to plot. By default: `_korali_result/`
- `--test` verifies that the plotter works, without plotting to screen.

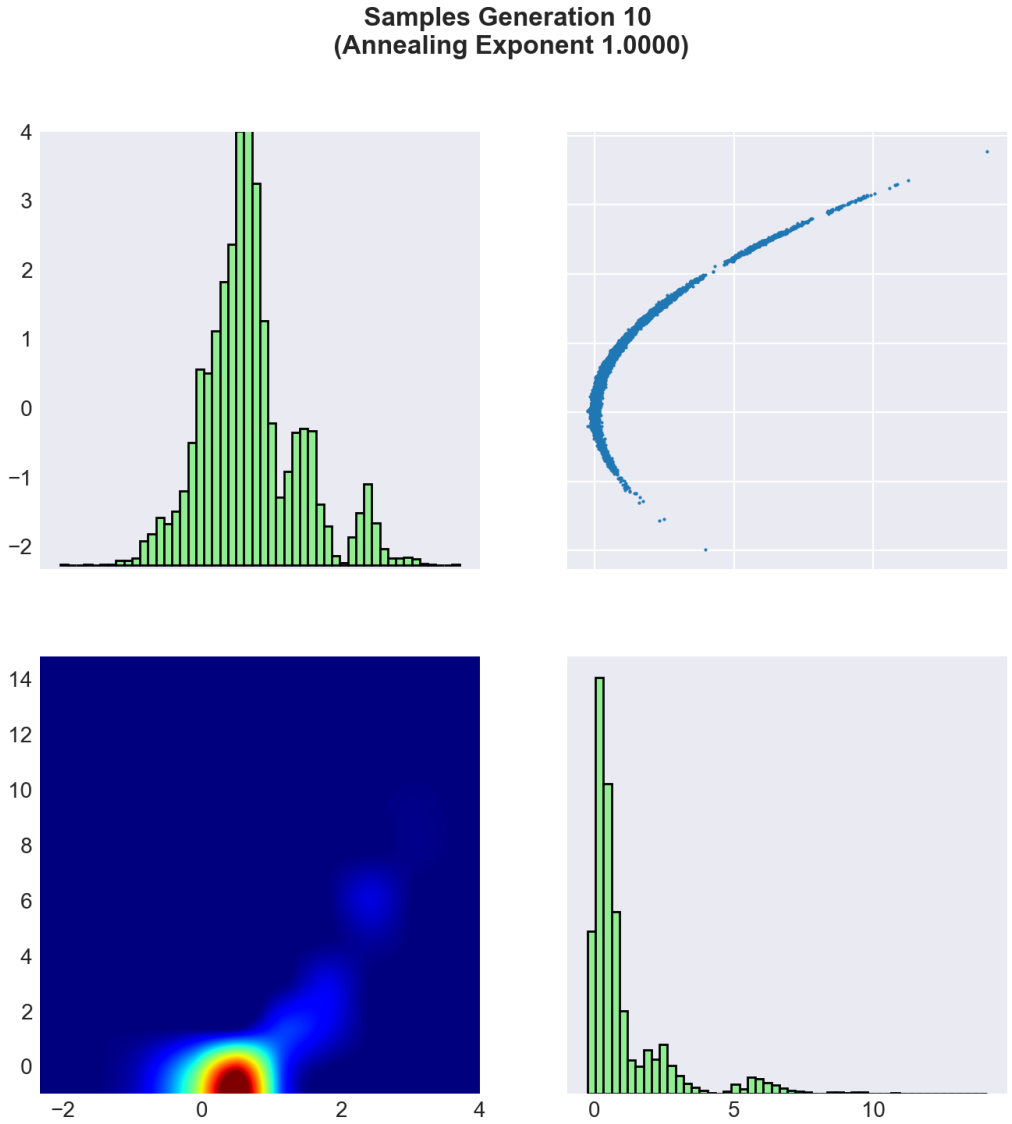
Examples

Plotting TMCMC

Here we explain technical details of the *TMCMC* result plot.

The `python3 -m korali.plot` command plots the distribution of the samples at every generation. The samples are read from the json-files stored in the output directory (`/_korali_result/`).

A plot of the samples obtained after the final generation of TMCMC function is given below. Here, the target function is the exponential of the negative of the 2-dimensional Rosenbrock function.



Diagonal Plots. The diagonal plots show the marginal probability densities of the samples for each variable. Note that the indices of the vertical axes correspond to the upper and lower triangle plot and not to the diagonal plots. **Upper Triangle**. In the plots in the upper triangle we see the actual realization of the samples. The axes represent

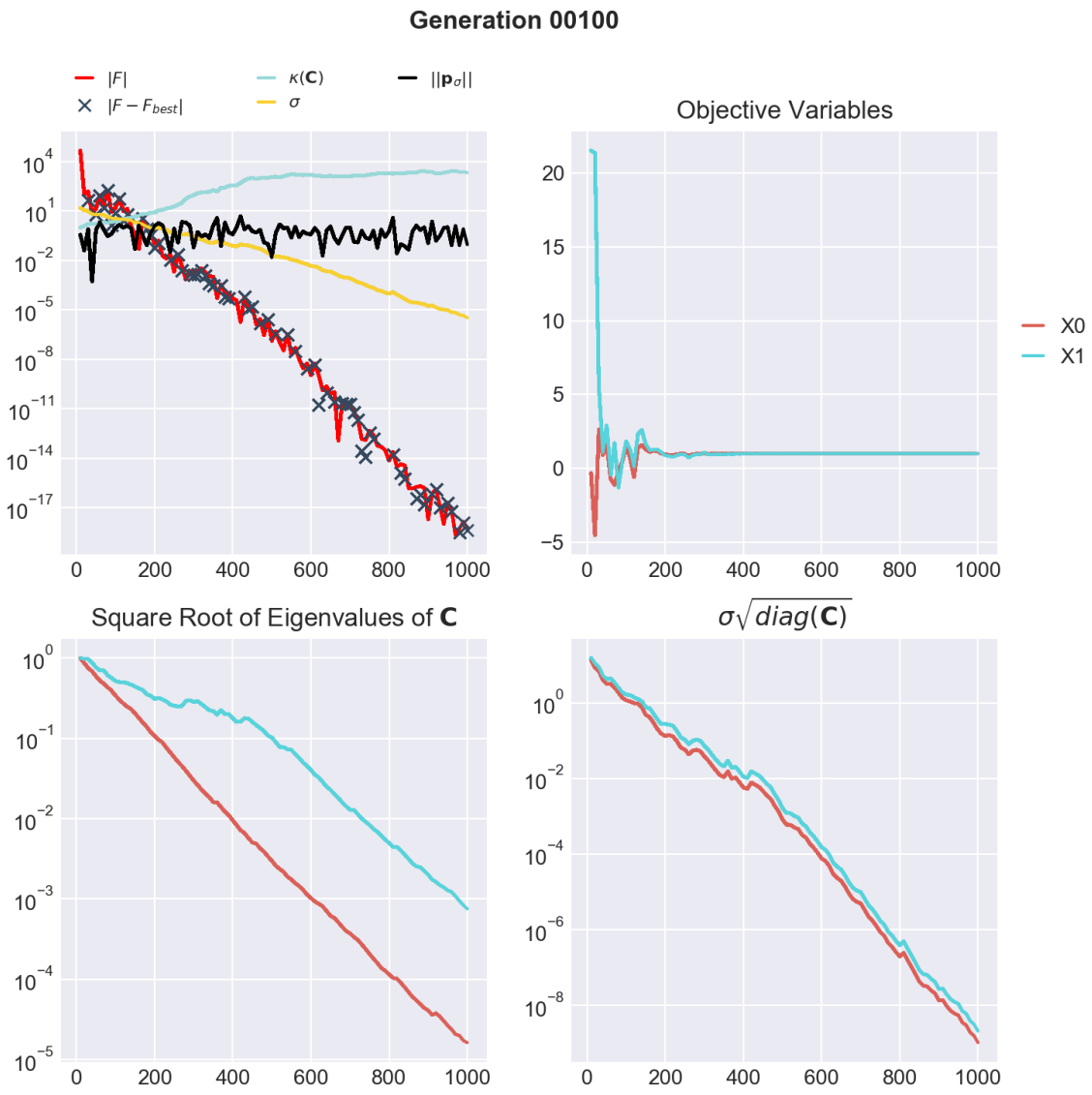
the dimensions, respectively the objective variables, of the problem and we show a two-dimensional plot for every variable pair. **Lower Triangle**. The lower triangle shows the probability density of the samples for each variable pair. The density is approximated through a smoothening operator applied to the number of samples that can be found in a given area.

Plotting CMAES

Here we explain *CMAES* result plot in further detail and how it can be used to validate your optimization.

The module korali.plot (run with `python3 -m korali.plot`) command visualizes some of the most meaningful states of CMA-ES stored in the result files in the output directory (`_korali_result`). To plot a running simulation use the command `python3 -m korali.plot --live` for incremental plots.

In the figure below we see the evolution of the CMA-ES algorithm during 100 optimization steps, respectively 1000 function evaluations (here the sample size is 10), of the negative 2-dimensional Rosenbrock function.



Quadrant 1: The first quadrant (upper left) shows 4 graphs plus markers (crosses):

- $|F|$ is the best function evaluation of the current generation. Note that the colour of F changes if $F < 0$ (red) or $F \geq 0$ (blue). Also, the absolute value of F is plotted since the vertical axis is given in log-scale.
- $\kappa(\mathbf{C})$ (cyan): This line shows the condition of the covariance matrix of the proposal distribution at every generation. The condition is given by the ratio of the largest Eigenvalue to the smallest Eigenvalue. A large condition number may lead to numerical instabilities, this can be treated by normalizing the domain of the objective variables.
- $||\mathbf{p}_\sigma||$ (black): The evolution path is a measure of the travel direction of the mean of the proposal distribution of CMA-ES. The Euclidean norm of the evolution path plays an important rule in the Sigma updating rule.
- σ is the scaling parameter of the covariance matrix. The scaling parameter is updated at every generation. If Sigma becomes very large or small it may have an adverse effect on the optimization.
- $|F - F_{best}|$ (crosses) : At every generation we calculate the absolute difference betwen the current best function

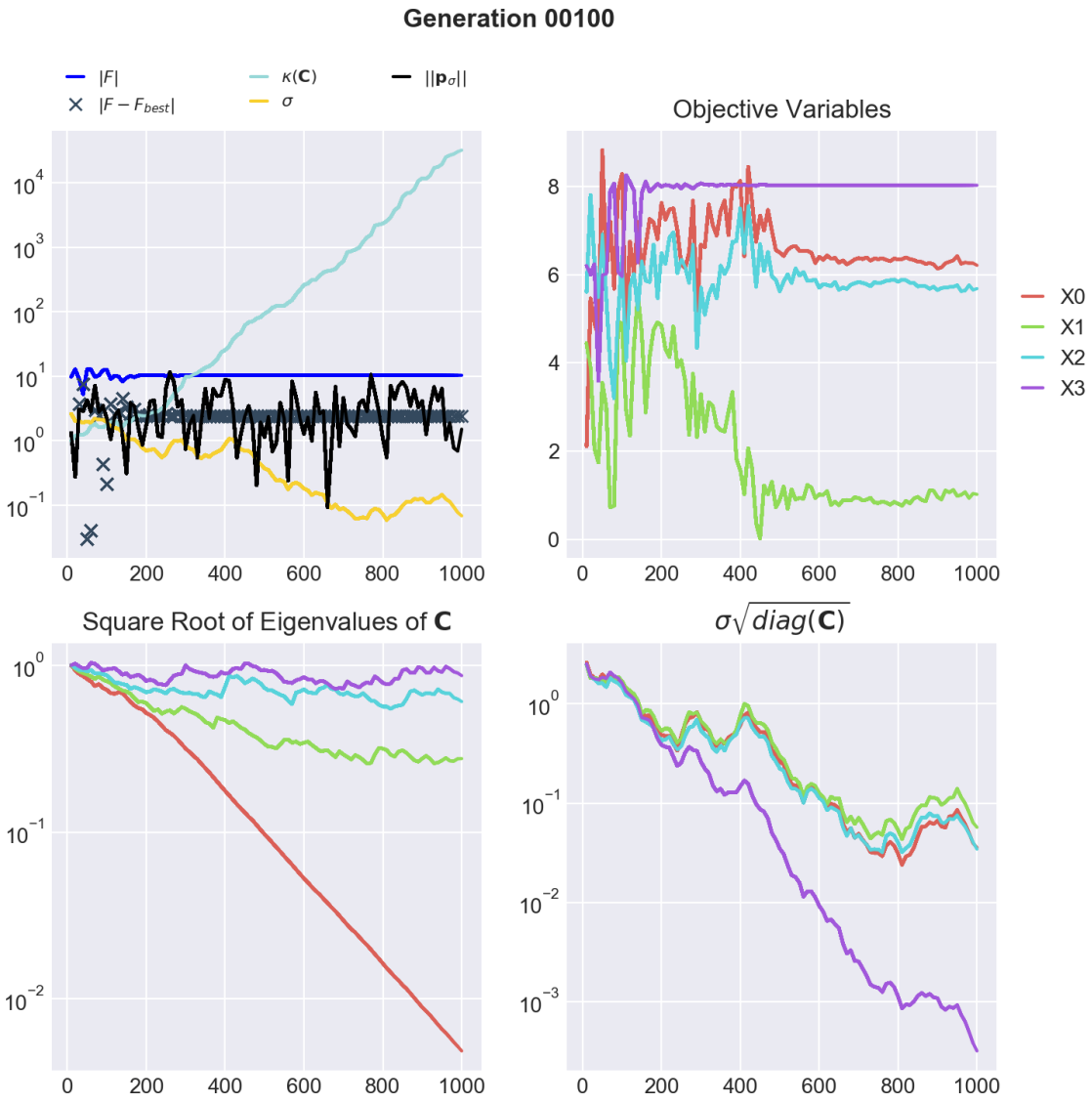
evaluation (F) and the overall best found evaluation (F_{best}) of CMA-ES. The crosses appear only if the current generation does not impeove the overall result, i.e. $F < F_{best}$ in current generation and $|F - F_{best}|$ is greater 0.

A good indicator of convergance of CMA-ES to the global maximum is given by a steady decrease of $|F - F_{best}|$.

Quadrant 2: Objective Variables: This plot shows a solid line for each objective variable corresponding to the evaluation of $|F|$ and a dashed line for each variable corresponding to F_{best}. **Quadrant 3**: Square Root of Eigenvalues $\mathbf{C}$: The square root of the Eigenvalues of $\mathbf{C}$ are the lengths of the axes of the (unscaled) covariance matrix. Optimally the lengths of the axes are of same magnitude. **Quadrant 4**: $\sigma\sqrt{diag(\mathbf{C})}$: the square root of the diagonal elements of the (scaled) covariance matrix of the proposal distribution approximate the standard deviation of the parameters. Ideally the standard deviations of all coordinates are of same magnitude.

Example: Shekel function

The following figure shows the results of an unsuccessful maximization of the negative of the Shekel function in 4 dimensions and with 10 local maxima.



We know that the Shekel function has a global minimum at (4, 4, 4, 4), respectivel maximum in the negative case. In quadrant 2 we see that CMA-ES converged to a different result.

In general the global optimum is not known, following observations indicate ill convergence. Restarting CMA-ES from different starting points as well as tuning CMA-ES internal parameters might improve optimization:

- Increasing condition (quadrant 1) of the covariance matrix of the proposal distribution, respectively diverging axes lenghts and standard deviations (quadrants 3 & 4).
- None decreasing values for $|F - F_{best}|$. Arguably CMA-ES found a better function evaluation on a different hill but the algorithm is trapped (the objective variables stabilized sampling does not overcome the saddle points).

1.5.2 Korali Profiler

Usage

Syntax: `python3 -m korali.profiler [--input PROFILING_FILE] [--output OUTPUT_FILE] [--tend END_TIME] [--test]`

Where:

- `--input` specifies the profiler file to load. By default: `profiling.json`.
- `--output` indicates the output file path and type (e.g., eps, png). If not specified, it prints to screen.
- `--tend` indicates time lapse to print. If not specified, it will print the entire execution.
- `--test` verifies that the plotter works, without plotting to screen.

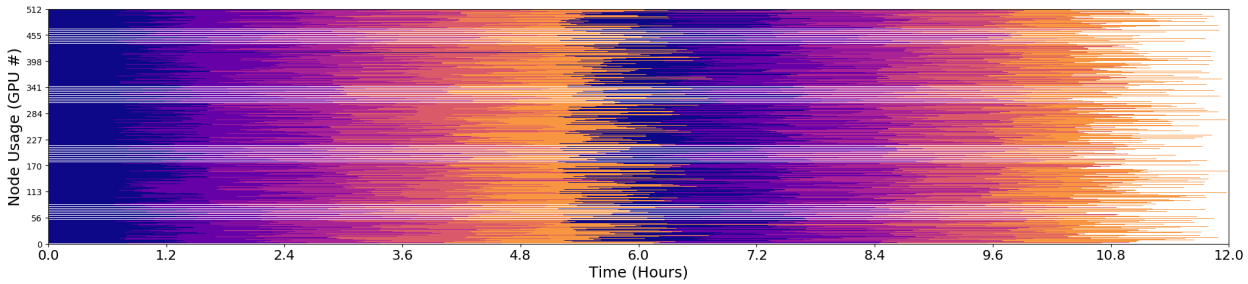
Examples

Profiling Multiple Experiments

In this example, we plot the execution timeline of a 512 workers (each executing Mirheo on a GPU) solving five different experiments simultaneously.

- Profiling File: example_multiple_512Nodes.json

The image below shows in the y-axis the worker id, and in the x-axis the elapsed time. Colored lines show when a worker is active executing a model, with different colors corresponding to a different experiment. White segments indicate that the worker is idle, waiting for new samples to arrive.

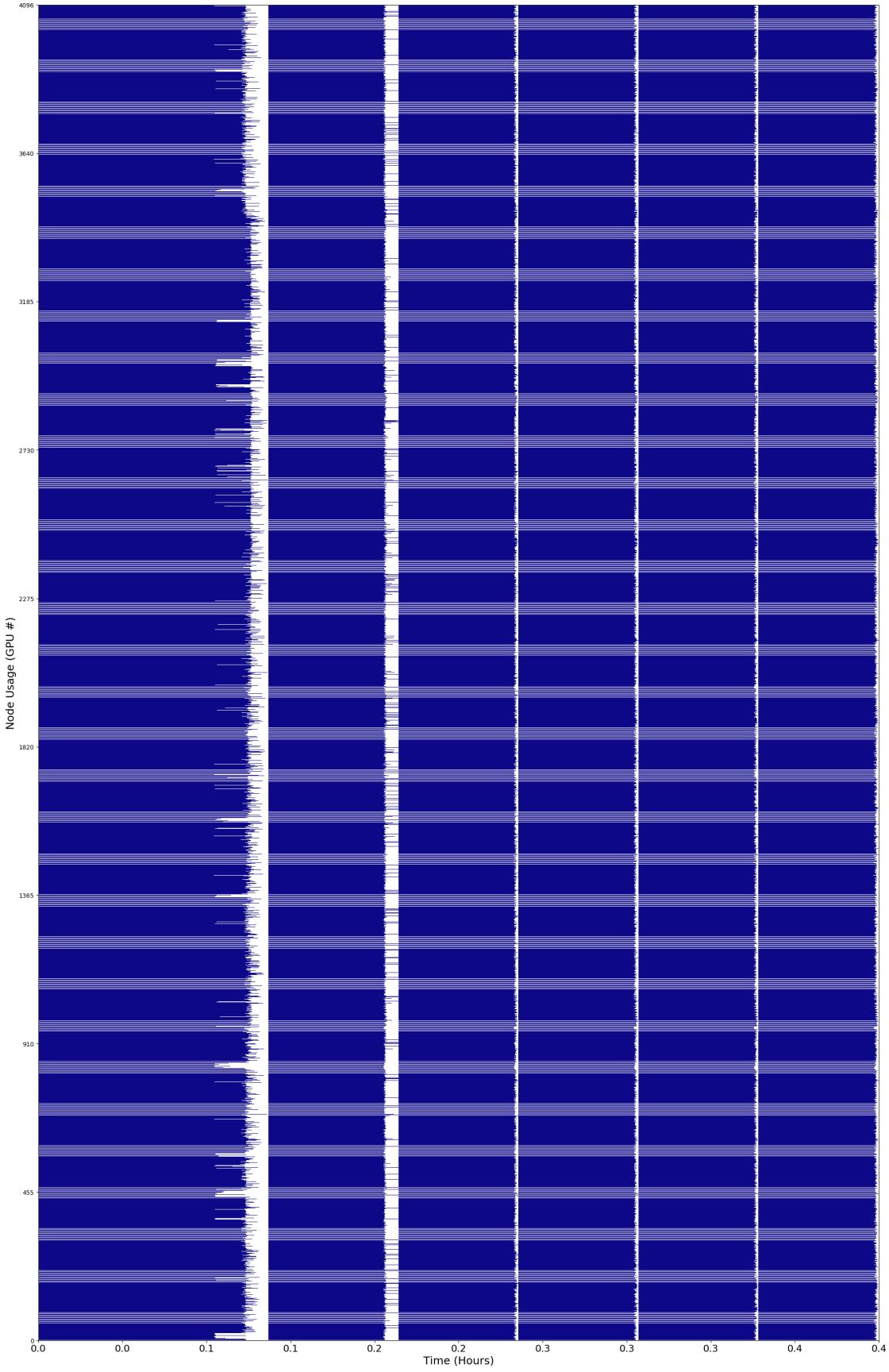


Profiling a Single Large-Experiment

In this example, we plot the execution timeline of a 4096 workers (each executing Mirheo on a GPU) solving a single experiment.

- Profiling File: example_single_4096Nodes.json

In this case, since workers can only draw from a single experiment, they tend to spend more time idle. This is caused by load imbalance, when some samples require more time to finish, forcing others to wait until the next generation.


Node Usage (GPU #)
4096
3640
3185
2730
2275
1820
1365
910
455
0
0.0
0.0
0.1
0.1
0.2
0.2
0.3
0.3
0.4
Time (Hours)

1.5.3 Korali RL Viewer

Usage

Plots the result of RL experiments and allows for comparing multiple results simultaneously.

Syntax: `python3 -m korali.rlview [--dir (RESULTS_DIR1 RESULTS_DIR2 ...)] [--minReward 1.00] [--maxReward 1.00] [--updateFrequency 1.00] [--test] [--check]`

Where:

- `--dir` specifies the source path(s) of Korali results to plot, separated by space. By default: `_korali_result/`
- `--minReward` specifies the lower bound on the y-axis (reward)
- `--minReward` specifies the upper bound on the y-axis (reward)
- `--updateFrequency` specifies the how often should the plotter show live updates
- `--test` verifies that the tool works, without plotting to screen.
- `--check` verifies that the tool is correctly installed, for testing purposes.

1.6 Bayesian Inference

In this section, we show different ways to solve Bayesian Inference problems.

Sub-Categories

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/bayesian.inference/reference/

1.6.1 Inference with Reference Data

In this tutorial we show how to **optimize** and **sample** the posterior distribution of a Bayesian inference problem.

Problem Description

In this example we will solve the inverse problem of estimating the Variables of a linear model using noisy data. We consider the computational model,

$$f(x;\vartheta) = \vartheta_0 + \vartheta_1 x\,,$$

for $x \in \mathbb{R}$. We assume the following error model,

$$y = f(x;\vartheta) + \varepsilon\,,$$

with ε a random variable that follows normal distribution with zero mean and σ standard deviation. This assumption leads to the likelihood function,

$$p(y|\varphi, x) = \mathcal{N}(\, y \,|\, f(x;\vartheta), \sigma^2\,)\,.$$

where $\varphi = (\vartheta, \sigma)$ is the parameter vector that contains the computational variables and the variables of the statistical model.

The data on which we condition our posterior distribution is defined in the model (see source code).

We call this data set $d = \{x_i, y_i\}_{i=1}^{5}$. Assuming that each datum is independent, the likelihood of d under the linear model is given by

$$p(y|\vartheta, x) = \prod_{i=1}^{6} \mathcal{N}(\, y_i \,|\, f(x_i, \vartheta), \sigma^2 \,) \,.$$

In order to identify the distribution of φ conditioned on the observations d we use Bayes' theorem

$$p(\varphi|y, x) = \frac{p(y|\varphi, x)\, p(\varphi)}{p(y)} \,.$$

As a prior information we choose the uniform distribution in $[-5, 5]$ for ϑ and the uniform distribution in $[0, 10]$ for σ.

The Objective Function

Create a folder named *model*. Inside, create a file with name *posteriorModel.py* and paste the following code,

```
#!/usr/bin/env python

def model( s, x ):

    v1 = s["Parameters"][0]
    v2 = s["Parameters"][1]

    result = [ ]
    for i in range(len(x)):
      result.append(v1*x[i] + v2)

    s["Reference Evaluations"] = result
```

This function corresponds implements the computational model that corresponds to $f(x\vartheta) = \vartheta_0 + \vartheta_1 x$. Note: The following might be outdated: The object *s* must be of type *Korali::modelData* This class provides the methods *getParameter* and *addResult*. For a detailed presentation see [here]

In the same file add the following functions that return the data presented in the table above,

```
def getReferenceData():
  y=[]
  y.append(3.2069);
  y.append(4.1454);
  y.append(4.9393);
  y.append(6.0588);
  y.append(6.8425);
  return y

def getReferencePoints():
  x=[]
  x.append(1.0);
  x.append(2.0);
  x.append(3.0);
  x.append(4.0);
  x.append(5.0);
  return x
```

Optimization with CMA-ES

First, open a file and import the korali module

```
#!/usr/bin/env python3
import korali
```

Import the computational model,

```
import sys
sys.path.append('./model')
from posteriorModel import *
```

The Korali Experiment Object

Next we construct a *Korali.Experiment* object and set the computational model, where we already pass the data,

```
e = korali.Experiment()
e["Problem"]["Computational Model"] = lambda sampleData: model(sampleData,␣
↪getReferencePoints())
```

The reference points *x* returned by *getReferencePoints()* correspond to the *input* variables of the model. The function that is passed to Korali should not have an argument for *x*. We have to create an intermediate lambda function that will hide *x* from korali.

```
lambda sampleData: model(sampleData, getReferencePoints())
```

The Problem Type

The *Type* of the *Problem* is characterized as *Bayesian*

```
e["Problem"]["Type"] = "Evaluation/Bayesian/Inference/Reference"
```

When the Type is *Bayesian* we must set the type of likelihood and provide a vector with the *Reference Data* to Korali,

```
e["Problem"]["Likelihood Model"] = "Additive Normal"
e["Problem"]["Reference Data"] = getReferenceData()
```

The Variables

We define two *Variables* of type *Computational* that correspond to ϑ_0 and ϑ_1. The prior distribution of both is set to *Uniform*.

```
e["Variables"][0]["Name"] = "a"
e["Variables"][0]["Bayesian Type"] = "Computational"
e["Variables"][0]["Prior Distribution"] = "Uniform 0"
e["Variables"][0]["Initial Mean"] = +0.0
e["Variables"][0]["Initial Standard Deviation"] = +1.0

e["Variables"][1]["Name"] = "b"
e["Variables"][1]["Bayesian Type"] = "Computational"
e["Variables"][1]["Prior Distribution"] = "Uniform 1"
e["Variables"][1]["Initial Mean"] = +0.0
e["Variables"][1]["Initial Standard Deviation"] = +1.0
```

The last parameter we add is of *Type Statistical* and corresponds to the variable σ in the likelihood function,

```
e["Variables"][2]["Name"] = "Sigma"
e["Variables"][2]["Bayesian Type"] = "Statistical"
e["Variables"][2]["Prior Distribution"] = "Uniform 2"
e["Variables"][2]["Initial Mean"] = +2.5
e["Variables"][2]["Initial Standard Deviation"] = +0.5
```

The Solver

Next, we choose the solver *CMA-ES*, the population size to be *24*.

```
e["Solver"]["Type"] = "Optimizer/CMAES"
e["Solver"]["Population Size"] = 24
```

And activating one of its available termination criteria.

```
e["Solver"]["Termination Criteria"]["Max Generations"] = 100
```

We also need to configure the problem's random distributions, which we referred to when defining our variables,

```
e["Distributions"][0]["Name"] = "Uniform 0"
e["Distributions"][0]["Type"] = "Univariate/Uniform"
e["Distributions"][0]["Minimum"] = -5.0
e["Distributions"][0]["Maximum"] = +5.0

e["Distributions"][1]["Name"] = "Uniform 1"
e["Distributions"][1]["Type"] = "Univariate/Uniform"
e["Distributions"][1]["Minimum"] = -5.0
e["Distributions"][1]["Maximum"] = +5.0

e["Distributions"][2]["Name"] = "Uniform 2"
e["Distributions"][2]["Type"] = "Univariate/Uniform"
e["Distributions"][2]["Minimum"] = 0.0
e["Distributions"][2]["Maximum"] = +5.0
```

For a detailed description of CMA-ES settings see *CMAES*

Finally, we configure the output, and then need to add a call to the run() routine to start the Korali engine.

```
e["File Output"]["Frequency"] = 5
e["Console Output"]["Frequency"] = 5

k = korali.Engine()
k.run(e)
```

Running

We are now ready to run our example: *./run-cmaes.py* The results are saved in the folder *_korali_result/*.

Plotting

You can see the results of CMA-ES by running the command, *python3 -m korali.plot –dir _korali_result_cmaes*

Sampling with TMCMC

To sample the posterior distribution, we set the solver to *TMCMC* sampler and set a few settings,

```
e["Solver"]["Type"] = "Sampler/TMCMC"
e["Solver"]["Population Size"] = 5000
```

For a detailed description of the TMCMC settings see *TMCMC*

Finally, we need to add a call to the run() routine to start the Korali engine.

```
k.run(e)
```

Running

We are now ready to run our example: *./run-tmcmc.py*

The results are saved in the folder *_korali_result/*.

Plottting

You can see a histogram of the results by running the command *python3 -m korali.plot –dir _korali_result_tmcmc*

Sampling with Nested Sampling

To sample the posterior distribution with the Nested sampler wa set a few settings,

```
e["Solver"]["Type"] = "Sampler/Nested"
e["Solver"]["Number Live Points"] = 1500
e["Solver"]["Resampling Method"] = "Ellipse" # (Default)
```

or instead use Mulit Ellipsoidal Sampling

```
e["Solver"]["Type"] = "Sampler/Nested"
e["Solver"]["Number Live Points"] = 1500
e["Solver"]["Resampling Method"] = "Multi Ellipse"
```

For a detailed description of the Nested Sampling settings see *Nested*

Finally, we need to add a call to the run() routine to start the Korali engine.

```
k.run(e)
```

Running

We are now ready to run our example: *./run-nested.py* respectively *./run-multinest.py*

Plottting

You can see a histogram of the results by running the command *python3 -m korali.plot –dir _korali_result_nested*

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/design/

1.7 Design

In this tutorial we show how to determine an optimal **design** for a given function. The example is based on section 5.1. of https://arxiv.org/abs/1108.4146.

1.7.1 Problem Description

We are given the measurement model $y(\vartheta, d) = g(\vartheta, d) + \epsilon = \vartheta^3 d^2 + \theta \exp(-|0.2 - d|) + \epsilon$ for $\epsilon \sim U(0, 1)$. We want to estimate the exptected information gain

$$U(d) = \int_{\mathcal{Y}} \int_{\mathcal{T}} \ln \frac{p(y \mid \vartheta, d)}{p(y \mid d)} p(y \mid \vartheta, d) p(\vartheta) \mathrm{d}\vartheta \mathrm{d}y$$

in the parameter $\vartheta \sim U(0, 1)$ of a measurement y for a given choice of design $d \in [0, 1]$.

1.7.2 The Model

Create a folder named *model*. Inside, create a file with name *model.py* which implements the model,

```
def model(s):
    theta = np.array(s["Parameters"])
    d = np.array(s["Designs"])
    res = theta * theta * theta * d * d + theta * np.exp(-np.abs(0.2 - d))
    s["Model Evaluation"] = res.tolist()
```

This implements $g(\vartheta, d)$ and we do not include the stochastic error.

1.7.3 The Problem Type

Then, we set the type of the problem to *Design* and set the measurement model

```
e["Problem"]["Type"] = "Design"
e["Problem"]["Model"] = model
```

1.7.4 The Variables

In this problem there is three variables: ϑ, d, and y. We want to regard designs $d \in [0, 1]$ at 101 equidistant points. For the parameter and measurements we take 10^5 samples $\vartheta^{(i)} \sim U(0, 1)$ and $y^{(i,j)}(d) = g(\vartheta^{(i)}, d) + \epsilon^{(j)}$. This is reflected in the configuration of the variables as follows:

```
e["Distributions"][0]["Name"] = "Uniform"
e["Distributions"][0]["Type"] = "Univariate/Uniform"
e["Distributions"][0]["Minimum"] = 0.0
e["Distributions"][0]["Maximum"] = 1.0

indx = 0
e["Variables"][indx]["Name"] = "d1"
e["Variables"][indx]["Type"] = "Design"
e["Variables"][indx]["Number Of Samples"] = 101
e["Variables"][indx]["Lower Bound"] = 0.0
e["Variables"][indx]["Upper Bound"] = 1.0
e["Variables"][indx]["Distribution"] = "Grid"
indx += 1

e["Variables"][indx]["Name"] = "theta"
e["Variables"][indx]["Type"] = "Parameter"
e["Variables"][indx]["Lower Bound"] = 0.0
e["Variables"][indx]["Upper Bound"] = 1.0
e["Variables"][indx]["Number Of Samples"] = 1e2
e["Variables"][indx]["Distribution"] = "Uniform"
indx += 1

e["Variables"][indx]["Name"] = "y1"
e["Variables"][indx]["Type"] = "Measurement"
e["Variables"][indx]["Number Of Samples"] = 1e2
indx += 1
```

1.7.5 The Solver

We choose the solver *Designer*, don't set the execution per generation, to have the evaluations of the model performed in one generation. Furthermore we set the standard deviation for the measurement error

```
e["Solver"]["Type"] = "Designer"
e["Solver"]["Sigma"] = 1e-2
```

1.7.6 Running

Finally, we need to add a call to the run() routine to start the Korali engine.

```
k.run(e)
```

In order to launch the example we use *python3 ./run-quadrature-integration.py*. Per default, the results are saved in the folder *_korali_result/*.

1.8 Hierarchical Bayesian

In this section, we show different ways to solve Hierarchical Bayesian problems.

Sub-Categories

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/hierarchical.bayesian/basic/

1.8.1 Hierarchical Bayesian Inference (Basic)

In this tutorial we show how to perform hierarchical Bayesian inference.

Hierarchical Bayesian Inference is set up in 3 phases:

- sample the posterior distributions conditioned on each data set
- sample the hyper parameter
- sample the posterior given hyperparameter and one (a) data set or (b) data sets combined.

For each phase we provided an individual *python* file.

Running

All 3 phases can be run with the shell script *./run-hierarchical.sh*

The results are saved in sub dirs of the folder */setup*.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/hierarchical.bayesian/extended/

1.8.2 Hierarchical Bayesian Inference (Extended)

In this tutorial we show how to perform hierarchical Bayesian inference.

Hierarchical Bayesian Inference is set up in 3 phases:

- sample the posterior distributions conditioned on each data set
- sample the hyper parameter
- sample the posterior given hyperparameter and one (a) data set or (b) data sets combined.

For each phase we provided an individual *python* file.

In phase 0 we generate synthetic data.

Running

All 3 phases can be run with the shell script

```
./run-hierarchical.sh
```

The results are saved in sub dirs of the folder */setup*.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/integration/

1.9 Integration

In this tutorial we show how to **integrate** a given function.

1.9.1 Problem Description

We are given the function $f(x, y, z) = x^2 + y^2 + z^2$ for $x, y, z \in [0, 1]^3$. We want to find the integral of this function over its domain.

1.9.2 The Integrand

Create a folder named *model*. Inside, create a file with name *integrands.py* and paste the following code,

```
def integrand( sample ):
  x = sample["Parameters"][0]
  y = sample["Parameters"][1]
  z = sample["Parameters"][2]
  sample["Evaluation"] = x**2+y**2+z**2
```

This is the function we want to integrate.

1.9.3 The Problem Type

Then, we set the type of the problem to *Integration*, set the function to integrate and chose the integration method

```
e["Problem"]["Type"] = "Integration"
e["Problem"]["Integrand"] = lambda s : integrand(s)
```

1.9.4 The Variables

In this problem there is three variables, *X*, *Y* and *Z*, whose domain we set to [0,1] and in case of Monte Carlo Integration assume an uniform distribution. Furthermore we assume 11 or 9 samples per dimension. Remember that the number of gridpoints must be odd for the Simpson method.

```
e["Variables"][0]["Name"] = "x"
e["Variables"][0]["Number Of Gridpoints"] = 11
e["Variables"][0]["Lower Bound"] = 0.0
e["Variables"][0]["Upper Bound"] = 1.0
```

(continues on next page)

(continued from previous page)

```
e["Variables"][1]["Name"] = "y"
e["Variables"][1]["Lower Bound"] = 0.0
e["Variables"][1]["Upper Bound"] = 1.0
e["Variables"][1]["Number Of Gridpoints"] = 9

e["Variables"][2]["Name"] = "z"
e["Variables"][2]["Lower Bound"] = 0.0
e["Variables"][2]["Upper Bound"] = 1.0
e["Variables"][2]["Number Of Gridpoints"] = 11
```

1.9.5 The Solver

We choose the solver *Integrator*, don't set the execution per generation, to have the summation be performed in one generation,

```
e["Solver"]["Type"] = "Integrator/Quadrature"
e["Solver"]["Method"] = "Simpson"
```

For a detailed description of Integrator settings see *Integrator*.

Finally, we need to add a call to the run() routine to start the Korali engine.

```
k.run(e)
```

1.9.6 Running

We are now ready to run our example: *python3 ./run-quadrature-integration.py*

The results are saved in the folder *_korali_result/*.

1.10 Optimization

In this section, we provide example optimization problems solved with Korali.

Sub-Categories

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/optimization/discrete/

1.10.1 Discrete Optimization: Searching the Global Maximum

In this tutorial we show how to optimize a function with discrete input parameters along with continuous ones.

Problem Description

We want to minimize the following function:

The Objective Function

We create a folder *model*, and inside a file *model.py*, where we define the function that we want to optimize:

```
def model(d):
  npar = 10
  res = 0.0
  v = d["Parameters"]

  for i in range(npar):
    if( i == 0 or i == 1 or i == 3 or i == 6):
        res += pow( 10, 6.0*i/npar) * round(v[i]) * round(v[i])
    else:
        res += pow( 10, 6.0*i/npar) * v[i] * v[i]

  d["Evaluation"] = -res;
```

Then, in another file, for example *run-cmaes.py*, we start by importing the function we just defined (assuming *model.py* is in subfolder *model* relative to the current file), and creating an *Experiment* which we will configure,

```
import sys
sys.path.append('model')
from model import *

import korali
e = korali.Experiment()
```

The Problem Type

We choose *direct evaluation* as problem type, set the objective function to our previously defined *model()*, and choose maximization as objective,

```
e["Problem"]["Type"] = "Evaluation/Direct/Basic"
e["Problem"]["Objective"] = "Maximize"
e["Problem"]["Objective Function"] = model
```

The Variables

We define 10 variables, of which four (x_1, x_2, x_4, x_7) are discrete. Also, we limit their domain to [-19, 21] each.

```
for i in range(10) :
  e["Variables"][i]["Name"] = "X" + str(i)
  e["Variables"][i]["Initial Mean"] = 1.0
  e["Variables"][i]["Lower Bound"]  = -19.0
  e["Variables"][i]["Upper Bound"]  = +21.0

# We set some of them as discrete.
e["Variables"][0]["Granularity"] = 1.0
e["Variables"][1]["Granularity"] = 1.0
```

(continues on next page)

(continued from previous page)

```
e["Variables"][3]["Granularity"] = 1.0
e["Variables"][6]["Granularity"] = 1.0
```

The Solver

We choose the solver *CMA-ES* and set two termination criteria,

```
e["Solver"]["Type"] = "Optimizer/CMAES"
e["Solver"]["Population Size"] = 8
e["Solver"]["Termination Criteria"]["Min Value Difference Threshold"] = 1e-9
e["Solver"]["Termination Criteria"]["Max Generations"] = 5000
```

Output configuration

To redcue output frequency of result files and on the console we set

```
e["File Output"]["Frequency"] = 50
e["Console Output"]["Frequency"] = 50
```

The Korali Engine Object

We create a Korali engine, and tell it to run the experiment we defined,

```
k = korali.Engine()
k.run(e)
```

Running

We are now ready to run our example: *./run-cmaes.py*

The results are saved in the folder *_korali_result/*.

Plotting

You can see the results of CMA-ES by running the command, *python3 -m korali.plot* which visualizes the results found in folder *_korali_result*.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/optimization/constrained/

1.10.2 Constrained Optimization: Searching the Global Maximum

In this tutorial we show how to solve a **constrained optimization** problem CEC 2006 Test Problem g09), defined as: Find $x^\star = \arg\min_x \; f(x)$, under the constraints $g_i(x) \leq 0$.

Problem Description

We want to solve the problem:

under the four constraints $g_i(x)$:

The Objective Function

Create a folder named *model*. Inside, create a file with name *model.py* and paste the following code,

```
#!/usr/bin/env python

def g09( k ):

  d = k["Parameters"]
  res = (d[0] - 10.0)**2 + 5.0 * (d[1] - 12.0)**2            \
        + d[2]**4  + 3.0 * (d[3] - 11.0)**2                  \
        + 10.0 * d[4]**6 + 7.0 * d[5]**2 + d[6]**4.       \
        - 4.0 * d[5] * d[6] - 10.0 * d[5] - 8.0 * d[6];

  k["Evaluation"] = -res;
```

This computational model represents our objective function.

For the constraints, add the following code in the same file,

```
def g1(k):
  v = k["Parameters"]
  k["Evaluation"] = -127.0 + 2 * v[0] * v[0] + 3.0 * pow(v[1], 4) + v[2] + 4.0 * v[3]␣
↪* v[3] + 5.0 * v[4]


def g2(k):
  v = k["Parameters"]
  k["Evaluation"] = -282.0 + 7.0 * v[0] + 3.0 * v[1] + 10.0 * v[2] * v[2] + v[3] -␣
↪v[4]


def g3(k):
  v = k["Parameters"]
  k["Evaluation"] = -196.0 + 23.0 * v[0] + v[1] * v[1] + 6.0 * v[5] * v[5] - 8.0 *␣
↪v[6]

def g4(k):
  v = k["Parameters"]
  k["Evaluation"] = 4.0 * v[0] * v[0] + v[1] * v[1] - 3.0 * v[0] * v[1] + 2.0 * v[2]␣
↪* v[2] + 5.0 * v[5] - 11.0 * v[6]
```

Optimization with (C)CMA-ES

First, open a file and import the korali module

```
#!/usr/bin/env python3
import korali
```

Import the computational model,

```
import sys
sys.path.append('./model')
from model import *
from constraints import *
```

The Korali Object

Next we construct a *korali.Experiment* object,

```
e = korali.Experiment()
```

Add the objective function and the constraints in the Korali object,

```
e["Problem"]["Objective Function"] = g09
e["Problem"]["Constraints"] = [ g1, g2, g3, g4 ]
```

The Problem Type

Then, we set the type of the problem to *Direct Evaluation*

```
e["Problem"]["Type"] = "Evaluation/Direct/Basic"
e["Problem"]["Objective"] = "Maximize"
```

The Variables

We add 7 variables to the experiment and set their domain,

```
for i in range(7) :
  e["Variables"][i]["Name"] = "X" + str(i)
  e["Variables"][i]["Lower Bound"] = -10.0
  e["Variables"][i]["Upper Bound"] = +10.0
```

The Solver

We choose the solver *CMA-ES*,

```
e["Solver"]["Type"] = "Optimizer/CMAES"
```

Then we set a few parameters for CCMA-ES,

```
e["Solver"]["Is Sigma Bounded"] = True
e["Solver"]["Population Size"] = 32
e["Solver"]["Viability Population Size"] = 4
e["Solver"]["Termination Criteria"]["Max Value"] = -680.630057374402 - 1e-4
e["Solver"]["Termination Criteria"]["Max Generations"] = 500
```

For a detailed description of CCMA-ES settings see *CMAES*.

We configure output settings,

```
e["File Output"]["Frequency"] = 50
e["Console Output"]["Frequency"] = 50
```

Finally, we need to create a Korali *Engine* object add a call to its run() routine, to start the engine.

```
k = korali.Engine()
k.run(e)
```

Running

We are now ready to run our example:*./run-ccmaes.py*

The results are saved in the folder *_korali_result/*.

Plotting

You can see the results of CMA-ES by running the command, *python3 -m korali.plot*

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/optimization/stochastic/

1.10.3 Optimization: Searching the Global Maximum

In this tutorial we show how to **optimize** a given function.

Problem Description

We are given the function $f(\vartheta) = -\vartheta^2$ for $\vartheta \in [-10, 10]$. We want to find the maximum of the function in the given interval.

The Objective Function

Create a folder named *model*. Inside, create a file with name *directModel.py* and paste the following code,

```
#!/usr/bin/env python

def evaluateModel(p):
    x = p["Parameters"][0]
    p["Evaluation"] = -x*x
```

This is the computational model that represents our objective function.

Optimization with CMAES

First, open a file (you could name it 'run-cmaes.py') and import the korali module

```
#!/usr/bin/env python3
import korali
```

Import the computational model,

```
import sys
sys.path.append('./model')
from directModel import *
```

The Korali Engine and Experiment Objects

Next we construct a *korali.Engine* and a *korali.Experiment* object and set the computational model,

```
k = korali.Engine()
e = korali.Experiment()

e["Problem"]["Objective Function"] = evaluateModel
```

The Problem Type

Then, we set the type of the problem to *Direct Evaluation*, and the objective to maximization,

```
e["Problem"]["Type"] = "Optimization"
```

The Variables

In this problem there is only one variable, *X*, whose domain we set to [-10,10],

```
e["Variables"][0]["Name"] = "X"
e["Variables"][0]["Lower Bound"] = -10.0
e["Variables"][0]["Upper Bound"] = +10.0
```

The Solver

We choose the solver *CMAES*, set the population size to be *32* and two termination criteria,

```
e["Solver"]["Type"] = "Optimizer/CMAES"
e["Solver"]["Population Size"] = 32
e["Solver"]["Termination Criteria"]["Min Value Difference Threshold"] = 1e-7
e["Solver"]["Termination Criteria"]["Max Generations"] = 100
```

For a detailed description see CMAES settings.

Finally, we need to add a call to the run() routine to start the Korali engine.

```
k.run(e)
```

Running

We are now ready to run our example:

```
./run-cmaes
```

Or, alternatively:

```
python3 ./run-cmaes
```

The results are saved in the folder *_korali_result/*.

Plotting

You can see the results of CMA-ES by running the command,

```
python3 -m korali.plot
```

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/optimization/multiobjective/

1.10.4 Multi-Objective Optimization: Searching the Pareto front

In this examples we show how to maximize multiple objectives simultaneously.

The solver (MOCMAES) returns a set of non-dominated samples.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/optimization/gradient/

1.10.5 Gradient-Based Optimization

In this tutorial we show how to **optimize** a given function that returns both the evaluation of the model F(x) as well as the gradient of the function at that point.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/propagation/

1.11 Model Propagation

In this tutorial we show how to pass multiple input sets through a computational model (this setup may be used to do uncertainty propagation and distribute the workload).

1.11.1 Problem Description

We are given a set of parameters to evaluate in a file 'samplesOut.dat'. We want to execute a model function $f(\theta)$ on given parameters.

1.11.2 Propagate Model Evaluations

1.11.3 The Korali Object

Initialize a korali object

```
e = korali.Experiment()
```

1.11.4 The Problem Type

The type of problem is *Execution/Model*.

```
e["Problem"]["Type"] = "Execution/Model"
```

1.11.5 The Variables

In the file we have means and variances to evaluate:

```
e["Variables"][0]["Name"] = "Mean"
 e["Variables"][0]["Loaded Values"] = means
 e["Variables"][1]["Name"] = "Variance"
 e["Variables"][1]["Loaded Values"] = variances
```

1.11.6 The Solver

We set the solver and choose how many samples are evaluated per generation.

```
e["Solver"]["Type"] = "Executor"
 e["Solver"]["Executions Per Generation"] = 1
```

1.11.7 Running

We are now ready to run our example: *./run-execution.py*

The results are saved in the folder *_korali_result/*.

1.12 Propagation of Uncertainty

In this tutorial we sample the posterior of a linear problem, as in the *Bayesian inference examples*. Then we evaluate the linear model for all the parameters that are in the sample database and different input values. We use these model evaluations to compute the uncertainty in the predictions by plotting credible intervals.

The first part is identical to the *Bayesian inference example*.

For the second part, first we define a new input variable

```
x = np.linspace(0,7,100)
```

These are the new points on which we are going to evaluate the linear model on all the samples from the previous step. We define a new function for the propagation that is similar to the one we used for the sampling phase

```
def model_propagation(s, X):

 a = s['Parameters'][0]
 b = s['Parameters'][1]

 s['sigma'] = s['Parameters'][2]
 s['X'] = X.tolist()
 s['Evaluations'] = []
 for x in X:
   s['Evaluations'] += [a * x + b]
```

The fields `'sigma'`, `'X'` and `'Evaluations'` are going to be saved for each sample under `['Samples'][k]`. In order for these variable to be saved we have to set

```
e['Store Sample Information'] = True
```

Next, we load the json file with the results from the sampling

```
with open('_korali_result_samples/latest') as f: d = json.load(f)
```

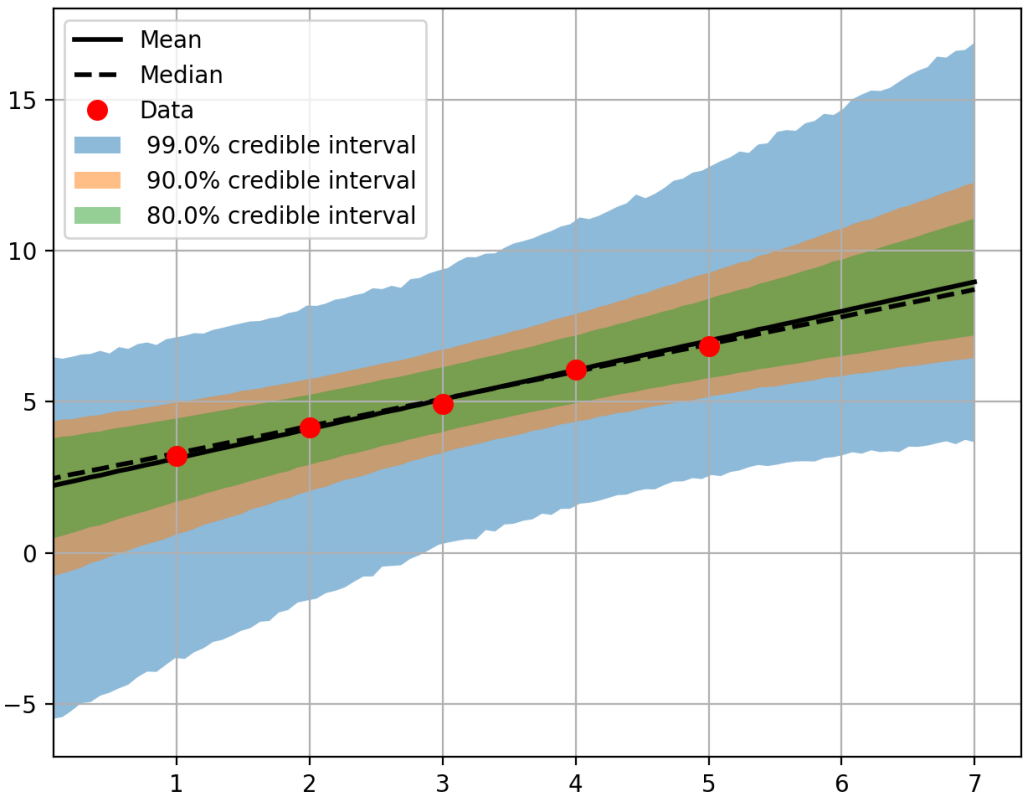
Then we copy the samples to the Korali variables

```
e['Variables'][0]['Name'] = 'a'
 e['Variables'][0]['Precomputed Values'] = [ x[0] for x in d['Results']['Sample␣
↪Database'] ]
 e['Variables'][1]['Name'] = 'b'
 e['Variables'][1]['Precomputed Values'] = [ x[1] for x in d['Results']['Sample␣
↪Database'] ]
 e['Variables'][2]['Name'] = 'sigma'
 e['Variables'][2]['Precomputed Values'] = [ x[2] for x in d['Results']['Sample␣
↪Database'] ]
```

Note that the samples are saved under `d['Results']['Sample Database']`. Finally we run the experiment.

Optionally, we can compute the credible intervals and plot the results. You have to uncomment the last two lines of the script.

```
from plots import *
 plot_credible_intervals('./_korali_result_propagation/latest', data)
```



Hint: Example code: https://github.com/cselab/korali/tree/master/examples/reaction/

1.13 Reaction: Simulating the evolution of reactants

In this tutorial we show how to **simulate** reactions using a stochastic simulator method (SSM).

1.13.1 The Problem Type

We set the type of the problem to *Reaction*

```
e["Problem"]["Type"] = "Reaction"
```

1.13.2 Problem Description

A set of reaction equations can be defined as follows:

```
e["Problem"]["Reactions"][0]["Equation"] = "[X1]->Y1"
e["Problem"]["Reactions"][0]["Rate"] = 0.1

e["Problem"]["Reactions"][1]["Equation"] = "[X2]+Y1->Y2+Z1"
e["Problem"]["Reactions"][1]["Rate"] = 0.1

e["Problem"]["Reactions"][2]["Equation"] = "2 Y1+Y2->3 Y1"
e["Problem"]["Reactions"][2]["Rate"] = 5e-5

e["Problem"]["Reactions"][3]["Equation"] = "Y1->Z2"
e["Problem"]["Reactions"][3]["Rate"] = 5.
```

Variables declared with enclosing square brackets, e.g. [X1], are considered reservoirs and remain unchanged during the simulation.

1.13.3 The Variables

Each reactant name must be declared as a variable with its initial number of reactants.

```
e["Variables"][0]["Name"] = "[X1]"
e["Variables"][0]["Initial Reactant Number"] = 50000

e["Variables"][1]["Name"] = "[X2]"
e["Variables"][1]["Initial Reactant Number"] = 500

e["Variables"][2]["Name"] = "Y1"
e["Variables"][2]["Initial Reactant Number"] = 1000

e["Variables"][3]["Name"] = "Y2"
e["Variables"][3]["Initial Reactant Number"] = 2000

e["Variables"][4]["Name"] = "Z1"
e["Variables"][4]["Initial Reactant Number"] = 0

e["Variables"][5]["Name"] = "Z2"
e["Variables"][5]["Initial Reactant Number"] = 0
```

1.13.4 The Solver

The solver method, the simulation length, the number of simulated trajectories and solver specific configurations are defined in the solver section of the korali application

```
e["Solver"]["Type"] = "SSM/SSA"
e["Solver"]["Simulation Length"] = 20.
e["Solver"]["Simulations Per Generation"] = 100
e["Solver"]["Termination Criteria"]["Max Num Simulations"] = 1000
e["Solver"]["Diagnostics"]["Num Bins"] = 500
```

For a detailed description of available solver settings see the SSA or TauLeaping documentation.

1.13.5 Output

The output directory and the number of output files can be configured as

```
e["File Output"]["Enabled"] = True
e["File Output"]["Path"] = '_korali_results'
e["File Output"]["Frequency"] = 1
```

1.13.6 Plotting

You can see the averaged trajectories of the SSM by running the command (trajectories are averaged in bins that have been previously defined)

```
python3 -m korali.plot --dir _korali_results
```

1.14 Deep Reinforcement Learning

Examples of Korali used for learning the best policy for a variety of reinforcement learning problems.

Sub-Categories

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/reinforcement.learning/upswing/

1.14.1 Upswing (Python)

Examples of different algorithms solving the upswing problem on python.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/reinforcement.learning/mountaincart/

1.14.2 Mountain Cart (Python)

In this example the goal of the RL agent is to move a cart in a U-shaped valley as high as possible. For that, it shall switch between applying left and right directed forces in order to benefit from gravity.

in _model/env.py set output = True and visualize states with script plotStates.py use following command for a simplistic visualization of states

```
ffmpeg -framerate 10 -pattern_type glob -i 'figures/*.png' -c:v libx264 -r 30 -pix_
↪fmt yuv420p cart.mp4
```

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/reinforcement.learning/cartpole/

1.14.3 Cartpole (Python)

Examples of different algorithms solving the cartpole balancing problem on python.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/reinforcement.learning/flowControl/

1.14.4 Multi-Agent Reinforcement Learning for Flow Control

Code-to-be-extended to MARL, based on

M. A. Bucci et al., Control of chaotic systems by deep reinforcement learning, Proceedings of the Royal Society A (2019), https://doi.org/10.1098/rspa.2019.0351

Example

A standalone simulation can be run

```
python main.py
```

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/reinforcement.learning/ABF2D/

1.14.5 Study Case: Reinforcement Learning on Artificial Bacterial Flagella (ABF) in 2D

A 2D version of artificial bacterial flagella (ABF) subjected to a rotating magnetic field. The goal is to bring those swimmers within a target circle by controlling a uniform magnetic field.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/reinforcement.learning/swarm/

1.14.6 Multi-Agent Reinforcement Learning on Active Particles

Ensemble of N point particles travelling at a fixed speed $|u| = 1$.

The reinforcement learning agent i that has available as *state* the distance r_{ij}, direction vector $\boldsymbol{r}_{ij}$, and angles θ_{ij} to the M nearest neighbours $j = 1, \ldots, M$. The action determines the wished new direction $u = (u_x, u_y, u_z)$ of the particle. The reward is computed as the sum of a pairwise potential between the neihest neighbours

$$r_t = \sum_{i=0}^{M} V(r)$$

As example potential is the Lennard-Jones potential as well as its harmonic approximation. The particles orientation is updated by rotating the current orientation by an angle α towards the wished new direction. Then it moves by updating the position as $x \to x + \Delta t u$.

Example

Verbous example with a trivial policy $a = [1, 0, 0]$ and visualisation turned on can be run using

```
python main.py --visualize 1 --numIndividuals 10 --numTimesteps 100 --
↪numNearestNeighbours 5
```

The Reinforcement Learning can be ran using

```
python run-vracer.py --numIndividuals 100 --numTimesteps 100 --numNearestNeighbours 3
```

In order to run the evaluation use

```
python eval-vracer.py --visualize 1 --numIndividuals 10 --numTimesteps 100 --
↪numNearestNeighbours 3
```

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/sampling/

1.15 Sampling

In this tutorial we show how to directly **sample** from a function using Metropolis Hastings (MCMC).

1.15.1 Problem Description

We are given the function $g(\vartheta) = \exp(-\vartheta^2)$ for $\vartheta \in [-10, 10]$.

We assume that f represents the *unnormalized* density of a distribution. We want to draw samples from this distribution.

For the rest of the tutorial we will work with the function $f(\vartheta) = \log g(\vartheta) = -\vartheta^2$ for numerical reasons. In general we advise users of Korali to work in log space.

1.15.2 The Objective Function

Create a folder named *model*. Inside, create a file with name *directModel.py* and paste the following code,

$$!/usr/bin/envpython$$
$$defevaluateModel(x) : v = x[”Parameters”][0]x[”Evaluation”] = -v * v$$

This is the computational model that represents our objective function.

1.15.3 Sampling with MCMC

First, open a file and import the korali module

```
#!/usr/bin/env python3
import korali
```

Import the computational model,

```
import sys
sys.path.append('./model')
from directModel import *
```

1.15.4 The Korali Experiment Object

Next we construct a *korali.Experiment* object and set the computational model,

```
e = korali.Experiment()
e["Problem"]["Objective Function"] = model
```

1.15.5 The Problem Type

Then, we set the type of the problem to *Direct Evaluation*

```
e["Problem"]["Type"] = "Evaluation/Direct/Basic"
```

1.15.6 The Variables

In this problem there is only one variable,

```
e["Variables"][0]["Name"] = "X"
```

1.15.7 The Solver

We choose the solver *MCMC* and set the initial mean and standard deviation of the parameter *X*.

```
e["Solver"]["Type"]  = "MCMC"
e["Variables"][0]["Initial Mean"] = 0.0
e["Variables"][0]["Initial Standard Deviation"] = 1.0


e["Solver"]["Burn In"] = 500
e["Solver"]["Termination Criteria"]["Max Samples"] = 5000
```

We also set some settings for MCMC. For a detailed description of the MCMC settings, see *MCMC*

1.15.8 Configuring the output

To reduce the output frequency we write

```
e["File Output"]["Frequency"]    = 500
e["Console Output"]["Frequency"] = 500
e["Console Output"]["Verbosity"] = "Detailed"
```

1.15.9 Running

Finally, we are ready to run the simulation,

```
k = korali.Engine()
k.run(e)
```

The results are saved in the folder *_korali_result/*.

1.15.10 Plotting

You can see a histogram of the results by running the command *python3 -m korali.plot*

1.16 Supervised Learning

Example problems for supervised learning.

Sub-Categories

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/supervised.learning/sineWave/

1.16.1 Sine Wave

These are simple examples of the use of NNs to approximate sine waves with (run-rnn.py) and without (run-ffn.py) time-dependence.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/supervised.learning/mnist/

1.16.2 Image Recognition (MNIST)

This example uses the MNIST database to recognize handrwitten numbers [0-9] from a training and testing set.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/checkpoint.resume/

1.17 Checkpoint / Resume

In this tutorial we show how to restart a previous Korali run.

1.17.1 Steps to Restart Execution

In this tutorial we show how to restart a previously saved checkpoint.

For more information, see *Checkpoint / Resume*.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/composite.korali/

1.18 Korali + Korali Model

This tutorial demonstrates that Korali can be composed: models evaluations can contain in themselves a new Korali experiment.

In this example, the main experiment describes an optimization over the parameter X. Each evaluation will optimize over the parameter Y to find the value of Y which maximizes X*Y.

Both, the main Korali and subsequent Korali instances run concurrently using 2 workers each, resulting in a 2 + 2x2 = 6 Korali workers instantiated.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/concurrent.execution/

1.19 Concurrent Execution

In this tutorial we show how an external model can be executed in parallel with Korali's concurrent conduit.

For more information on parallel execution, see *Parallel Execution*. For more information on the concurrent conduit execution, see *Concurrent Conduit*.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/dry.runs/

1.20 Partial Runs

In this example how the user can use dry runs to initialize a Korali experiment/engine without actually running it. This can be useful in testing, to make sure that the configuration is correct without the need to perform run of a model that might be too computationally demanding or require a special environment to run.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/multiple.experiments/

1.21 Running Multiple Experiments

In this tutorial we show how you can execute a series of experiments, in order to benefit from Korali's oversubscription capabilities.

Create a series of experiments:

```
for i in range(8):
  e = korali.Experiment()
  e["Problem"]["Type"] = "Evaluation/Bayesian/Inference/Reference"
  e["Problem"]["Likelihood Model"] = "Additive Normal"
  e["Problem"]["Reference Data"] = getReferenceData()
  e["Problem"]["Computational Model"] = lambda sampleData: model(sampleData,␣
↪getReferencePoints())

  # Configuring CMA-ES parameters
  e["Solver"]["Type"] = "Optimizer/CMAES"
  ...
```

1.21.1 Set Experiment Vector

We can store experiments in a list *eList*:

```
# Adding Experiment to vector
eList.append(e)
```

1.21.2 Run Experiment Vector

We can run all experiments in one Korali application

```
k.run(eList)
```

1.21.3 Running

We are now ready to run our example: *python3 ./run-cmaes*

1.21.4 Resuming Previous Run

Runs with multiple Korali experiments can also be resumed from a previous execution. For more information, see *Checkpoint / Resume*.

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/partial.runs/

1.22 Partial Runs

In this tutorial we show how to stop and restart a previous Korali run during the lifetime of an application (without the need of checkpoint files being stored). During the first and the second phase you may adjust a solvers settings. All scripts in this tutorial follow this structure:

1.22.1 Run

Set a *Termination Criteria* and run:

```
print('------------------------------------------------------')
print('Now running first 50 generations...')
print('------------------------------------------------------')

e["Solver"]["Termination Criteria"]["Max Generations"] = 50
k.run(e)
```

1.22.2 Restart

Update *Termination Criteria* and restart with *run*:

```
print('------------------------------------------------------')
print('Now running last 50 generations...')
print('------------------------------------------------------')

e["Solver"]["Termination Criteria"]["Max Generations"] = 100
k.run(e)
```

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/running.cxx/

1.23 Running C++

In this tutorial we show how Korali can be used with c++. For this we optimize a model with the solver *CMA-ES* and *LM-CMA*. Here we want to find the parameters $v = (Intensity, PosX, PosY, Sigma)$ that maximize the posterior in a Bayesian problem.

1.23.1 How to run the example

Run the *Makefile* to compile the executables. Then you can run am example, e.g. *./run-cmaes* This should output information about the process and result of the optimization.

1.23.2 Short explanation

The problem to be solved is a static heat conduction problem, with a candle as static heat source. The variables *Intensity* , *PosX*, *PosY* are position and intensity of the candle. *Sigma* is the standard deviation of the noise in the *Additive Normal* noise model - the noise ϵ that is added to the function f (heat2Dsolver, see below) to obtain the measured temperature at each data point.

1.23.3 Computational Model and Data Points

First, we create the Korali engine and an experiment that we will configure,

```
auto k = korali::Engine();
auto e = korali::Experiment();
auto p = heat2DInit(&argc, &argv);
```

Here, *heat2DInit*, defined in [heat2d.cpp](model/heat2d.cpp), returns the data points (triples (xPos, yPos, refTemp)) as *p*. We model refTemp as a function of xPos and yPos (a function whose parameters $v1$ we want to determine), in addition to some noise: $refTemp(xPos, yPos) = f_{v1}(xPos, yPos) + \epsilon$. The distribution of the noise ϵ depends on parameters $v2$. We want to estimate $v = (v1, v2)$.

We next set the problem type to Bayesian inference, assign the objective values (refTemp values) of our data as *Reference Data* and set the computational model to the function *heat2DSolver* (our f above), defined in [heat2d.cpp](model/heat2d.cpp),

```
e["Problem"]["Type"] = "Evaluation/Bayesian/Inference/Reference";
e["Problem"]["Likelihood Model"] = "Additive Normal";
e["Problem"]["Reference Data"] = p.refTemp;
e["Problem"]["Computational Model"] = &heat2DSolver;
```

Function [*heat2DSolver*](model/heat2d.cpp) internally already has access to the data points created by *heat2DInit*. The function calculates temperature values iteratively on a grid over the domain of xPos and yPos, using the Gauss-Seidel method. To get the temperature values at the data points and set them as *Reference Evaluations*, *heat2DSolver* finds a point on the grid close to each data point and returns the temperature value at this grid point.

1.23.4 Solver

Then, we decide on *CMAES* as solver and configure its parameters,

```
e["Solver"]["Type"] = "Optimizer/CMAES";
e["Solver"]["Population Size"] = 32;
e["Solver"]["Termination Criteria"]["Max Generations"] = 100;
```

1.23.5 Variables and Prior Distributions

We then need to define four variables, as well as a prior distribution for each of them,

```
e["Distributions"][0]["Name"] = "Uniform 0";
e["Distributions"][0]["Type"] = "Univariate/Uniform";
e["Distributions"][0]["Minimum"] = 10.0;
e["Distributions"][0]["Maximum"] = 60.0;

e["Distributions"][1]["Name"] = "Uniform 1";
e["Distributions"][1]["Type"] = "Univariate/Uniform";
```

(continues on next page)

(continued from previous page)

```
e["Distributions"][1]["Minimum"] = 0.0;
e["Distributions"][1]["Maximum"] = 0.5;

e["Distributions"][2]["Name"] = "Uniform 2";
e["Distributions"][2]["Type"] = "Univariate/Uniform";
e["Distributions"][2]["Minimum"] = 0.6;
e["Distributions"][2]["Maximum"] = 1.0;

e["Distributions"][3]["Name"] = "Uniform 3";
e["Distributions"][3]["Type"] = "Univariate/Uniform";
e["Distributions"][3]["Minimum"] = 0.0;
e["Distributions"][3]["Maximum"] = 20.0;

e["Variables"][0]["Name"] = "Intensity";
e["Variables"][0]["Bayesian Type"] = "Computational";
e["Variables"][0]["Prior Distribution"] = "Uniform 0";
e["Variables"][0]["Initial Mean"] = 30.0;
e["Variables"][0]["Initial Standard Deviation"] = 5.0;

e["Variables"][1]["Name"] = "PosX";
e["Variables"][1]["Bayesian Type"] = "Computational";
e["Variables"][1]["Prior Distribution"] = "Uniform 1";
e["Variables"][1]["Initial Mean"] = 0.25;
e["Variables"][1]["Initial Standard Deviation"] = 0.01;

e["Variables"][2]["Name"] = "PosY";
e["Variables"][2]["Bayesian Type"] = "Computational";
e["Variables"][2]["Prior Distribution"] = "Uniform 2";
e["Variables"][2]["Initial Mean"] = 0.8;
e["Variables"][2]["Initial Standard Deviation"] = 0.1;

e["Variables"][3]["Name"] = "Sigma";
e["Variables"][3]["Bayesian Type"] = "Statistical";
e["Variables"][3]["Prior Distribution"] = "Uniform 3";
e["Variables"][3]["Initial Mean"] = 10.0;
e["Variables"][3]["Initial Standard Deviation"] = 1.0;
```

1.23.6 Running the Optimization

Finally, we call the *run()* routine to run the optimization, to find those parameters v that are most likely, using Bayes rule: We want to find v that maximize $P(v|X) = P(X|v) * prior(v)$, i.e, the likelihood of the data times their prior.

```
k.run(e);
```

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/running.mpi.cxx/

1.24 Running C++ MPI Applications

In this tutorial we show how a C++ MPI model can be executed with Korali.

For more information on running Korali applications in parallel, see *Parallel Execution*. For more information on running Korali on MPI, see *Distributed Conduit*.

1.24.1 MPI Init

Do not forget to init MPI inside the Korali application:

```
MPI_Init(&argc, &argv);
```

1.24.2 Distributed Conduit

Run with the *Distributed* conduit to benefit from parallelized model evaluations. Note that we need to provide it with the MPI communicator we want to use for this instance of Korali. Next, we set *Ranks Per Worker* to determine how many MPI ranks will be assigned to each Korali worker. This particular example uses *n* MPI ranks per worker, where *n* is passed by argument.

```
k.setMPIComm(MPI_COMM_WORLD);
k["Conduit"]["Type"] = "Distributed";
k["Conduit"]["Ranks Per Worker"] = n;
```

1.24.3 Profiling

In some cases it might be useful to activate Korali's internal profiler to analyze how efficiently workers executed. To enable it, add the following option:

```
k["Profiling"]["Detail"] = "Full";
k["Profiling"]["Frequency"] = 0.5;
```

1.24.4 Computational Model

If the computational model requires communication between the MPI ranks, you need to obtain the worker-specific sub-communicator

```
#include "mpi.h"
#include <korali.hpp>
...
int myRank, rankCount;
MPI_Comm comm = *(MPI_Comm*) korali::getWorkerMPIComm();
MPI_Comm_rank(comm, &myRank);
MPI_Comm_size(comm, &rankCount);
```

Note that all MPI ranks shall write results to the sample object.

1.24.5 Run

To launch korali, use the corresponding MPI launcher, with a number of MPI ranks that equals k*n+1, where *k* is the number of Korali workers to use, *n* is the number of MPI Ranks per worker, and 1 MPI rank is assigned to the Korali engine. In this example, we launch two workers with 4 ranks each, hence we need 9 MPI ranks.

```
mpirun -n 9 ./run-cmaes 4
```

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/running.mpi.python/

1.25 Running Python MPI Applications

In this tutorial we show how a Python MPI model can be executed with Korali.

For more information on running Korali applications in parallel, see *Parallel Execution*. For more information on running Korali on MPI, see *Distributed Conduit*.

1.25.1 MPI Init

Do not forget to init MPI inside the Korali application:

```
from mpi4py import MPI
```

1.25.2 Distributed Conduit

Run with the *Distributed* conduit to benefit from parallelized model evaluations. Note that we need to provide it with the MPI communicator we want to use for this instance of Korali. Next, we set *Ranks Per Worker* to determine how many MPI ranks will be assigned to each Korali worker. This particular example uses 4 MPI ranks per worker.

```
k.setMPIComm(MPI_COMM_WORLD);
k["Conduit"]["Type"] = "Distributed";
k["Conduit"]["Ranks Per Worker"] = 4;
```

1.25.3 Profiling

In some cases it might be useful to activate Korali's internal profiler to analyze how efficiently workers executed. To enable it, add the following option:

```
k["Profiling"]["Detail"] = "Full";
k["Profiling"]["Frequency"] = 0.5;
```

1.25.4 Computational Model

If the computational model requires communication between the MPI ranks, you need to obtain the worker-specific sub-communicator

```
import korali
...
comm = korali.getWorkerMPIComm()
rank = comm.Get_rank()
size = comm.Get_size()
```

1.25.5 Run

To launch korali, use the corresponding MPI launcher, with a number of MPI ranks that equals k*n+1, where *k* is the number of Korali workers to use, *n* is the number of MPI Ranks per worker, and 1 MPI rank is assigned to the Korali engine. In this example, we launch two workers with 4 ranks each, hence we need 9 MPI ranks.

```
mpirun -n 9 ./run-cmaes
```

Hint: Example code: https://github.com/cselab/korali/tree/master/examples/features/save.results/

1.26 Preserving Results

In this tutorial we show how Korali can be used to store important outputs from the computational model that otherwise would be lost after evaluation.

1.26.1 Computational Model

In the model we assign quantity of interest (QoI) to the Korali *Sample*:

```
# Store QoI
d["Apples"]  = a
d["Bananas"] = b
```

1.26.2 Execute

In Korali we have to set this flag to store the values of *Apples* and *Bananas*:

```
e["Store Sample Information"] = True
```

All evaluations can be found in the files in *_korali_results*.

To access the saved result of the first sample of the 8th generation, execute

```
import json
with open('gen00000008.json') as f:
  data = json.load(f)
```

(continues on next page)

(continued from previous page)

```
data["Samples"][0]["Apples"]
data["Samples"][0]["Bananas"]
```

1.27 Experiment

A Korali Experiment describes the Problem to be solved, and the method to use to solve it. A detailed explanation on how to configure and run an experiment, see *Korali Usage Basics*.

1.27.1 Usage

```
k = korali.Experiment()
```

1.27.2 Configuration

These are settings required by this module.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Specifies the initializing seed for the generation of random numbers. If 0 is specified, Korali will automatically derivate a new seed base on the current time.

Preserve Random Number Generator States

- **Usage**: e["Preserve Random Number Generator States"] = *True/False*
- **Description**: Indicates that the engine must preserve the state of their RNGs for reproducibility purposes.

Distributions

- **Usage**: e["Distributions"] = List of *Distribution/Univariate*
- **Description**: Represents the distributions to use during execution.

Variables

- **Usage**: e["Variables"] = List of *Variable*
- **Description**: Sample coordinate information.

Problem

- **Usage**: e["Problem"] = *Problem*
- **Description**: Represents the configuration of the problem to solve.

Solver

- **Usage**: e["Solver"] = *Solver*
- **Description**: Represents the state and configuration of the solver algorithm.

File Output / Path

- **Usage**: e["File Output"]["Path"] = *string*
- **Description**: Specifies the path of the results directory.

File Output / Use Multiple Files

- **Usage**: e["File Output"]["Use Multiple Files"] = *True/False*
- **Description**: If true, Korali stores a different generation file per generation with incremental numbering. If disabled, Korali stores the latest generation files into a single file, overwriting previous results.

File Output / Enabled

- **Usage**: e["File Output"]["Enabled"] = *True/False*
- **Description**: Specifies whether the partial results should be saved to the results directory.

File Output / Frequency

- **Usage**: e["File Output"]["Frequency"] = *unsigned integer*
- **Description**: Specifies how often (in generations) will partial result files be saved on the results directory. The default, 1, indicates that every generation's results will be saved. 0 indicates that only the latest is saved.

Store Sample Information

- **Usage**: e["Store Sample Information"] = *True/False*
- **Description**: Specifies whether the sample information should be saved to samples.json in the results path.

Console Output / Verbosity

- **Usage**: e["Console Output"]["Verbosity"] = *string*
- **Description**: Specifies how much information will be displayed on console when running Korali.
- **Options**:
  - "*Silent*": Prints no information to console, except in case of errors.
  - "*Minimal*": Prints minimal information about the progress of the engine.
  - "*Normal*": Prints information about the progress of the engine, plus information on the solver/problem.
  - "*Detailed*": Prints detailed information about the progress of the engine, plus detailed information on the solver/problem.

Console Output / Frequency

- **Usage**: e["Console Output"]["Frequency"] = *unsigned integer*
- **Description**: Specifies how often (in generations) will partial results be printed on console. The default, 1, indicates that every generation's results will be printed.

1.27.3 Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Console Output": {
    "Frequency": 1,
    "Verbosity": "Normal"
    },
"Current Generation": 0,
"Distributions": [],
"File Output": {
    "Enabled": true,
```

(continues on next page)

(continued from previous page)

```
    "Frequency": 1,
    "Path": "_korali_result",
    "Use Multiple Files": true
    },
"Is Finished": false,
"Preserve Random Number Generator States": false,
"Random Seed": 0,
"Store Sample Information": false
}
```

1.28 Problems

Problem modules describe a statistical/learning/other problem to be solved by Korali. One or more *solvers* may exist (or be added to Korali) that can be used to solve a particular problem. To specify a problem type, use the following syntax:

```
e = korali.Experiment()
e["Problem"]["Type"] = "Optimization"
```

For more information, see *Korali Usage Basics*.

Sub-Categories:

1.28.1 Probability Distribution Sampling

Usage

```
e["Problem"]["Type"] = "Sampling"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Sampler/MCMC*
- *Sampler/HMC*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Probability Function

- **Usage**: e["Problem"]["Probability Function"] = *Computational Model*
- **Description**: Stores the probability distribution function to evaluate.

1.28.2 Supervised Learning

Describes a problem where parameters of a function approximator are optimized such that they minimize a given *Loss Function*. The user provides *Training Data* and *Validation Data* for cross-validation.

Usage

```
e["Problem"]["Type"] = "SupervisedLearning"
```

Compatible Solvers

This problem can be solved using the following modules:

- *DeepSupervisor*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Training Batch Size

- **Usage**: e["Problem"]["Training Batch Size"] = *unsigned integer*
- **Description**: Stores the batch size of the training dataset.

Testing Batch Size

- **Usage**: e["Problem"]["Testing Batch Size"] = *unsigned integer*
- **Description**: Stores the batch size of the testing dataset.

Max Timesteps

- **Usage**: e["Problem"]["Max Timesteps"] = *unsigned integer*
- **Description**: Stores the length of the sequence for recurrent neural networks.

Input / Data

- **Usage**: e["Problem"]["Input"]["Data"] = List of Lists of List of float
- **Description**: Provides the input data with layout T*N*IC, where T is the sequence length, N is the batch size and IC is the vector size of the input.

Input / Size

- **Usage**: e["Problem"]["Input"]["Size"] = *unsigned integer*
- **Description**: Indicates the vector size of the input (IC).

Solution / Data

- **Usage**: e["Problem"]["Solution"]["Data"] = List of Lists of float
- **Description**: Provides the solution for one-step ahead prediction with layout N*OC, where N is the batch size and OC is the vector size of the output.

Solution / Size

- **Usage**: e["Problem"]["Solution"]["Size"] = *unsigned integer*
- **Description**: Indicates the vector size of the output (OC).

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Input": {
    "Data": []
    },
"Max Timesteps": 1,
"Solution": {
    "Data": []
    }
}
```

1.28.3 Optimization

Solves the optimization problem of continuous/discrete variables, given a model function $f(x)$ given the form:

$$\begin{aligned} \underset{x}{\text{maximize}} \quad & f(x) && (1.1) \\ \text{subject to} \quad & g_i(x) \leq 0, \quad i = 1, \ldots, n && (1.2) \end{aligned}$$

Where:

- $f : \mathbb{R}^n \to \mathbb{R}$: is the *objective function* to be maximized over the n-variable vector x
- $g_i(x) \leq 0$: are a set of inequality contraints to be satisfied.

Usage

```
e["Problem"]["Type"] = "Optimization"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Optimizer*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Num Objectives

- **Usage**: e["Problem"]["Num Objectives"] = *unsigned integer*
- **Description**: Number of return values to expect from objective function.

Objective Function

- **Usage**: e["Problem"]["Objective Function"] = *Computational Model*
- **Description**: Stores the function to evaluate.

Constraints

- **Usage**: e["Problem"]["Constraints"] = List of *Computational Model*
- **Description**: Stores constraints to the objective function.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Constraints": [],
"Has Discrete Variables": false,
"Num Objectives": 1
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{    }
```

1.28.4 Simulation of Reaction

Usage

```
e["Problem"]["Type"] = "Reaction"
```

Compatible Solvers

This problem can be solved using the following modules:

- *SSM*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Initial Reactant Number

- **Usage**: e["Variables"][*index*]["Initial Reactant Number"] = *integer*
- **Description**: The initial amount of the reactant.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Reactant Number": 0
}
```

1.28.5 Reinforcement Learning

Describes a sequential decision making problem. We are given an environment that transitions to a state s' and returns a reward r for a given action a and state s with probability $p(s', r|s, a)$. We want to find the policy π that choses an action a for a given state s with probability $\pi(a|s)$ such that for every state s the chosen action a is such that the value function

$$V^{\pi}(s) = \underset{\substack{a_t \sim \pi(\cdot|s_t) \\ s_{t+1}, r_t \sim p(\cdot,\cdot|s_t,a_t)}}{} \left[\sum_{t=0}^{\infty} \gamma^t r_t \middle| s_0 = s \right]$$

is maximal. Here γ is the discount factor.

We distinguish discrete and continuous action domains. **Sub-Categories:**

Discrete Reinforcement Learning

Specialization of the Reinforcement Learning Problem for continuous action domains.

Usage

```
e["Problem"]["Type"] = "ReinforcementLearning/Discrete"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Agent/Discrete*
- *Agent*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Type

- **Usage**: e["Variables"][*index*]["Type"] = *string*
- **Description**: Indicates if the variable belongs to the state or action vector.
- **Options**:
  - "*State*": The variable describes a state.
  - "*Action*": The variable describes an action.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = float
- **Description**: Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = float
- **Description**: Upper bound for the variable's value.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Possible Actions

- **Usage**: e["Problem"]["Possible Actions"] = List of Lists of float
- **Description**: The set of all possible actions.

Agents Per Environment

- **Usage**: e["Problem"]["Agents Per Environment"] = *unsigned integer*
- **Description**: Number of agents in a given environment .

Policies Per Environment

- **Usage**: e["Problem"]["Policies Per Environment"] = *unsigned integer*
- **Description**: Number of policies in a given environment. All agents share the same policy or all have individual policy.

Environment Count

- **Usage**: e["Problem"]["Environment Count"] = *unsigned integer*
- **Description**: Maximum number of different types of environments.

Environment Function

- **Usage**: e["Problem"]["Environment Function"] = *Computational Model*
- **Description**: Function to initialize and run an episode in the environment.

Actions Between Policy Updates

- **Usage**: e["Problem"]["Actions Between Policy Updates"] = *unsigned integer*
- **Description**: Number of actions to take before requesting a new policy.

Testing Frequency

- **Usage**: e["Problem"]["Testing Frequency"] = *unsigned integer*
- **Description**: Number of episodes after which the policy will be tested.

Policy Testing Episodes

- **Usage**: e["Problem"]["Policy Testing Episodes"] = *unsigned integer*
- **Description**: Number of test episodes to run the policy (without noise) for, for which the average sum of rewards will serve to evaluate the termination criteria.

Custom Settings

- **Usage**: e["Problem"]["Custom Settings"] = knlohmann::json
- **Description**: Any used-defined settings required by the environment.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Actions Between Policy Updates": 0,
"Agents Per Environment": 1,
"Custom Settings": {    },
"Environment Count": 1,
"Policies Per Environment": 1,
"Policy Testing Episodes": 10,
"Testing Frequency": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Lower Bound": -Infinity,
"Type": "State",
"Upper Bound": Infinity
}
```

Continuous Reinforcement Learning

Specialization of the Reinforcement Learning Problem for continuous action domains.

Usage

```
e["Problem"]["Type"] = "ReinforcementLearning/Continuous"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Agent/Continuous*
- *Agent*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Type

- **Usage**: e["Variables"][*index*]["Type"] = *string*
- **Description**: Indicates if the variable belongs to the state or action vector.
- **Options**:
  - "*State*": The variable describes a state.
  - "*Action*": The variable describes an action.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = float
- **Description**: Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = float
- **Description**: Upper bound for the variable's value.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Agents Per Environment

- **Usage**: e["Problem"]["Agents Per Environment"] = *unsigned integer*
- **Description**: Number of agents in a given environment .

Policies Per Environment

- **Usage**: e["Problem"]["Policies Per Environment"] = *unsigned integer*
- **Description**: Number of policies in a given environment. All agents share the same policy or all have individual policy.

Environment Count

- **Usage**: e["Problem"]["Environment Count"] = *unsigned integer*
- **Description**: Maximum number of different types of environments.

Environment Function

- **Usage**: e["Problem"]["Environment Function"] = *Computational Model*
- **Description**: Function to initialize and run an episode in the environment.

Actions Between Policy Updates

- **Usage**: e["Problem"]["Actions Between Policy Updates"] = *unsigned integer*

- **Description**: Number of actions to take before requesting a new policy.

Testing Frequency

- **Usage**: e["Problem"]["Testing Frequency"] = *unsigned integer*
- **Description**: Number of episodes after which the policy will be tested.

Policy Testing Episodes

- **Usage**: e["Problem"]["Policy Testing Episodes"] = *unsigned integer*
- **Description**: Number of test episodes to run the policy (without noise) for, for which the average sum of rewards will serve to evaluate the termination criteria.

Custom Settings

- **Usage**: e["Problem"]["Custom Settings"] = knlohmann::json
- **Description**: Any used-defined settings required by the environment.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Actions Between Policy Updates": 0,
"Agents Per Environment": 1,
"Custom Settings": {    },
"Environment Count": 1,
"Policies Per Environment": 1,
"Policy Testing Episodes": 10,
"Testing Frequency": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Lower Bound": -Infinity,
"Type": "State",
"Upper Bound": Infinity
}
```

1.28.6 Hierarchical Bayesian Inference

Sub-Categories:

Hierarchical Bayesian (Theta)

Usage

```
e["Problem"]["Type"] = "Hierarchical/Theta"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Sampler*
- *Optimizer*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Prior Distribution

- **Usage**: e["Variables"][*index*]["Prior Distribution"] = *string*
- **Description**: Determines the name of the distribution to use as prior distribution.

Distribution Index

- **Usage**: e["Variables"][*index*]["Distribution Index"] = *unsigned integer*
- **Description**: Determines the index number of the selected prior distribution.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Sub Experiment

- **Usage**: e["Problem"]["Sub Experiment"] = knlohmann::json
- **Description**: Results from one previously executed Bayesian experiment.

Psi Experiment

- **Usage**: e["Problem"]["Psi Experiment"] = knlohmann::json
- **Description**: Results from the hierarchical problem (Psi).

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Distribution Index": 0
}
```

Hierarchical Bayesian (Theta New)

Usage

```
e["Problem"]["Type"] = "Hierarchical/ThetaNew"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Sampler*
- *Optimizer*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Prior Distribution

- **Usage**: e["Variables"][*index*]["Prior Distribution"] = *string*
- **Description**: Determines the name of the distribution to use as prior distribution.

Distribution Index

- **Usage**: e["Variables"][*index*]["Distribution Index"] = *unsigned integer*
- **Description**: Determines the index number of the selected prior distribution.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Psi Experiment

- **Usage**: e["Problem"]["Psi Experiment"] = knlohmann::json
- **Description**: Results from the hierarchical Psi experiment.

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Distribution Index": 0
}
```

Hierarchical Bayesian (Psi)

Usage

```
e["Problem"]["Type"] = "Hierarchical/Psi"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Sampler*
- *Optimizer*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Prior Distribution

- **Usage**: e["Variables"][*index*]["Prior Distribution"] = *string*
- **Description**: Determines the name of the distribution to use as prior distribution.

Distribution Index

- **Usage**: e["Variables"][*index*]["Distribution Index"] = *unsigned integer*
- **Description**: Determines the index number of the selected prior distribution.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Sub Experiments

- **Usage**: e["Problem"]["Sub Experiments"] = List of knlohmann::json
- **Description**: Provides results from previous Bayesian Inference sampling experiments.

Conditional Priors

- **Usage**: e["Problem"]["Conditional Priors"] = List of *string*
- **Description**: List of conditional priors to use in the hierarchical problem.

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Distribution Index": 0
}
```

1.28.7 Propagation

Usage

```
e["Problem"]["Type"] = "Propagation"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Executor*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Precomputed Values

- **Usage**: e["Variables"][*index*]["Precomputed Values"] = List of *real number*
- **Description**: Contains predetermined values for the variables to evaluate.

Prior Distribution

- **Usage**: e["Variables"][*index*]["Prior Distribution"] = *string*
- **Description**: Indicates the name of the distribution to use as prior distribution.

Distribution Index

- **Usage**: e["Variables"][*index*]["Distribution Index"] = *unsigned integer*

- **Description**: Stores the the index number of the selected prior distribution.

Sampled Values

- **Usage**: e["Variables"][*index*]["Sampled Values"] = List of *real number*
- **Description**: Contains values sampled from prior.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Execution Model

- **Usage**: e["Problem"]["Execution Model"] = *Computational Model*
- **Description**: Stores the function to evaluate.

Number Of Samples

- **Usage**: e["Problem"]["Number Of Samples"] = *unsigned integer*
- **Description**: Number of samples to draw from Prior distribution.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Number Of Samples": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Distribution Index": 0,
"Precomputed Values": [],
"Prior Distribution": " ",
"Sampled Values": []
}
```

1.28.8 Integration

The integration problem is considering the numerical approximation of integrals

$$I = \int f(x)\mathrm{d}x$$

using

$$I \approx \sum_{i=1}^{N} w_i f(x_i)$$

The supported methods are Monte Carlo Integration and Quadrature. For Quadrature the weights for the Rectangle rule, the Trapezoidal rule and the Simpson rule are given, and there is the possibility to provide own weights and evaluation points.

Usage

```
e["Problem"]["Type"] = "Integration"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Integrator*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Integrand

- **Usage**: e["Problem"]["Integrand"] = *Computational Model*
- **Description**: Stores the function to integrate.

1.28.9 Design

The design problem considers the expected information gain of measurements for the experimental design s, given by

$$U(s) = \int_{\mathcal{Y}} \int_{\mathcal{T}} \ln \frac{p(y \mid \vartheta, s)}{p(y \mid s)} p(y \mid \vartheta, s) p(\vartheta) \mathrm{d}\vartheta \mathrm{d}y$$

where y denotes the measurements for parameters ϑ. The goal is to determine the optimal experimental design

$$s^\star = \arg\max_s U(s)$$

Usage

```
e["Problem"]["Type"] = "Design"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Designer*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Type

- **Usage**: e["Variables"][*index*]["Type"] = *string*
- **Description**: Indicates what the variable descibes.
- **Options**:
  - "*Design*": The variable describes a design.
  - "*Parameter*": The variable describes an parameter.
  - "*Measurement*": The variable describes an measurement.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: Upper bound for the variable's value.

Distribution

- **Usage**: e["Variables"][*index*]["Distribution"] = *string*
- **Description**: Indicates the distribution of the variable.

Number Of Samples

- **Usage**: e["Variables"][*index*]["Number Of Samples"] = *unsigned integer*

- **Description**: Number of Samples per Direction.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Model

- **Usage**: e["Problem"]["Model"] = *Computational Model*
- **Description**: Stores the model function.

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Distribution": " ",
"Lower Bound": -Infinity,
"Number Of Samples": 0,
"Upper Bound": Infinity
}
```

1.28.10 Bayesian Inference

In a *Bayesian Inference* problem, we have a probability model that consists of a conditional probability $p(d|\vartheta)$ of data d given a set of variables ϑ, and a prior $p(\vartheta)$ for the problem variables.

The solver is applied to the posterior distribution of the problem variables:

$$p(\vartheta|d) = \frac{p(d|\vartheta)p(\vartheta)}{p(d)}$$

Subtypes

For data stemming from a computational model, for which you only want to choose a noise distribution and variable prior, a Bayesian *Reference Likelihood* problem can be used.

A *Custom Likelihood* problem allows any kind of user defined probability model $p(d|\vartheta)$.

If your problem has additional (latent) variables in addition to the parameters of interest, the subtypes of Bayesian *Latent Variable* problems can be used.

Sub-Categories:

Likelihood by Reference

A Bayesian *Reference* problem is for data that originate from a computational model f:

$$d = (x_j, y_j)_{j=1...N} \; with$$
$$y_j = f(x_j) + \epsilon$$

The distribution of noise ϵ defines the likelihood model of the data. You can choose between three types of noise likelihood models: *Normal*, *Negative Binomial* and *Positive Normal*.

The following likelihood functions are available in Korali:

Normal

$$p(d|\vartheta) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}((x-\mu)/\sigma)^2}$$

where μ is the mean and σ is the Standard Deviation.

Positive Normal

The *Normal* likelihood truncated at 0.

StudentT

$$p(d|\vartheta) = \frac{\Gamma((n+1)/2)}{\sqrt{n\pi}\Gamma(n/2)} (1 + d^2/n)^{-(n+1)/2}$$

where n is refered to as Degrees Of Freedom.

Positive StudentT

The *StudentT* likelihood truncated at 0.

Poisson

$$p(d|\vartheta) = \frac{\lambda^d e^{-\lambda}}{d!}$$

where λ is the mean.

Geometric

$$p(d|\vartheta) = \lambda(1-\lambda)^{d-1}$$

where λ is the mean.

Negative Binomial

$$p(d|\vartheta) = \binom{d+r-1}{d} p^r (1-p)^d$$

where p is the success probability and r is the dispersion parameter.

Usage

```
e["Problem"]["Type"] = "Bayesian/Reference"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Sampler*
- *Optimizer*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Prior Distribution

- **Usage**: e["Variables"][*index*]["Prior Distribution"] = *string*
- **Description**: Indicates the name of the distribution to use as prior distribution.

Distribution Index

- **Usage**: e["Variables"][*index*]["Distribution Index"] = *unsigned integer*
- **Description**: Stores the the index number of the selected prior distribution.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Computational Model

- **Usage**: e["Problem"]["Computational Model"] = *Computational Model*
- **Description**: Stores the computational model. It should the evaluation of the model at the given reference data points.

Reference Data

- **Usage**: e["Problem"]["Reference Data"] = List of *real number*
- **Description**: Reference data required to calculate likelihood. Model evaluations are compared against these data.

Likelihood Model

- **Usage**: e["Problem"]["Likelihood Model"] = *string*
- **Description**: Specifies the likelihood model to approximate the reference data to.
- **Options**:
  - "*Normal*": The user specifies the mean and the standard deviation of the normal likelihood.
  - "*Positive Normal*": The user specifies the mean and the standard deviations of the truncated normal on [0,infty]
  - "*StudentT*": The user specifies the degrees of freedom (>0) of the Student's t-distribution.
  - "*Positive StudentT*": The user specifies the degrees of freedom (>0) of the half Student's t-distribution.
  - "*Poisson*": The user specifies the mean (>0) of the Poisson distribution.
  - "*Geometric*": The user specifies the inverse mean (>0) of the Geometric distribution.
  - "*Negative Binomial*": The user specifies the mean and the dispersion parameter of the Negative Binomial distribution.

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Distribution Index": 0
}
```

Custom Likelihood

While a Bayesian *Reference* type problem is for data that originate from a functional dependency, $d = (x_j, y_j)_{j=1...N}$ with $y_j = f(x_j) + \epsilon$, a *Custom Likelihood* model makes no such assumption.

With a *Custom Likelihood*, the function $p(d|\vartheta)$ is given directly by a user-defined model of the form $f : \mathbb{R}^N \rightarrow \mathbb{R}$, where N is the number of variables.

Likelihood Models

Additive Normal Likelihood

Whereas with an *Additive Normal Likelihood*, the computational model is assumed to be of the form $f(x; \vartheta)$, where d is a set of *M* given data points. The output of the model represents the values of the function at the given points for which Korali can build a likelihood function $p(d|\vartheta)$, and a prior probability density $p(\vartheta)$.

Currently, Korali uses a Normal estimator for the error component of the likelihood calculation, using a statistical-type variable, *sigma*:

$$p(d|\vartheta) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}((x-\mu)/\sigma)^2}$$

With a *Custom Likelihood*, the function $p(d|\vartheta)$ is given directly by a user-defined model of the form $f : \mathbb{R}^N \rightarrow \mathbb{R}$, where N is the number of variables.

Usage

```
e["Problem"]["Type"] = "Bayesian/Custom"
```

Compatible Solvers

This problem can be solved using the following modules:

- *Sampler*
- *Optimizer*

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Prior Distribution

- **Usage**: e["Variables"][*index*]["Prior Distribution"] = *string*
- **Description**: Indicates the name of the distribution to use as prior distribution.

Distribution Index

- **Usage**: e["Variables"][*index*]["Distribution Index"] = *unsigned integer*
- **Description**: Stores the the index number of the selected prior distribution.

Name

- **Usage**: e["Variables"][*index*]["Name"] = *string*
- **Description**: Defines the name of the variable.

Configuration

These are settings required by this module.

Likelihood Model

- **Usage**: e["Problem"]["Likelihood Model"] = *Computational Model*
- **Description**: Stores the user-defined likelihood model. It should return the value of the Log Likelihood of the given sample.

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Distribution Index": 0
}
```

1.29 Solver

Solver modules are algorithms/method to obtain the solution to a particular *problem*. Solvers are selected in each experiment e with the following syntax:

```
e = korali.Experiment()
e["Solver"]["Type"] = "Optimizer/CMAES"
```

For more information, see *Korali Usage Basics*. **Sub-Categories:**

1.29.1 Samplers

Sub-Categories:

HMC (Hamiltonian Monte Carlo)

This is an implementation of the *Hamiltonian Monte Carlo* algorithm, as published in Hoffman and Gelman. This solver can also be configured to run the standard *HMC* method.

Usage

```
e["Solver"]["Type"] = "Sampler/HMC"
```

Results

These are the results produced by this solver:

Sample Database

- **Usage**: e[“Results”][“Sample Database”] = List of Lists of *real number*
- **Description**: Collection of samples describing the probability distribution.

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment’s variables when this module is selected.

Initial Mean

- **Usage**: e[“Variables”][*index*][“Initial Mean”] = *real number*
- **Description**: Specifies the Initial Mean of the proposal distribution.

Initial Standard Deviation

- **Usage**: e[“Variables”][*index*][“Initial Standard Deviation”] = *real number*
- **Description**: Specifies the Standard Deviation for each variable. The proposal distribution is defined through a covariance matrix with the variance of the variables in its diagonal.

Lower Bound

- **Usage**: e[“Variables”][*index*][“Lower Bound”] = *real number*
- **Description**: Specifies the lower bound for the variable’s value.

Upper Bound

- **Usage**: e[“Variables”][*index*][“Upper Bound”] = *real number*
- **Description**: Specifies the upper bound for the variable’s value.

Initial Value

- **Usage**: e[“Variables”][*index*][“Initial Value”] = *real number*
- **Description**: Defines the initial value at or around which the algorithm shall start looking for an optimum.

Configuration

These are settings required by this module.

Burn In

- **Usage**: e[“Solver”][“Burn In”] = *unsigned integer*
- **Description**: Specifies the number of preliminary HMC steps before samples are being drawn. This may reduce effects from improper initialization.

Use Diagonal Metric

- **Usage**: e["Solver"]["Use Diagonal Metric"] = *True/False*
- **Description**: Specifies if Metric is restricted to be diagonal.

Num Integration Steps

- **Usage**: e["Solver"]["Num Integration Steps"] = *unsigned integer*
- **Description**: Number of Integration steps used in Leapfrog scheme. Only relevant if Adaptive Step Size not used.

Max Integration Steps

- **Usage**: e["Solver"]["Max Integration Steps"] = *unsigned integer*
- **Description**: Number of Integration steps used in Leapfrog scheme. Only relevant if Adaptive Step Size is used.

Use NUTS

- **Usage**: e["Solver"]["Use NUTS"] = *True/False*
- **Description**: Specifies if No-U-Turn Sampler (NUTS) is used.

Step Size

- **Usage**: e["Solver"]["Step Size"] = *real number*
- **Description**: Step size used in Leapfrog scheme.

Use Adaptive Step Size

- **Usage**: e["Solver"]["Use Adaptive Step Size"] = *True/False*
- **Description**: Controls whether dual averaging technique for adaptive step size calibration is used.

Target Acceptance Rate

- **Usage**: e["Solver"]["Target Acceptance Rate"] = *real number*
- **Description**: Desired Acceptance Rate for Adaptive Step Size calibration.

Acceptance Rate Learning Rate

- **Usage**: e["Solver"]["Acceptance Rate Learning Rate"] = *real number*
- **Description**: Learning rate of running acceptance rate estimate.

Target Integration Time

- **Usage**: e["Solver"]["Target Integration Time"] = *real number*
- **Description**: Targeted Integration Time for Leapfrog scheme. Only relevant if Adaptive Step Size used.

Adaptive Step Size Speed Constant

- **Usage**: e["Solver"]["Adaptive Step Size Speed Constant"] = *real number*
- **Description**: Controls how fast the step size is adapted. Only relevant if Adaptive Step Size used.

Adaptive Step Size Stabilization Constant

- **Usage**: e["Solver"]["Adaptive Step Size Stabilization Constant"] = *real number*
- **Description**: Controls stability of adaptive step size calibration during the inital iterations. Only relevant if Adaptive Step Size used.

Adaptive Step Size Schedule Constant

- **Usage**: e["Solver"]["Adaptive Step Size Schedule Constant"] = *real number*

- **Description**: Controls the weight of the previous step sizes. Only relevant if Adaptive Step Size used. The smaller the higher the weight.

Max Depth

- **Usage**: e["Solver"]["Max Depth"] = *unsigned integer*
- **Description**: Sets the maximum depth of NUTS binary tree.

Version

- **Usage**: e["Solver"]["Version"] = *string*
- **Description**: Metric can be set to 'Static', 'Euclidean' or 'Riemannian'.

Inverse Regularization Parameter

- **Usage**: e["Solver"]["Inverse Regularization Parameter"] = *real number*
- **Description**: Controls hardness of inverse metric approximation: For large values the Inverse Metric is closer the to Hessian (and therefore closer to degeneracy in certain cases).

Max Fixed Point Iterations

- **Usage**: e["Solver"]["Max Fixed Point Iterations"] = *unsigned integer*
- **Description**: Max number of fixed point iterations during implicit leapfrog scheme.

Step Size Jitter

- **Usage**: e["Solver"]["Step Size Jitter"] = *real number*
- **Description**: Step Size Jitter to vary trajectory length. Number must be in the interval [0.0. 1.0]. A uniform realization between [-(Step Size Jitter) * (Step Size), (Step Size Jitter) * (Step Size)) is sampled and added to the current Step Size.

Initial Fast Adaption Interval

- **Usage**: e["Solver"]["Initial Fast Adaption Interval"] = *unsigned integer*
- **Description**: Initial warm-up interval during which step size is adaptively adjusted.

Final Fast Adaption Interval

- **Usage**: e["Solver"]["Final Fast Adaption Interval"] = *unsigned integer*
- **Description**: Final warm-up interval during which step size is adaptively adjusted.

Initial Slow Adaption Interval

- **Usage**: e["Solver"]["Initial Slow Adaption Interval"] = *unsigned integer*
- **Description**: Lenght of first (out of 5) warm-up intervals during which euclidean metric is adapted. The length of each following slow adaption intervals is doubled.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Samples

- **Usage**: e["Solver"]["Max Samples"] = *unsigned integer*
- **Description**: Number of Samples to Generate.

- **Criteria**: `_sampleDatabase.size() >= _maxSamples`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Acceptance Count NUTS": 0.0,
"Acceptance Rate Learning Rate": 0.85,
"Adaptive Step Size Schedule Constant": 0.75,
"Adaptive Step Size Speed Constant": 0.05,
"Adaptive Step Size Stabilization Constant": 10.0,
"Burn In": 300,
"Final Fast Adaption Interval": 50,
"Initial Fast Adaption Interval": 75,
"Initial Slow Adaption Interval": 25,
"Inverse Regularization Parameter": 1.0,
"Max Depth": 5,
"Max Fixed Point Iterations": 8,
"Max Integration Steps": 100,
"Model Evaluation Count": 0,
"Multivariate Generator": {
    "Type": "Multivariate/Normal"
    },
"Normal Generator": {
    "Mean": 0.0,
    "Standard Deviation": 1.0,
    "Type": "Univariate/Normal"
    },
"Num Integration Steps": 4,
"Step Size": 0.1,
"Step Size Jitter": 0.0,
"Target Acceptance Rate": 0.65,
"Target Integration Time": 1.0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000,
    "Max Samples": 500
    },
"Uniform Generator": {
    "Maximum": 1.0,
```

(continues on next page)

(continued from previous page)

```
    "Minimum": 0.0,
    "Type": "Univariate/Uniform"
    },
"Use Adaptive Step Size": true,
"Use Diagonal Metric": true,
"Use NUTS": true,
"Variable Count": 0,
"Version": "Euclidean"
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Value": -Infinity,
"Lower Bound": -Infinity,
"Upper Bound": Infinity
}
```

Nested Sampling

This is an implementation of the *Nested Sampling* by John Skilling, as published in *https://projecteuclid.org/euclid.ba/1340370944*.

The implementation of the *Multi Ellipse* proposal distribution is based on the work of Feroz et. al. *https://academic.oup.com/mnras/article/398/4/1601/981502*.

Our version of the Multi Nest algorithm include a pior repartitioning strategy *https://link.springer.com/article/10.1007/s11222-018-9841-3* to efficiently sample unrepresentative priors.

Usage

```
e["Solver"]["Type"] = "Sampler/Nested"
```

Results

These are the results produced by this solver:

Posterior Sample Database

- **Usage**: e[“Results”][“Posterior Sample Database”] = List of Lists of *real number*
- **Description**: Samples that approximate the posterior distribution.

Posterior Sample LogPrior Database

- **Usage**: e[“Results”][“Posterior Sample LogPrior Database”] = List of *real number*
- **Description**: Log Priors of Samples in Posterior Samples Database.

Posterior Sample LogLikelihood Database

- **Usage**: e["Results"]["Posterior Sample LogLikelihood Database"] = List of *real number*
- **Description**: Log Likelihood of Samples in Posterior Samples Database.

Sample Database

- **Usage**: e["Results"]["Sample Database"] = List of Lists of *real number*
- **Description**: Collection of samples describing the probability distribution.

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: Specifies the lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: Specifies the upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: Defines the initial value at or around which the algorithm shall start looking for an optimum.

Configuration

These are settings required by this module.

Number Live Points

- **Usage**: e["Solver"]["Number Live Points"] = *unsigned integer*
- **Description**: Number of live samples.

Batch Size

- **Usage**: e["Solver"]["Batch Size"] = *unsigned integer*
- **Description**: Number of samples to discard and replace per generation, maximal number of parallel sample evaluation.

Add Live Points

- **Usage**: e["Solver"]["Add Live Points"] = *True/False*
- **Description**: Add live points to dead points.

Resampling Method

- **Usage**: e["Solver"]["Resampling Method"] = *string*
- **Description**: Method to generate new candidates (can be set to either 'Box' or 'Ellipse', 'Multi Ellipse').

Proposal Update Frequency

- **Usage**: e["Solver"]["Proposal Update Frequency"] = *unsigned integer*

- **Description**: Frequency of resampling distribution update (e.g. ellipse rescaling for Ellipse).

Ellipsoidal Scaling

- **Usage**: e["Solver"]["Ellipsoidal Scaling"] = *real number*
- **Description**: Scaling factor of ellipsoidal (only relevant for 'Ellipse' and 'Multi Ellipse' proposal).

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Min Log Evidence Delta

- **Usage**: e["Solver"]["Min Log Evidence Delta"] = *real number*
- **Description**: Minimal difference between estimated remaining log evidence and current logevidence.
- **Criteria**: `(_k->_currentGeneration > 1) && (_logEvidenceDifference <= _minLogEvidenceDelta)`

Max Effective Sample Size

- **Usage**: e["Solver"]["Max Effective Sample Size"] = *unsigned integer*
- **Description**: Estimated maximal evidence gain smaller than accumulated evidence by given factor.
- **Criteria**: `_maxEffectiveSampleSize <= _effectiveSampleSize`

Max Log Likelihood

- **Usage**: e["Solver"]["Max Log Likelihood"] = *unsigned integer*
- **Description**: Terminates if loglikelihood of sample removed from live set exceeds given value.
- **Criteria**: `_maxLogLikelihood <= _lStar`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Add Live Points": true,
"Batch Size": 1,
"Ellipsoidal Scaling": 1.0,
"Model Evaluation Count": 0,
"Multivariate Generator": {
    "Type": "Multivariate/Normal"
    },
"Normal Generator": {
    "Mean": 0.0,
    "Standard Deviation": 1.0,
    "Type": "Univariate/Normal"
    },
"Number Live Points": 1500,
"Proposal Update Frequency": 1500,
"Resampling Method": "Ellipse",
"Termination Criteria": {
    "Max Effective Sample Size": 10000000.0,
    "Max Generations": 10000000000,
    "Max Log Likelihood": 10000000.0,
    "Max Model Evaluations": 1000000000,
    "Min Log Evidence Delta": 0.01
    },
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Type": "Univariate/Uniform"
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Value": -Infinity,
"Lower Bound": -Infinity,
"Upper Bound": Infinity
}
```

MCMC (Delayed Rejection Adaptive Metropolis Algorithm)

This is an implementation of the *Delayed Rejection Adaptive Metropolis* algorithm, as published in Haario2006. This solver can also be configured to run the standard *Metropolis Hastings* method.

Usage

```
e["Solver"]["Type"] = "Sampler/MCMC"
```

Results

These are the results produced by this solver:

Sample Database

- **Usage**: e["Results"]["Sample Database"] = List of Lists of *real number*
- **Description**: Collection of samples describing the probability distribution.

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Initial Mean

- **Usage**: e["Variables"][*index*]["Initial Mean"] = *real number*
- **Description**: Specifies the Initial Mean of the proposal distribution.

Initial Standard Deviation

- **Usage**: e["Variables"][*index*]["Initial Standard Deviation"] = *real number*
- **Description**: Specifies the Standard Deviation for each variable. The proposal distribution is defined through a covariance matrix with the variance of the variables in its diagonal.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: Specifies the lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: Specifies the upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: Defines the initial value at or around which the algorithm shall start looking for an optimum.

Configuration

These are settings required by this module.

Burn In

- **Usage**: e["Solver"]["Burn In"] = *unsigned integer*
- **Description**: Specifies the number of preliminary MCMC steps before samples are being drawn. This may reduce effects from improper initialization.

Leap

- **Usage**: e["Solver"]["Leap"] = *unsigned integer*
- **Description**: Generates a Markov Chain containing samples from every 'Leap'-th step. This will increase the overall Chain Length by a factor of 'Leap'.

Rejection Levels

- **Usage**: e["Solver"]["Rejection Levels"] = *unsigned integer*
- **Description**: Controls the number of accept-reject stages per MCMC step (by default, this value is set 1, for values greater 1 the delajed rejection algorithm is active.

Use Adaptive Sampling

- **Usage**: e["Solver"]["Use Adaptive Sampling"] = *True/False*
- **Description**: Specifies if covariance matrix of the proposal distribution is calculated from the samples.

Non Adaption Period

- **Usage**: e["Solver"]["Non Adaption Period"] = *unsigned integer*
- **Description**: Number of steps (after Burn In steps) during which the initial covariance is used instead of the Chain Covariance. If 0 (default) is specified, this value is calibrated as $5%$ of the Max Chain Length (only relevant for Adaptive Sampling).

Chain Covariance Scaling

- **Usage**: e["Solver"]["Chain Covariance Scaling"] = *real number*
- **Description**: Learning rate of the Chain Covariance (only relevant for Adaptive Sampling).

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Samples

- **Usage**: e["Solver"]["Max Samples"] = *unsigned integer*
- **Description**: Number of Samples to Generate.
- **Criteria**: `_sampleDatabase.size() >= _maxSamples`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Burn In": 0,
"Chain Covariance Scaling": 1.0,
"Leap": 1,
"Model Evaluation Count": 0,
"Non Adaption Period": 0,
"Normal Generator": {
    "Mean": 0.0,
    "Standard Deviation": 1.0,
    "Type": "Univariate/Normal"
    },
"Rejection Levels": 1,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000,
    "Max Samples": 5000
    },
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Type": "Univariate/Uniform"
    },
"Use Adaptive Sampling": false,
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Value": -Infinity,
"Lower Bound": -Infinity,
"Upper Bound": Infinity
}
```

TMCMC (Transitional Markov Chain Monte Carlo)

This is the implementation of the *Transitional Markov Chain Monte Carlo* algorithm, as published in Ching2007.

TMCMC avoids sampling from difficult target probability densities (e.g. posterior distributions in a Bayesian inference problem) but samples from a series of intermediate PDFs that converge to the target PDF. This technique is also known as Sampling Importance Resampling in the Bayesian community.

Usage

```
e["Solver"]["Type"] = "Sampler/TMCMC"
```

Results

These are the results produced by this solver:

Posterior Sample Datase

- **Usage**: e["Results"]["Posterior Sample Datase"] = List of Lists of *real number*
- **Description**: Samples that approximate the posterior distribution.

Posterior Sample LogPrior Database

- **Usage**: e["Results"]["Posterior Sample LogPrior Database"] = List of *real number*
- **Description**: Log Priors of Samples in Posterior Samples Database.

Posterior Sample LogLikelihood Database

- **Usage**: e["Results"]["Posterior Sample LogLikelihood Database"] = List of *real number*
- **Description**: Log Likelihood of Samples in Posterior Samples Database.

Sample Database

- **Usage**: e["Results"]["Sample Database"] = List of Lists of *real number*
- **Description**: Collection of samples describing the probability distribution.

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: Specifies the lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: Specifies the upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: Defines the initial value at or around which the algorithm shall start looking for an optimum.

Configuration

These are settings required by this module.

Version

- **Usage**: e["Solver"]["Version"] = *string*
- **Description**: Indicates which variant of the TMCMC algorithm to use.
- **Options**:
  - "*TMCMC*": Uses the TMCMC algorithm.
  - "*mTMCMC*": Uses the mTMCMC algoritm.

Population Size

- **Usage**: e["Solver"]["Population Size"] = *unsigned integer*
- **Description**: Specifies the number of samples drawn from the posterior distribution at each generation.

Max Chain Length

- **Usage**: e["Solver"]["Max Chain Length"] = *unsigned integer*
- **Description**: Chains longer than Max Chain Length will be broken and samples will be duplicated (replacing samples associated with a chain length of 0). Max Chain Length of 1 corresponds to the BASIS algorithm [Wu2018].

Burn In

- **Usage**: e["Solver"]["Burn In"] = *unsigned integer*
- **Description**: Specifies the number of additional TMCMC steps per chain per generation (except for generation 0 and 1).

Per Generation Burn In

- **Usage**: e["Solver"]["Per Generation Burn In"] = List of *unsigned integer*
- **Description**: Specifies the number of additional TMCMC steps per chain at specified generations (this property will overwrite Default Burn In at specified generations). The first entry of the vector corresponds to the 2nd TMCMC generation.

Target Coefficient Of Variation

- **Usage**: e["Solver"]["Target Coefficient Of Variation"] = *real number*
- **Description**: Target coefficient of variation of the plausibility weights to update the annealing exponent ρ (by default, this value is 1.0 as suggested in [Ching2007]).

Covariance Scaling

- **Usage**: e["Solver"]["Covariance Scaling"] = *real number*
- **Description**: Scaling factor β^2 of Covariance Matrix (by default, this value is 0.04 as suggested in [Ching2007]).

Min Annealing Exponent Update

- **Usage**: e["Solver"]["Min Annealing Exponent Update"] = *real number*
- **Description**: Minimum increment of the exponent ρ. This parameter prevents TMCMC from stalling.

Max Annealing Exponent Update

- **Usage**: e["Solver"]["Max Annealing Exponent Update"] = *real number*

- **Description**: Maximum increment of the exponent ρ (by default, this value is 1.0 (inactive)).

Step Size

- **Usage**: e["Solver"]["Step Size"] = *real number*
- **Description**: Scaling factor of gradient and proposal distribution (only relevant for mTMCMC).

Domain Extension Factor

- **Usage**: e["Solver"]["Domain Extension Factor"] = *real number*
- **Description**: Defines boundaries for eigenvalue adjustments of proposal distribution (only relevant for mTMCMC).

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Target Annealing Exponent

- **Usage**: e["Solver"]["Target Annealing Exponent"] = *real number*
- **Description**: Determines the annealing exponent ρ to achieve before termination. TMCMC converges if ρ equals 1.0.
- **Criteria**: `_previousAnnealingExponent >= _targetAnnealingExponent`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Burn In": 0,
"Covariance Scaling": 0.04,
"Domain Extension Factor": 0.2,
"Max Annealing Exponent Update": 1.0,
"Max Chain Length": 1,
"Min Annealing Exponent Update": 1e-05,
"Model Evaluation Count": 0,
"Multinomial Generator": {
```

(continues on next page)

(continued from previous page)

```
    "Type": "Specific/Multinomial"
    },
"Multivariate Generator": {
    "Type": "Multivariate/Normal"
    },
"Per Generation Burn In": [],
"Step Size": 0.1,
"Target Coefficient Of Variation": 1.0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000,
    "Target Annealing Exponent": 1.0
    },
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Type": "Univariate/Uniform"
    },
"Variable Count": 0,
"Version": "TMCMC"
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Value": -Infinity,
"Lower Bound": -Infinity,
"Upper Bound": Infinity
}
```

1.29.2 Executor

This solver acts as executor.

Usage

```
e["Solver"]["Type"] = "Executor"
```

Results

These are the results produced by this solver:

Configuration

These are settings required by this module.

Executions Per Generation

- **Usage**: e["Solver"]["Executions Per Generation"] = *unsigned integer*
- **Description**: Specifies the number of model executions per generation. By default this setting is 0, meaning that all executions will be performed in the first generation. For values greater 0, executions will be split into batches and split int generations for intermediate output.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Executions Per Generation": 500000000,
"Model Evaluation Count": 0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000
    },
"Variable Count": 0
}
```

1.29.3 Designer

This solver determines the optimal experimental design for a given measurement model. In a first step, we sample the prior distribution of the parameters $\vartheta^{(i)} \sim p(\vartheta)$ for $i = 1, \ldots, N_\vartheta$. Given the samples, we evaluate the model $F(\vartheta^{(i)}, s)$ and save the results for the candidate locations of the design parameters $F(\vartheta^{(i)}, s)$. In the last step, the likelihood $p(y|\vartheta, s)$ is sampled for each of the parameters. Using the created samples $y^{(i,j)} \sim p(y|\vartheta^{(i)}, s)$ for $i = 1, \ldots, N_y$ we approximate the utility using numerical integration

$$\hat{U}(s) = \sum_{i=1}^{N_\vartheta} \sum_{j=1}^{N_y} w_{ij} \left[\ln p(y^{(i,j)}|\vartheta^{(i)}, s) - \ln \left(\sum_{k=1}^{N_\vartheta} w_k p(y^{(i,j)}|\vartheta^{(k)}, s) \right) \right]$$

and perform the optimization over the space of design space is performed in order to determine the optimal design.

Usage

```
e["Solver"]["Type"] = "Designer"
```

Results

These are the results produced by this solver:

Utility

- **Usage**: e["Results"]["Utility"] = List of *real number*
- **Description**: Evaluation of utility.

Optimal Design

- **Usage**: e["Results"]["Optimal Design"] = *real number*
- **Description**: Coordinates of the optimal design.

Configuration

These are settings required by this module.

Executions Per Generation

- **Usage**: e["Solver"]["Executions Per Generation"] = *unsigned integer*
- **Description**: Specifies the number of model executions per generation. By default this setting is 0, meaning that all executions will be performed in the first generation. For values greater 0, executions will be split into batches and split int generations for intermediate output.

Sigma

- **Usage**: e["Solver"]["Sigma"] = *real number*
- **Description**: Standard deviation for measurement.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Executions Per Generation": 0,
"Model Evaluation Count": 0,
"Normal Generator": {
    "Mean": 0.0,
    "Standard Deviation": 1.0,
    "Type": "Univariate/Normal"
    },
"Sigma": 0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000
    },
"Variable Count": 0
}
```

1.29.4 Optimizers

These modules solve problem of type *optimization*.

Sub-Categories:

Adam

This implements *Adam* for stochastic optimisation as published in https://arxiv.org/pdf/1412.6980.pdf.

Usage

```
e["Solver"]["Type"] = "Optimizer/Adam"
```

Results

These are the results produced by this solver:

Best Gradient(x)

- **Usage**: e["Results"]["Best Gradient(x)"] = List of *real number*
- **Description**: Values of dF(x) for the x parameters that produced the best F(x) found so far.

Best F(x)

- **Usage**: e["Results"]["Best F(x)"] = *real number*
- **Description**: Optimal value of F(x) found so far.

Best Parameters

- **Usage**: e["Results"]["Best Parameters"] = List of *real number*
- **Description**: Value for the x parameters that produced the best F(x).

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: [Hint] Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: [Hint] Upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: [Hint] Initial value at or around which the algorithm shall start looking for an optimum.

Initial Mean

- **Usage**: e["Variables"][*index*]["Initial Mean"] = *real number*
- **Description**: [Hint] Initial mean for the proposal distribution. This value must be defined between the variable's Mininum and Maximum settings (by default, this value is given by the center of the variable domain).

Initial Standard Deviation

- **Usage**: e["Variables"][*index*]["Initial Standard Deviation"] = *real number*
- **Description**: [Hint] Initial standard deviation of the proposal distribution for a variable (by default, this value is given by 30% of the variable domain width).

Minimum Standard Deviation Update

- **Usage**: e["Variables"][*index*]["Minimum Standard Deviation Update"] = *real number*
- **Description**: [Hint] Lower bound for the standard deviation updates of the proposal distribution for a variable. Korali increases the scaling factor sigma if this value is undershot.

Values

- **Usage**: e["Variables"][*index*]["Values"] = List of *real number*
- **Description**: [Hint] Locations to evaluate the Objective Function.

Configuration

These are settings required by this module.

Beta1

- **Usage**: e["Solver"]["Beta1"] = *real number*
- **Description**: Smoothing factor for momentum update.

Beta2

- **Usage**: e["Solver"]["Beta2"] = *real number*
- **Description**: Smoothing factor for gradient update.

Eta

- **Usage**: e["Solver"]["Eta"] = *real number*
- **Description**: Learning Rate (Step Size)

Epsilon

- **Usage**: e["Solver"]["Epsilon"] = *real number*
- **Description**: Term to facilitate numerical stability.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Min Gradient Norm

- **Usage**: e["Solver"]["Min Gradient Norm"] = *real number*
- **Description**: Specifies the minimal norm for the gradient of function with respect to Parameters.
- **Criteria**: `(_k->_currentGeneration > 1) && (_gradientNorm <= _minGradientNorm)`

Max Gradient Norm

- **Usage**: e["Solver"]["Max Gradient Norm"] = *real number*

- **Description**: Specifies the minimal norm for the gradient of function with respect to Parameters.
- **Criteria**: `(_k->_currentGeneration > 1) && (_gradientNorm >= _maxGradientNorm)`

Max Value

- **Usage**: e["Solver"]["Max Value"] = *real number*
- **Description**: Specifies the maximum target fitness to stop maximization.
- **Criteria**: `_k->_currentGeneration > 1 && (+_bestEverValue > _maxValue)`

Min Value Difference Threshold

- **Usage**: e["Solver"]["Min Value Difference Threshold"] = *real number*
- **Description**: Specifies the minimum fitness differential between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (fabs(_currentBestValue - _previousBestValue) < _minValueDifferenceThreshold)`

Max Infeasible Resamplings

- **Usage**: e["Solver"]["Max Infeasible Resamplings"] = *unsigned integer*
- **Description**: Maximum number of resamplings per candidate per generation if sample is outside of Lower and Upper Bound.
- **Criteria**: `(_maxInfeasibleResamplings > 0) && (_infeasibleSampleCount >= _maxInfeasibleResamplings)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Beta1": 0.9,
"Beta2": 0.999,
"Epsilon": 1e-08,
"Eta": 0.001,
"Model Evaluation Count": 0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Gradient Norm": 1000000000000.0,
```

(continues on next page)

(continued from previous page)

```
    "Max Infeasible Resamplings": 1000000,
    "Max Model Evaluations": 1000000000,
    "Max Value": Infinity,
    "Min Gradient Norm": 1e-12,
    "Min Value Difference Threshold": -Infinity
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Mean": NaN,
"Initial Standard Deviation": NaN,
"Initial Value": NaN,
"Lower Bound": -Infinity,
"Minimum Standard Deviation Update": 0.0,
"Upper Bound": Infinity,
"Values": []
}
```

MADGRAD

This implements **MADGRAD** (A **M** omentumized, **Ad** aptive, Dual Averaged **Grad** ient Method for Stochastic Optimization) for stochastic optimisation as published in https://arxiv.org/abs/2101.11075.

Usage

```
e["Solver"]["Type"] = "Optimizer/MADGRAD"
```

Results

These are the results produced by this solver:

Best Gradient(x)

- **Usage**: e["Results"]["Best Gradient(x)"] = List of *real number*
- **Description**: Values of dF(x) for the x parameters that produced the best F(x) found so far.

Best F(x)

- **Usage**: e["Results"]["Best F(x)"] = *real number*
- **Description**: Optimal value of F(x) found so far.

Best Parameters

- **Usage**: e["Results"]["Best Parameters"] = List of *real number*
- **Description**: Value for the x parameters that produced the best F(x).

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: [Hint] Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: [Hint] Upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: [Hint] Initial value at or around which the algorithm shall start looking for an optimum.

Initial Mean

- **Usage**: e["Variables"][*index*]["Initial Mean"] = *real number*
- **Description**: [Hint] Initial mean for the proposal distribution. This value must be defined between the variable's Mininum and Maximum settings (by default, this value is given by the center of the variable domain).

Initial Standard Deviation

- **Usage**: e["Variables"][*index*]["Initial Standard Deviation"] = *real number*
- **Description**: [Hint] Initial standard deviation of the proposal distribution for a variable (by default, this value is given by 30% of the variable domain width).

Minimum Standard Deviation Update

- **Usage**: e["Variables"][*index*]["Minimum Standard Deviation Update"] = *real number*
- **Description**: [Hint] Lower bound for the standard deviation updates of the proposal distribution for a variable. Korali increases the scaling factor sigma if this value is undershot.

Values

- **Usage**: e["Variables"][*index*]["Values"] = List of *real number*
- **Description**: [Hint] Locations to evaluate the Objective Function.

Configuration

These are settings required by this module.

Eta

- **Usage**: e["Solver"]["Eta"] = *real number*
- **Description**: Learning Rate (Step Size)

Weight Decay

- **Usage**: e["Solver"]["Weight Decay"] = *real number*
- **Description**: Smoothing factor for variable update.

Epsilon

- **Usage**: e["Solver"]["Epsilon"] = *real number*
- **Description**: Term to facilitate numerical stability

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Min Gradient Norm

- **Usage**: e["Solver"]["Min Gradient Norm"] = *real number*
- **Description**: Specifies the minimal norm for the gradient of function with respect to Parameters.
- **Criteria**: `(_k->_currentGeneration > 1) && (_gradientNorm <= _minGradientNorm)`

Max Gradient Norm

- **Usage**: e["Solver"]["Max Gradient Norm"] = *real number*
- **Description**: Specifies the minimal norm for the gradient of function with respect to Parameters.
- **Criteria**: `(_k->_currentGeneration > 1) && (_gradientNorm >= _maxGradientNorm)`

Max Value

- **Usage**: e["Solver"]["Max Value"] = *real number*
- **Description**: Specifies the maximum target fitness to stop maximization.
- **Criteria**: `_k->_currentGeneration > 1 && (+_bestEverValue > _maxValue)`

Min Value Difference Threshold

- **Usage**: e["Solver"]["Min Value Difference Threshold"] = *real number*
- **Description**: Specifies the minimum fitness differential between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (fabs(_currentBestValue - _previousBestValue) < _minValueDifferenceThreshold)`

Max Infeasible Resamplings

- **Usage**: e["Solver"]["Max Infeasible Resamplings"] = *unsigned integer*
- **Description**: Maximum number of resamplings per candidate per generation if sample is outside of Lower and Upper Bound.
- **Criteria**: `(_maxInfeasibleResamplings > 0) && (_infeasibleSampleCount >= _maxInfeasibleResamplings)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Epsilon": 1e-06,
"Eta": 0.01,
"Model Evaluation Count": 0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Gradient Norm": 1000000000000.0,
    "Max Infeasible Resamplings": 1000000,
    "Max Model Evaluations": 1000000000,
    "Max Value": Infinity,
    "Min Gradient Norm": 1e-12,
    "Min Value Difference Threshold": -Infinity
    },
"Variable Count": 0,
"Weight Decay": 0.9
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Mean": NaN,
"Initial Standard Deviation": NaN,
"Initial Value": NaN,
"Lower Bound": -Infinity,
"Minimum Standard Deviation Update": 0.0,
"Upper Bound": Infinity,
"Values": []
}
```

CMAES (Covariance Matrix Adaptation Evolution Strategy)

This is the implementation of the *Covariance Matrix Adaptation Evolution Strategy*, as published in Hansen2006. In an evolution strategy, new candidate solutions are sampled according to a multivariate normal distribution in $\mathbb{R}^n$. Recombination amounts to selecting a new mean value for the distribution. Mutation amounts to adding a random vector, a perturbation with zero mean. Pairwise dependencies between the variables in the distribution are represented by a covariance matrix. The covariance matrix adaptation (CMA) is a method to update the covariance matrix of this distribution. CMAES works iteratively, evaluating a number λ of samples per generation, and improving the covariance matrix for the samples in the next generation.

This solver also implements the *Constrained Covariance Matrix Adaptation Evolution Strategy*, as published in Arampatzis2019 and a version that includes gradient information Chen2009.

CCMAES is an extension of CMAES for constrained optimization problems. It uses the principle of *viability boundaries* to find an initial mean vector for the proposal distribution that does not violate constraints, and secondly it uses a *constraint handling technique* to efficiently adapt the proposal distribution to the constraints.

Usage

```
e["Solver"]["Type"] = "Optimizer/CMAES"
```

Results

These are the results produced by this solver:

Best F(x)

- **Usage**: e[“Results”][“Best F(x)”] = *real number*
- **Description**: Optimal value of F(x) found so far.

Best Parameters

- **Usage**: e[“Results”][“Best Parameters”] = List of *real number*
- **Description**: Value for the x parameters that produced the best F(x).

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Granularity

- **Usage**: e[“Variables”][*index*][“Granularity”] = *real number*
- **Description**: Specifies the granularity of a discrete variable, a granularity of 1.0 means that the variable can only take values in (.., -1.0, 0.0, +1.0, +2.0, ..) where the levels are set symmetric around the initial mean (here 0.0).

Lower Bound

- **Usage**: e[“Variables”][*index*][“Lower Bound”] = *real number*
- **Description**: [Hint] Lower bound for the variable's value.

Upper Bound

- **Usage**: e[“Variables”][*index*][“Upper Bound”] = *real number*

- **Description**: [Hint] Upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: [Hint] Initial value at or around which the algorithm shall start looking for an optimum.

Initial Mean

- **Usage**: e["Variables"][*index*]["Initial Mean"] = *real number*
- **Description**: [Hint] Initial mean for the proposal distribution. This value must be defined between the variable's Mininum and Maximum settings (by default, this value is given by the center of the variable domain).

Initial Standard Deviation

- **Usage**: e["Variables"][*index*]["Initial Standard Deviation"] = *real number*
- **Description**: [Hint] Initial standard deviation of the proposal distribution for a variable (by default, this value is given by 30% of the variable domain width).

Minimum Standard Deviation Update

- **Usage**: e["Variables"][*index*]["Minimum Standard Deviation Update"] = *real number*
- **Description**: [Hint] Lower bound for the standard deviation updates of the proposal distribution for a variable. Korali increases the scaling factor sigma if this value is undershot.

Values

- **Usage**: e["Variables"][*index*]["Values"] = List of *real number*
- **Description**: [Hint] Locations to evaluate the Objective Function.

Configuration

These are settings required by this module.

Population Size

- **Usage**: e["Solver"]["Population Size"] = *unsigned integer*
- **Description**: Specifies the number of samples to evaluate per generation (preferably $4+3*log(N)$, where N is the number of variables).

Mu Value

- **Usage**: e["Solver"]["Mu Value"] = *unsigned integer*
- **Description**: Number of best samples (offspring samples) used to update the covariance matrix and the mean (by default it is half the Sample Count).

Mu Type

- **Usage**: e["Solver"]["Mu Type"] = *string*
- **Description**: Weights given to the Mu best values to update the covariance matrix and the mean.
- **Options**:
  - "*Linear*": Distributes Mu weights linearly decreasing.
  - "*Equal*": Distributes Mu weights equally.
  - "*Logarithmic*": Distributes Mu weights logarithmically decreasing.

  - "*Proportional*": Distributes Mu weights proportional to objective function evaluation.

Initial Sigma Cumulation Factor

- **Usage**: e["Solver"]["Initial Sigma Cumulation Factor"] = *real number*
- **Description**: Controls the learning rate of the conjugate evolution path (by default this variable is internally calibrated).

Initial Damp Factor

- **Usage**: e["Solver"]["Initial Damp Factor"] = *real number*
- **Description**: Controls the updates of the covariance matrix scaling factor (by default this variable is internally calibrated).

Use Gradient Information

- **Usage**: e["Solver"]["Use Gradient Information"] = *True/False*
- **Description**: Include gradient information for proposal distribution update.

Gradient Step Size

- **Usage**: e["Solver"]["Gradient Step Size"] = float
- **Description**: Scaling factor for gradient step, only relevant if gradient information used.

Is Sigma Bounded

- **Usage**: e["Solver"]["Is Sigma Bounded"] = *True/False*
- **Description**: Sets an upper bound for the covariance matrix scaling factor. The upper bound is given by the average of the initial standard deviation of the variables.

Initial Cumulative Covariance

- **Usage**: e["Solver"]["Initial Cumulative Covariance"] = *real number*
- **Description**: Controls the learning rate of the evolution path for the covariance update (must be in (0,1], by default this variable is internally calibrated).

Diagonal Covariance

- **Usage**: e["Solver"]["Diagonal Covariance"] = *True/False*
- **Description**: Covariance matrix updates will be optimized for diagonal matrices.

Mirrored Sampling

- **Usage**: e["Solver"]["Mirrored Sampling"] = *True/False*
- **Description**: Generate the negative counterpart of each random number during sampling.

Viability Population Size

- **Usage**: e["Solver"]["Viability Population Size"] = *unsigned integer*
- **Description**: Specifies the number of samples per generation during the viability regime, i.e. during the search for a parameter vector not violating the constraints.

Viability Mu Value

- **Usage**: e["Solver"]["Viability Mu Value"] = *unsigned integer*
- **Description**: Number of best samples used to update the covariance matrix and the mean during the viability regime (by default this variable is half the Viability Sample Count).

Max Covariance Matrix Corrections

- **Usage**: e["Solver"]["Max Covariance Matrix Corrections"] = *unsigned integer*
- **Description**: Max number of covairance matrix adaptions per generation during the constraint handling loop.

Target Success Rate

- **Usage**: e["Solver"]["Target Success Rate"] = *real number*
- **Description**: Controls the updates of the covariance matrix scaling factor during the viability regime.

Covariance Matrix Adaption Strength

- **Usage**: e["Solver"]["Covariance Matrix Adaption Strength"] = *real number*
- **Description**: Controls the covariane matrix adaption strength if samples violate constraints.

Normal Vector Learning Rate

- **Usage**: e["Solver"]["Normal Vector Learning Rate"] = *real number*
- **Description**: Learning rate of constraint normal vectors (must be in (0, 1], by default this variable is internally calibrated).

Global Success Learning Rate

- **Usage**: e["Solver"]["Global Success Learning Rate"] = *real number*
- **Description**: Learning rate of success probability of objective function improvements.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Condition Covariance Matrix

- **Usage**: e["Solver"]["Max Condition Covariance Matrix"] = *real number*
- **Description**: Specifies the maximum condition of the covariance matrix.
- **Criteria**: `_k->_currentGeneration > 1 && (_maximumCovarianceEigenvalue >= _maxConditionCovarianceMatrix * _minimumCovarianceEigenvalue)`

Min Standard Deviation

- **Usage**: e["Solver"]["Min Standard Deviation"] = *real number*
- **Description**: Specifies the minimal standard deviation for any variable in any proposed sample.
- **Criteria**: `_k->_currentGeneration > 1 && (_currentMinStandardDeviation <= _minStandardDeviation)`

Max Standard Deviation

- **Usage**: e["Solver"]["Max Standard Deviation"] = *real number*
- **Description**: Specifies the maximal standard deviation for any variable in any proposed sample.
- **Criteria**: `_k->_currentGeneration > 1 && (_currentMaxStandardDeviation >= _maxStandardDeviation)`

Max Value

- **Usage**: e["Solver"]["Max Value"] = *real number*

- **Description**: Specifies the maximum target fitness to stop maximization.
- **Criteria**: `_k->_currentGeneration > 1 && (+_bestEverValue > _maxValue)`

Min Value Difference Threshold

- **Usage**: e["Solver"]["Min Value Difference Threshold"] = *real number*
- **Description**: Specifies the minimum fitness differential between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (fabs(_currentBestValue - _previousBestValue) < _minValueDifferenceThreshold)`

Max Infeasible Resamplings

- **Usage**: e["Solver"]["Max Infeasible Resamplings"] = *unsigned integer*
- **Description**: Maximum number of resamplings per candidate per generation if sample is outside of Lower and Upper Bound.
- **Criteria**: `(_maxInfeasibleResamplings > 0) && (_infeasibleSampleCount >= _maxInfeasibleResamplings)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Best Ever Value": -Infinity,
"Covariance Matrix Adaption Strength": 0.1,
"Current Max Standard Deviation": -Infinity,
"Current Min Standard Deviation": Infinity,
"Diagonal Covariance": false,
"Global Success Learning Rate": 0.2,
"Gradient Step Size": 0.01,
"Initial Cumulative Covariance": -1.0,
"Initial Damp Factor": -1.0,
"Initial Sigma Cumulation Factor": -1.0,
"Is Sigma Bounded": false,
"Max Covariance Matrix Corrections": 1000000,
"Maximum Covariance Eigenvalue": -Infinity,
"Minimum Covariance Eigenvalue": Infinity,
"Mirrored Sampling": false,
```

(continues on next page)

(continued from previous page)

```
"Model Evaluation Count": 0,
"Mu Type": "Logarithmic",
"Mu Value": 0,
"Normal Generator": {
    "Mean": 0.0,
    "Standard Deviation": 1.0,
    "Type": "Univariate/Normal"
    },
"Normal Vector Learning Rate": -1.0,
"Population Size": 0,
"Target Success Rate": 0.1818,
"Termination Criteria": {
    "Max Condition Covariance Matrix": Infinity,
    "Max Generations": 10000000000,
    "Max Infeasible Resamplings": 1000000,
    "Max Model Evaluations": 1000000000,
    "Max Standard Deviation": Infinity,
    "Max Value": Infinity,
    "Min Standard Deviation": -Infinity,
    "Min Value Difference Threshold": -Infinity
    },
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Type": "Univariate/Uniform"
    },
"Use Gradient Information": false,
"Variable Count": 0,
"Viability Mu Value": 0,
"Viability Population Size": 2
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Granularity": 0.0,
"Initial Mean": NaN,
"Initial Standard Deviation": NaN,
"Initial Value": NaN,
"Lower Bound": -Infinity,
"Minimum Standard Deviation Update": 0.0,
"Upper Bound": Infinity,
"Values": []
}
```

gridSearch

This solver implements a uniform grid-search to find the optimum $x^\star$ for a given set of parameters $x = (x_1, \ldots, x_n)$. Therefore it computes the value for the given objective function

$$f(x_i) \quad \forall i = 1, \ldots, N$$

and returns the optimum over the so computed values

$$f(x^\star) = \max\{f(x_1), \ldots, f(x_N)\}$$

and the corresponding argument of the maximum $x^\star$.

Usage

```
e["Solver"]["Type"] = "Optimizer/GridSearch"
```

Results

These are the results produced by this solver:

Best F(x)

- **Usage**: e["Results"]["Best F(x)"] = *real number*
- **Description**: Optimal value of F(x) found so far.

Best Parameters

- **Usage**: e["Results"]["Best Parameters"] = List of *real number*
- **Description**: Value for the x parameters that produced the best F(x).

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: [Hint] Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: [Hint] Upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: [Hint] Initial value at or around which the algorithm shall start looking for an optimum.

Initial Mean

- **Usage**: e["Variables"][*index*]["Initial Mean"] = *real number*

- **Description**: [Hint] Initial mean for the proposal distribution. This value must be defined between the variable's Mininum and Maximum settings (by default, this value is given by the center of the variable domain).

Initial Standard Deviation

- **Usage**: e["Variables"][*index*]["Initial Standard Deviation"] = *real number*
- **Description**: [Hint] Initial standard deviation of the proposal distribution for a variable (by default, this value is given by 30% of the variable domain width).

Minimum Standard Deviation Update

- **Usage**: e["Variables"][*index*]["Minimum Standard Deviation Update"] = *real number*
- **Description**: [Hint] Lower bound for the standard deviation updates of the proposal distribution for a variable. Korali increases the scaling factor sigma if this value is undershot.

Values

- **Usage**: e["Variables"][*index*]["Values"] = List of *real number*
- **Description**: [Hint] Locations to evaluate the Objective Function.

Configuration

These are settings required by this module.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Value

- **Usage**: e["Solver"]["Max Value"] = *real number*
- **Description**: Specifies the maximum target fitness to stop maximization.
- **Criteria**: `_k->_currentGeneration > 1 && (+_bestEverValue > _maxValue)`

Min Value Difference Threshold

- **Usage**: e["Solver"]["Min Value Difference Threshold"] = *real number*
- **Description**: Specifies the minimum fitness differential between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (fabs(_currentBestValue - _previousBestValue) < _minValueDifferenceThreshold)`

Max Infeasible Resamplings

- **Usage**: e["Solver"]["Max Infeasible Resamplings"] = *unsigned integer*
- **Description**: Maximum number of resamplings per candidate per generation if sample is outside of Lower and Upper Bound.
- **Criteria**: `(_maxInfeasibleResamplings > 0) && (_infeasibleSampleCount >= _maxInfeasibleResamplings)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Model Evaluation Count": 0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Infeasible Resamplings": 1000000,
    "Max Model Evaluations": 1000000000,
    "Max Value": Infinity,
    "Min Value Difference Threshold": -Infinity
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Mean": NaN,
"Initial Standard Deviation": NaN,
"Initial Value": NaN,
"Lower Bound": -Infinity,
"Minimum Standard Deviation Update": 0.0,
"Upper Bound": Infinity,
"Values": []
}
```

DEA (Differential Evolution Algorithm)

This is an implementation of the *Differential Evolution Algorithm* algorithm, as published in Storn1997.

DEA optimizes a problem by updating a population of candidate solutions through mutation and recombination. The update rules are simple and the objective function must not be differentiable. Our implementation includes various adaption and updating strategies Brest2006.

Usage

```
e["Solver"]["Type"] = "Optimizer/DEA"
```

Results

These are the results produced by this solver:

Best F(x)

- **Usage**: e["Results"]["Best F(x)"] = *real number*
- **Description**: Optimal value of F(x) found so far.

Best Parameters

- **Usage**: e["Results"]["Best Parameters"] = List of *real number*
- **Description**: Value for the x parameters that produced the best F(x).

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: [Hint] Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: [Hint] Upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: [Hint] Initial value at or around which the algorithm shall start looking for an optimum.

Initial Mean

- **Usage**: e["Variables"][*index*]["Initial Mean"] = *real number*
- **Description**: [Hint] Initial mean for the proposal distribution. This value must be defined between the variable's Mininum and Maximum settings (by default, this value is given by the center of the variable domain).

Initial Standard Deviation

- **Usage**: e["Variables"][*index*]["Initial Standard Deviation"] = *real number*
- **Description**: [Hint] Initial standard deviation of the proposal distribution for a variable (by default, this value is given by 30% of the variable domain width).

Minimum Standard Deviation Update

- **Usage**: e["Variables"][*index*]["Minimum Standard Deviation Update"] = *real number*
- **Description**: [Hint] Lower bound for the standard deviation updates of the proposal distribution for a variable. Korali increases the scaling factor sigma if this value is undershot.

Values

- **Usage**: e["Variables"][*index*]["Values"] = List of *real number*
- **Description**: [Hint] Locations to evaluate the Objective Function.

Configuration

These are settings required by this module.

Population Size

- **Usage**: e["Solver"]["Population Size"] = *unsigned integer*
- **Description**: Specifies the number of samples to evaluate per generation (preferably 5-10x the number of variables).

Crossover Rate

- **Usage**: e["Solver"]["Crossover Rate"] = *real number*
- **Description**: Controls the rate at which dimensions of the samples are mixed (must be in [0,1]).

Mutation Rate

- **Usage**: e["Solver"]["Mutation Rate"] = *real number*
- **Description**: Controls the scaling of the vector differentials (must be in [0,2], preferably < 1).

Mutation Rule

- **Usage**: e["Solver"]["Mutation Rule"] = *string*
- **Description**: Controls the Mutation Rate.
- **Options**:
  - "*Fixed*": Fixed mutation rate.
  - "*Self Adaptive*": Varying Crossover Rate and Mutation Rate, according to [Brest2006].

Parent Selection Rule

- **Usage**: e["Solver"]["Parent Selection Rule"] = *string*
- **Description**: Defines the selection rule of the parent vector.
- **Options**:
  - "*Random*": Select parent randomly.
  - "*Best*": Mutate only best variables.

Accept Rule

- **Usage**: e["Solver"]["Accept Rule"] = *string*

- **Description**: Sets the accept rule after sample mutation and evaluation.
- **Options**:
  - "*Best*": Update best sample if better than Best Ever Sample.
  - "*Greedy*": Accept all candiates better than parent.
  - "*Iterative*": Iterate through candidates and accept if Best Ever Value improved.
  - "*Improved*": Accept all candidates better than Best Ever Sample.

Fix Infeasible

- **Usage**: e["Solver"]["Fix Infeasible"] = *True/False*
- **Description**: If set true, Korali samples a random sample between Parent and the voiolated boundary. If set false, infeasible samples are mutated again until feasible.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Min Value

- **Usage**: e["Solver"]["Min Value"] = *real number*
- **Description**: Specifies the target fitness to stop minimization.
- **Criteria**: `(_k->_currentGeneration > 1) && (-_bestEverValue < _minValue)`

Min Step Size

- **Usage**: e["Solver"]["Min Step Size"] = *real number*
- **Description**: Specifies the minimal step size of the population mean from one gneration to another.
- **Criteria**: `_currentMinimumStepSize < _minStepSize`

Max Value

- **Usage**: e["Solver"]["Max Value"] = *real number*
- **Description**: Specifies the maximum target fitness to stop maximization.
- **Criteria**: `_k->_currentGeneration > 1 && (+_bestEverValue > _maxValue)`

Min Value Difference Threshold

- **Usage**: e["Solver"]["Min Value Difference Threshold"] = *real number*
- **Description**: Specifies the minimum fitness differential between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (fabs(_currentBestValue - _previousBestValue) < _minValueDifferenceThreshold)`

Max Infeasible Resamplings

- **Usage**: e["Solver"]["Max Infeasible Resamplings"] = *unsigned integer*
- **Description**: Maximum number of resamplings per candidate per generation if sample is outside of Lower and Upper Bound.

- **Criteria**: `(_maxInfeasibleResamplings > 0) && (_infeasibleSampleCount >= _maxInfeasibleResamplings)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Accept Rule": "Greedy",
"Best Ever Value": -Infinity,
"Best Sample Index": 0,
"Candidate Population": [
    []
],
"Crossover Rate": 0.9,
"Current Best Variables": [],
"Current Mean": [],
"Current Minimum Step Size": 0.0,
"Fix Infeasible": true,
"Max Distances": [],
"Model Evaluation Count": 0,
"Mutation Rate": 0.5,
"Mutation Rule": "Fixed",
"Normal Generator": {
    "Mean": 0.0,
    "Standard Deviation": 1.0,
    "Type": "Univariate/Normal"
    },
"Parent Selection Rule": "Random",
"Population Size": 200,
"Previous Best Ever Value": -Infinity,
"Previous Mean": [],
"Previous Value Vector": [],
"Sample Population": [
    []
],
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Infeasible Resamplings": 1000000,
    "Max Model Evaluations": 1000000000,
    "Max Value": Infinity,
```

(continues on next page)

(continued from previous page)

```
    "Min Step Size": -Infinity,
    "Min Value": -Infinity,
    "Min Value Difference Threshold": -Infinity
    },
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Type": "Univariate/Uniform"
    },
"Value Vector": [],
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Mean": NaN,
"Initial Standard Deviation": NaN,
"Initial Value": NaN,
"Lower Bound": -Infinity,
"Minimum Standard Deviation Update": 0.0,
"Upper Bound": Infinity,
"Values": []
}
```

RProp (Resilient Back Propagation)

This is an implementation of the *Resilient Backpropagation* algorithm. See the wikipedia article for reference.

Usage

```
e["Solver"]["Type"] = "Optimizer/Rprop"
```

Results

These are the results produced by this solver:

Best Gradient(x)

- **Usage**: e["Results"]["Best Gradient(x)"] = List of *real number*
- **Description**: Values of dF(x) for the x parameters that produced the best F(x) found so far.

Best F(x)

- **Usage**: e["Results"]["Best F(x)"] = *real number*
- **Description**: Optimal value of F(x) found so far.

Best Parameters

- **Usage**: e["Results"]["Best Parameters"] = List of *real number*
- **Description**: Value for the x parameters that produced the best F(x).

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: [Hint] Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: [Hint] Upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: [Hint] Initial value at or around which the algorithm shall start looking for an optimum.

Initial Mean

- **Usage**: e["Variables"][*index*]["Initial Mean"] = *real number*
- **Description**: [Hint] Initial mean for the proposal distribution. This value must be defined between the variable's Mininum and Maximum settings (by default, this value is given by the center of the variable domain).

Initial Standard Deviation

- **Usage**: e["Variables"][*index*]["Initial Standard Deviation"] = *real number*
- **Description**: [Hint] Initial standard deviation of the proposal distribution for a variable (by default, this value is given by 30% of the variable domain width).

Minimum Standard Deviation Update

- **Usage**: e["Variables"][*index*]["Minimum Standard Deviation Update"] = *real number*
- **Description**: [Hint] Lower bound for the standard deviation updates of the proposal distribution for a variable. Korali increases the scaling factor sigma if this value is undershot.

Values

- **Usage**: e["Variables"][*index*]["Values"] = List of *real number*
- **Description**: [Hint] Locations to evaluate the Objective Function.

Configuration

These are settings required by this module.

Delta0

- **Usage**: e["Solver"]["Delta0"] = *real number*
- **Description**: Initial Delta.

Delta Min

- **Usage**: e["Solver"]["Delta Min"] = *real number*
- **Description**: Minimum Delta, parameter for step size calibration.

Delta Max

- **Usage**: e["Solver"]["Delta Max"] = *real number*
- **Description**: Maximum Delta, parameter for step size calibration.

Eta Minus

- **Usage**: e["Solver"]["Eta Minus"] = *real number*
- **Description**: Parameter for step size calibration.

Eta Plus

- **Usage**: e["Solver"]["Eta Plus"] = *real number*
- **Description**: Parameter for step size calibration.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Gradient Norm

- **Usage**: e["Solver"]["Max Gradient Norm"] = *real number*
- **Description**: Maximum value of the norm of the gradient.
- **Criteria**: `_normPreviousGradient < _maxGradientNorm`

Max Stall Generations

- **Usage**: e["Solver"]["Max Stall Generations"] = *unsigned integer*
- **Description**: Maximum times stalled with function evaluation bigger than the best one.
- **Criteria**: :code:` _maxStallCounter >= _maxStallGenerations`

Parameter Relative Tolerance

- **Usage**: e["Solver"]["Parameter Relative Tolerance"] = *real number*
- **Description**: Relative tolerance in parameter difference between generations.
- **Criteria**: `_xDiff<_parameterRelativeTolerance && _xDiff>0`

Max Value

- **Usage**: e["Solver"]["Max Value"] = *real number*

- **Description**: Specifies the maximum target fitness to stop maximization.
- **Criteria**: `_k->_currentGeneration > 1 && (+_bestEverValue > _maxValue)`

Min Value Difference Threshold

- **Usage**: e["Solver"]["Min Value Difference Threshold"] = *real number*
- **Description**: Specifies the minimum fitness differential between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (fabs(_currentBestValue - _previousBestValue) < _minValueDifferenceThreshold)`

Max Infeasible Resamplings

- **Usage**: e["Solver"]["Max Infeasible Resamplings"] = *unsigned integer*
- **Description**: Maximum number of resamplings per candidate per generation if sample is outside of Lower and Upper Bound.
- **Criteria**: `(_maxInfeasibleResamplings > 0) && (_infeasibleSampleCount >= _maxInfeasibleResamplings)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Delta Max": 50,
"Delta Min": 1e-06,
"Delta0": 0.1,
"Eta Minus": 0.5,
"Eta Plus": 1.2,
"Model Evaluation Count": 0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Gradient Norm": 0.0,
    "Max Infeasible Resamplings": 1000000,
    "Max Model Evaluations": 1000000000,
    "Max Stall Generations": 20,
    "Max Value": Infinity,
    "Min Value Difference Threshold": -Infinity,
    "Parameter Relative Tolerance": 0.0001
```

(continues on next page)

(continued from previous page)

```
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Mean": NaN,
"Initial Standard Deviation": NaN,
"Initial Value": NaN,
"Lower Bound": -Infinity,
"Minimum Standard Deviation Update": 0.0,
"Upper Bound": Infinity,
"Values": []
}
```

MO-CMA-ES (Mutli-Objective Covariance Matrix Adaptation Evolution Strategy)

This is the implementation of the *Mutli-Objective Covariance Matrix Adaptation Evolution Strategy*, as published in Voss2010. The multi-objective covariance matrix adaptation evolution strategy (MO-CMA-ES) is an evolutionary algorithm for continuous vector-valued optimization. It combines indicator-based selection based on the contributing hypervolume with the efficient strategy parameter adaptation of the elitist covariance matrix adaptation evolution strategy (CMA-ES).

Usage

```
e["Solver"]["Type"] = "Optimizer/MOCMAES"
```

Results

These are the results produced by this solver:

Best F(x)

- **Usage**: e["Results"]["Best F(x)"] = *real number*
- **Description**: Optimal value of F(x) found so far.

Best Parameters

- **Usage**: e["Results"]["Best Parameters"] = List of *real number*
- **Description**: Value for the x parameters that produced the best F(x).

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: [Hint] Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: [Hint] Upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: [Hint] Initial value at or around which the algorithm shall start looking for an optimum.

Initial Mean

- **Usage**: e["Variables"][*index*]["Initial Mean"] = *real number*
- **Description**: [Hint] Initial mean for the proposal distribution. This value must be defined between the variable's Mininum and Maximum settings (by default, this value is given by the center of the variable domain).

Initial Standard Deviation

- **Usage**: e["Variables"][*index*]["Initial Standard Deviation"] = *real number*
- **Description**: [Hint] Initial standard deviation of the proposal distribution for a variable (by default, this value is given by 30% of the variable domain width).

Minimum Standard Deviation Update

- **Usage**: e["Variables"][*index*]["Minimum Standard Deviation Update"] = *real number*
- **Description**: [Hint] Lower bound for the standard deviation updates of the proposal distribution for a variable. Korali increases the scaling factor sigma if this value is undershot.

Values

- **Usage**: e["Variables"][*index*]["Values"] = List of *real number*
- **Description**: [Hint] Locations to evaluate the Objective Function.

Configuration

These are settings required by this module.

Population Size

- **Usage**: e["Solver"]["Population Size"] = *unsigned integer*
- **Description**: Specifies the number of samples to evaluate per generation (preferably $4+3*log(N)$, where N is the number of variables).

Mu Value

- **Usage**: e["Solver"]["Mu Value"] = *unsigned integer*

- **Description**: Number of best samples (offspring) advancing to the next generation (by default it is half the Sample Count).

Evolution Path Adaption Strength

- **Usage**: e["Solver"]["Evolution Path Adaption Strength"] = *real number*
- **Description**: Controls the learning rate of the conjugate evolution path (must be in (0,1], by default this variable is internally calibrated, variable Cc in reference).

Covariance Learning Rate

- **Usage**: e["Solver"]["Covariance Learning Rate"] = *real number*
- **Description**: Controls the learning rate of the covariance matrices (must be in (0,1], by default this variable is internally calibrated, variable Ccov in reference).

Target Success Rate

- **Usage**: e["Solver"]["Target Success Rate"] = *real number*
- **Description**: Value that controls the updates of the covariance matrix and the evolution path (must be in (0,1], variable Psucc in reference).

Threshold Probability

- **Usage**: e["Solver"]["Threshold Probability"] = *real number*
- **Description**: Threshold that defines update scheme for the covariance matrix and the evolution path (must be in (0,1], variable Pthresh in reference).

Success Learning Rate

- **Usage**: e["Solver"]["Success Learning Rate"] = *real number*
- **Description**: Learning Rate of success rates (must be in (0,1], by default this variable is internally calibrated, variable Cp in reference).

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Min Max Value Difference Threshold

- **Usage**: e["Solver"]["Min Max Value Difference Threshold"] = *real number*
- **Description**: Specifies the min max fitness differential between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (std::abs(*std::max_element(_currentBestValueD begin(), _currentBestValueDifferences.end())) < _minValueDifferenceThreshold)`

Min Variable Difference Threshold

- **Usage**: e["Solver"]["Min Variable Difference Threshold"] = *real number*
- **Description**: Specifies the min L2 norm of the best samples between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (*std::max_element(_currentBestVariableDiffere begin(), _currentBestVariableDifferences.end()) < _minVariableDifferenceThreshold)`

Min Standard Deviation

- **Usage**: e["Solver"]["Min Standard Deviation"] = *real number*
- **Description**: Specifies the minimal standard deviation.
- **Criteria**: `_k->_currentGeneration > 1 && (*std::max_element(_currentMinStandardDeviatio begin(), _currentMinStandardDeviations.end()) <= _minStandardDeviation)`

Max Standard Deviation

- **Usage**: e["Solver"]["Max Standard Deviation"] = *real number*
- **Description**: Specifies the maximal standard deviation.
- **Criteria**: `_k->_currentGeneration > 1 && (*std::min_element(_currentMaxStandardDeviatio begin(), _currentMaxStandardDeviations.end()) >= _maxStandardDeviation)`

Max Value

- **Usage**: e["Solver"]["Max Value"] = *real number*
- **Description**: Specifies the maximum target fitness to stop maximization.
- **Criteria**: `_k->_currentGeneration > 1 && (+_bestEverValue > _maxValue)`

Min Value Difference Threshold

- **Usage**: e["Solver"]["Min Value Difference Threshold"] = *real number*
- **Description**: Specifies the minimum fitness differential between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (fabs(_currentBestValue - _previousBestValue) < _minValueDifferenceThreshold)`

Max Infeasible Resamplings

- **Usage**: e["Solver"]["Max Infeasible Resamplings"] = *unsigned integer*
- **Description**: Maximum number of resamplings per candidate per generation if sample is outside of Lower and Upper Bound.
- **Criteria**: `(_maxInfeasibleResamplings > 0) && (_infeasibleSampleCount >= _maxInfeasibleResamplings)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Covariance Learning Rate": -1.0,
"Evolution Path Adaption Strength": -1.0,
"Model Evaluation Count": 0,
"Mu Value": 0,
"Multinormal Generator": {
    "Mean Vector": [
        0.0
    ],
    "Sigma": [
        1.0
    ],
    "Type": "Multivariate/Normal"
    },
"Population Size": 0,
"Success Learning Rate": 0.08,
"Target Success Rate": 0.175,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Infeasible Resamplings": 1000000,
    "Max Model Evaluations": 1000000000,
    "Max Standard Deviation": Infinity,
    "Max Value": Infinity,
    "Min Max Value Difference Threshold": -Infinity,
    "Min Standard Deviation": -Infinity,
    "Min Value Difference Threshold": -Infinity,
    "Min Variable Difference Threshold": -Infinity
    },
"Threshold Probability": 0.44,
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Type": "Univariate/Uniform"
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Mean": NaN,
"Initial Standard Deviation": NaN,
"Initial Value": NaN,
"Lower Bound": -Infinity,
"Minimum Standard Deviation Update": 0.0,
"Upper Bound": Infinity,
"Values": []
}
```

AdaBelief

This implements *AdaBelief* for gradient based optimisation as published in https://arxiv.org/abs/2010.07468.

Usage

```
e["Solver"]["Type"] = "Optimizer/AdaBelief"
```

Results

These are the results produced by this solver:

Best Gradient(x)

- **Usage**: e["Results"]["Best Gradient(x)"] = List of *real number*
- **Description**: Values of dF(x) for the x parameters that produced the best F(x) found so far.

Best F(x)

- **Usage**: e["Results"]["Best F(x)"] = *real number*
- **Description**: Optimal value of F(x) found so far.

Best Parameters

- **Usage**: e["Results"]["Best Parameters"] = List of *real number*
- **Description**: Value for the x parameters that produced the best F(x).

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: [Hint] Lower bound for the variable's value.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: [Hint] Upper bound for the variable's value.

Initial Value

- **Usage**: e["Variables"][*index*]["Initial Value"] = *real number*
- **Description**: [Hint] Initial value at or around which the algorithm shall start looking for an optimum.

Initial Mean

- **Usage**: e["Variables"][*index*]["Initial Mean"] = *real number*
- **Description**: [Hint] Initial mean for the proposal distribution. This value must be defined between the variable's Mininum and Maximum settings (by default, this value is given by the center of the variable domain).

Initial Standard Deviation

- **Usage**: e["Variables"][*index*]["Initial Standard Deviation"] = *real number*
- **Description**: [Hint] Initial standard deviation of the proposal distribution for a variable (by default, this value is given by 30% of the variable domain width).

Minimum Standard Deviation Update

- **Usage**: e["Variables"][*index*]["Minimum Standard Deviation Update"] = *real number*
- **Description**: [Hint] Lower bound for the standard deviation updates of the proposal distribution for a variable. Korali increases the scaling factor sigma if this value is undershot.

Values

- **Usage**: e["Variables"][*index*]["Values"] = List of *real number*
- **Description**: [Hint] Locations to evaluate the Objective Function.

Configuration

These are settings required by this module.

Beta1

- **Usage**: e["Solver"]["Beta1"] = *real number*
- **Description**: Smoothing factor for momentum update.

Beta2

- **Usage**: e["Solver"]["Beta2"] = *real number*
- **Description**: Smoothing for gradient update.

Eta

- **Usage**: e["Solver"]["Eta"] = *real number*
- **Description**: Learning Rate (Step Size)

Epsilon

- **Usage**: e["Solver"]["Epsilon"] = *real number*
- **Description**: Term to facilitate numerical stability.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Min Gradient Norm

- **Usage**: e["Solver"]["Min Gradient Norm"] = *real number*
- **Description**: Specifies the minimal norm for the gradient of function with respect to Parameters.
- **Criteria**: `(_k->_currentGeneration > 1) && (_gradientNorm <= _minGradientNorm)`

Max Gradient Norm

- **Usage**: e["Solver"]["Max Gradient Norm"] = *real number*

- **Description**: Specifies the minimal norm for the gradient of function with respect to Parameters.
- **Criteria**: `(_k->_currentGeneration > 1) && (_gradientNorm >= _maxGradientNorm)`

Max Value

- **Usage**: e["Solver"]["Max Value"] = *real number*
- **Description**: Specifies the maximum target fitness to stop maximization.
- **Criteria**: `_k->_currentGeneration > 1 && (+_bestEverValue > _maxValue)`

Min Value Difference Threshold

- **Usage**: e["Solver"]["Min Value Difference Threshold"] = *real number*
- **Description**: Specifies the minimum fitness differential between two consecutive generations before stopping execution.
- **Criteria**: `_k->_currentGeneration > 1 && (fabs(_currentBestValue - _previousBestValue) < _minValueDifferenceThreshold)`

Max Infeasible Resamplings

- **Usage**: e["Solver"]["Max Infeasible Resamplings"] = *unsigned integer*
- **Description**: Maximum number of resamplings per candidate per generation if sample is outside of Lower and Upper Bound.
- **Criteria**: `(_maxInfeasibleResamplings > 0) && (_infeasibleSampleCount >= _maxInfeasibleResamplings)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Beta1": 0.9,
"Beta2": 0.999,
"Epsilon": 1e-08,
"Eta": 0.001,
"Model Evaluation Count": 0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Gradient Norm": 1000000000000.0,
```

(continues on next page)

(continued from previous page)

```
    "Max Infeasible Resamplings": 1000000,
    "Max Model Evaluations": 1000000000,
    "Max Value": Infinity,
    "Min Gradient Norm": 1e-12,
    "Min Value Difference Threshold": -Infinity
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Mean": NaN,
"Initial Standard Deviation": NaN,
"Initial Value": NaN,
"Lower Bound": -Infinity,
"Minimum Standard Deviation Update": 0.0,
"Upper Bound": Infinity,
"Values": []
}
```

1.29.5 Agent

Agent in the Reinforcement Learning framework. The agent interacts with the environment by selecting actions given a state. The rule for this selection is based on a policy. The agents goal is to find the policy that maximizes the expected cumulative sum of rewards. We distinguish problems with discrete and continuous action spaces. **Sub-Categories:**

Discrete Agent

Specialization of the Agent module for handling problems with discrete action spaces.

Sub-Categories:

Discrete VRACER

This solver implements a discrete version of VRACER (https://arxiv.org/abs/1807.05827)

Usage

```
e["Solver"]["Type"] = "Agent/Discrete/DVRACER"
```

Results

These are the results produced by this solver:

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Configuration

These are settings required by this module.

Initial Inverse Temperature

- **Usage**: e["Solver"]["Initial Inverse Temperature"] = float
- **Description**: Initial inverse temperature of the softmax distribution. Large values lead to a distribution that is more concentrated around the action with highes Q-value estimate.

Mode

- **Usage**: e["Solver"]["Mode"] = *string*
- **Description**: Specifies the operation mode for the agent.
- **Options**:
  - "*Training*": Learns a policy for the reinforcement learning problem.
  - "*Testing*": Tests the policy with a learned policy.

Testing / Sample Ids

- **Usage**: e["Solver"]["Testing"]["Sample Ids"] = List of *unsigned integer*
- **Description**: A vector with the identifiers for the samples to test the hyperparameters with.

Testing / Current Policies

- **Usage**: e["Solver"]["Testing"]["Current Policies"] = knlohmann::json
- **Description**: The current hyperparameters of the policies to test.

Training / Average Depth

- **Usage**: e["Solver"]["Training"]["Average Depth"] = *unsigned integer*
- **Description**: Specifies the depth of the running training average to report.

Concurrent Workers

- **Usage**: e["Solver"]["Concurrent Workers"] = *unsigned integer*
- **Description**: Indicates the number of concurrent environments to use to collect experiences.

Episodes Per Generation

- **Usage**: e["Solver"]["Episodes Per Generation"] = *unsigned integer*
- **Description**: Number of reinforcement learning episodes per Korali generation (checkpoints are generated between generations).

Mini Batch / Size

- **Usage**: e["Solver"]["Mini Batch"]["Size"] = *unsigned integer*
- **Description**: The number of experiences to randomly select to train the neural network(s) with.

Time Sequence Length

- **Usage**: e["Solver"]["Time Sequence Length"] = *unsigned integer*
- **Description**: Indicates the number of contiguous experiences to pass to the NN for learning. This is only useful when using recurrent NNs.

Learning Rate

- **Usage**: e["Solver"]["Learning Rate"] = float
- **Description**: The initial learning rate to use for the NN hyperparameter optimization.

L2 Regularization / Enabled

- **Usage**: e["Solver"]["L2 Regularization"]["Enabled"] = *True/False*
- **Description**: Boolean to determine if l2 regularization will be applied to the neural networks.

L2 Regularization / Importance

- **Usage**: e["Solver"]["L2 Regularization"]["Importance"] = float
- **Description**: Coefficient for l2 regularization.

Neural Network / Hidden Layers

- **Usage**: e["Solver"]["Neural Network"]["Hidden Layers"] = knlohmann::json
- **Description**: Indicates the configuration of the hidden neural network layers.

Neural Network / Optimizer

- **Usage**: e["Solver"]["Neural Network"]["Optimizer"] = *string*
- **Description**: Indicates the optimizer algorithm to update the NN hyperparameters.

Neural Network / Engine

- **Usage**: e["Solver"]["Neural Network"]["Engine"] = *string*
- **Description**: Specifies which Neural Network backend to use.

Discount Factor

- **Usage**: e["Solver"]["Discount Factor"] = float
- **Description**: Represents the discount factor to weight future experiences.

Importance Weight Truncation Level

- **Usage**: e["Solver"]["Importance Weight Truncation Level"] = float
- **Description**: Represents the discount factor to weight future experiences.

Experience Replay / Serialize

- **Usage**: e["Solver"]["Experience Replay"]["Serialize"] = *True/False*

- **Description**: Indicates whether to serialize and store the experience replay after each generation. Disabling will reduce I/O overheads but will disable the checkpoint/resume function.

Experience Replay / Start Size

- **Usage**: e["Solver"]["Experience Replay"]["Start Size"] = *unsigned integer*
- **Description**: The minimum number of experiences before learning starts.

Experience Replay / Maximum Size

- **Usage**: e["Solver"]["Experience Replay"]["Maximum Size"] = *unsigned integer*
- **Description**: The size of the replay memory. If this number is exceeded, experiences are deleted.

Experience Replay / Off Policy / Cutoff Scale

- **Usage**: e["Solver"]["Experience Replay"]["Off Policy"]["Cutoff Scale"] = float
- **Description**: Initial Cut-Off to classify experiences as on- or off-policy. (c_max in https://arxiv.org/abs/1807.05827)

Experience Replay / Off Policy / Target

- **Usage**: e["Solver"]["Experience Replay"]["Off Policy"]["Target"] = float
- **Description**: Target fraction of off-policy experiences in the replay memory. (D in https://arxiv.org/abs/1807.05827)

Experience Replay / Off Policy / Annealing Rate

- **Usage**: e["Solver"]["Experience Replay"]["Off Policy"]["Annealing Rate"] = float
- **Description**: Annealing rate for Off Policy Cutoff Scale and Learning Rate. (A in https://arxiv.org/abs/1807.05827)

Experience Replay / Off Policy / REFER Beta

- **Usage**: e["Solver"]["Experience Replay"]["Off Policy"]["REFER Beta"] = float
- **Description**: Initial value for the penalisation coefficient for off-policiness. (beta in https://arxiv.org/abs/1807.05827)

Experiences Between Policy Updates

- **Usage**: e["Solver"]["Experiences Between Policy Updates"] = float
- **Description**: The number of experiences to receive before training/updating (real number, may be less than < 1.0, for more than one update per experience).

State Rescaling / Enabled

- **Usage**: e["Solver"]["State Rescaling"]["Enabled"] = *True/False*
- **Description**: Determines whether to normalize the states, such that they have mean 0 and standard deviation 1 (done only once after the initial exploration phase).

Reward / Rescaling / Enabled

- **Usage**: e["Solver"]["Reward"]["Rescaling"]["Enabled"] = *True/False*
- **Description**: Determines whether to normalize the rewards, such that they have mean 0 and standard deviation 1

Multi Agent Relationship

- **Usage**: e["Solver"]["Multi Agent Relationship"] = *string*
- **Description**: Specifies whether we are in an individual setting or collaborator setting.

- **Options**:
  - "*Individual*": Each Agent learns solely on his own.
  - "*Cooperation*": Rewards from each agent are averaged.
  - "*Competition*": Agent learns solely on his own and Experience Sharing is disabled.

Multi Agent Correlation

- **Usage**: e["Solver"]["Multi Agent Correlation"] = *True/False*
- **Description**: Specifies whether we take into account the dependencies of the agents or not.

Multi Agent Sampling

- **Usage**: e["Solver"]["Multi Agent Sampling"] = *string*
- **Description**: Specifies how to sample the minibatch.
- **Options**:
  - "*Tuple*": Sample expId and use same value for all agents.
  - "*Baseline*": Sample expId and agentId.
  - "*Experience*": Sample expId separately for all agents.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Episodes

- **Usage**: e["Solver"]["Max Episodes"] = *unsigned integer*
- **Description**: The solver will stop when the given number of episodes have been run.
- **Criteria**: `(_mode == "Training") && (_maxEpisodes > 0) && (_currentEpisode >= _maxEpisodes)`

Max Experiences

- **Usage**: e["Solver"]["Max Experiences"] = *unsigned integer*
- **Description**: The solver will stop when the given number of experiences have been gathered.
- **Criteria**: `(_mode == "Training") && (_maxExperiences > 0) && (_experienceCount >= _maxExperiences)`

Max Policy Updates

- **Usage**: e["Solver"]["Max Policy Updates"] = *unsigned integer*
- **Description**: The solver will stop when the given number of optimization steps have been performed.
- **Criteria**: `(_mode == "Training") && (_maxPolicyUpdates > 0) && (_policyUpdateCount >= _maxPolicyUpdates)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.

- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Concurrent Workers": 1,
"Discount Factor": 0.995,
"Episodes Per Generation": 1,
"Experience Replay": {
    "Off Policy": {
        "Annealing Rate": 0.0,
        "Cutoff Scale": 4.0,
        "REFER Beta": 0.3,
        "Target": 0.1
        },
    "Serialize": true
    },
"Importance Weight Truncation Level": 1.0,
"Initial Inverse Temperature": 1.0,
"L2 Regularization": {
    "Enabled": false,
    "Importance": 0.0001
    },
"Mini Batch": {
    "Size": 256
    },
"Model Evaluation Count": 0,
"Multi Agent Correlation": false,
"Multi Agent Relationship": "Individual",
"Multi Agent Sampling": "Tuple",
"Reward": {
    "Rescaling": {
        "Enabled": false
        }
    },
"State Rescaling": {
    "Enabled": false
    },
"Termination Criteria": {
    "Max Episodes": 0,
    "Max Experiences": 0,
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000,
    "Max Policy Updates": 0
    },
"Testing": {
```

(continues on next page)

(continued from previous page)

```
    "Best Policies": {    },
    "Current Policies": {    },
    "Sample Ids": []
    },
"Time Sequence Length": 1,
"Training": {
    "Average Depth": 100,
    "Best Policies": {    },
    "Current Policies": {    }
    },
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Name": "Agent / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{    }
```

Continuous Agent

Specialization of the Agent module for handling problems with continuous action spaces.

Sub-Categories:

VRACER

This solver implements VRACER as described in https://arxiv.org/abs/1807.05827

Usage

```
e["Solver"]["Type"] = "Agent/Continuous/VRACER"
```

Results

These are the results produced by this solver:

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Initial Exploration Noise

- **Usage**: e["Variables"][*index*]["Initial Exploration Noise"] = float
- **Description**: Initial standard deviation of the Gaussian distribution from which the given action is sampled.

Configuration

These are settings required by this module.

Policy / Distribution

- **Usage**: e["Solver"]["Policy"]["Distribution"] = *string*
- **Description**: Specifies which probability distribution to use for the policy.
- **Options**:
  - "*Normal*": Use a normal distribution for the policy.
  - "*Squashed Normal*": Use a normal distribution for the policy. The sampled action is passed through an arctan to bound the action domain.
  - "*Clipped Normal*": Use a normal distribution for the policy. Actions exceeding an upper or lower bound will be truncated.
  - "*Truncated Normal*": Use a truncated normal distribution for the policy.
  - "*Beta*": Use the beta distribution for for the policy.

Mode

- **Usage**: e["Solver"]["Mode"] = *string*
- **Description**: Specifies the operation mode for the agent.
- **Options**:
  - "*Training*": Learns a policy for the reinforcement learning problem.
  - "*Testing*": Tests the policy with a learned policy.

Testing / Sample Ids

- **Usage**: e["Solver"]["Testing"]["Sample Ids"] = List of *unsigned integer*
- **Description**: A vector with the identifiers for the samples to test the hyperparameters with.

Testing / Current Policies

- **Usage**: e["Solver"]["Testing"]["Current Policies"] = knlohmann::json
- **Description**: The current hyperparameters of the policies to test.

Training / Average Depth

- **Usage**: e["Solver"]["Training"]["Average Depth"] = *unsigned integer*
- **Description**: Specifies the depth of the running training average to report.

Concurrent Workers

- **Usage**: e["Solver"]["Concurrent Workers"] = *unsigned integer*
- **Description**: Indicates the number of concurrent environments to use to collect experiences.

Episodes Per Generation

- **Usage**: e["Solver"]["Episodes Per Generation"] = *unsigned integer*
- **Description**: Number of reinforcement learning episodes per Korali generation (checkpoints are generated between generations).

Mini Batch / Size

- **Usage**: e["Solver"]["Mini Batch"]["Size"] = *unsigned integer*
- **Description**: The number of experiences to randomly select to train the neural network(s) with.

Time Sequence Length

- **Usage**: e["Solver"]["Time Sequence Length"] = *unsigned integer*
- **Description**: Indicates the number of contiguous experiences to pass to the NN for learning. This is only useful when using recurrent NNs.

Learning Rate

- **Usage**: e["Solver"]["Learning Rate"] = float
- **Description**: The initial learning rate to use for the NN hyperparameter optimization.

L2 Regularization / Enabled

- **Usage**: e["Solver"]["L2 Regularization"]["Enabled"] = *True/False*
- **Description**: Boolean to determine if l2 regularization will be applied to the neural networks.

L2 Regularization / Importance

- **Usage**: e["Solver"]["L2 Regularization"]["Importance"] = float
- **Description**: Coefficient for l2 regularization.

Neural Network / Hidden Layers

- **Usage**: e["Solver"]["Neural Network"]["Hidden Layers"] = knlohmann::json
- **Description**: Indicates the configuration of the hidden neural network layers.

Neural Network / Optimizer

- **Usage**: e["Solver"]["Neural Network"]["Optimizer"] = *string*
- **Description**: Indicates the optimizer algorithm to update the NN hyperparameters.

Neural Network / Engine

- **Usage**: e["Solver"]["Neural Network"]["Engine"] = *string*
- **Description**: Specifies which Neural Network backend to use.

Discount Factor

- **Usage**: e["Solver"]["Discount Factor"] = float
- **Description**: Represents the discount factor to weight future experiences.

Importance Weight Truncation Level

- **Usage**: e["Solver"]["Importance Weight Truncation Level"] = float
- **Description**: Represents the discount factor to weight future experiences.

Experience Replay / Serialize

- **Usage**: e["Solver"]["Experience Replay"]["Serialize"] = *True/False*
- **Description**: Indicates whether to serialize and store the experience replay after each generation. Disabling will reduce I/O overheads but will disable the checkpoint/resume function.

Experience Replay / Start Size

- **Usage**: e["Solver"]["Experience Replay"]["Start Size"] = *unsigned integer*
- **Description**: The minimum number of experiences before learning starts.

Experience Replay / Maximum Size

- **Usage**: e["Solver"]["Experience Replay"]["Maximum Size"] = *unsigned integer*
- **Description**: The size of the replay memory. If this number is exceeded, experiences are deleted.

Experience Replay / Off Policy / Cutoff Scale

- **Usage**: e["Solver"]["Experience Replay"]["Off Policy"]["Cutoff Scale"] = float
- **Description**: Initial Cut-Off to classify experiences as on- or off-policy. (c_max in https://arxiv.org/abs/1807.05827)

Experience Replay / Off Policy / Target

- **Usage**: e["Solver"]["Experience Replay"]["Off Policy"]["Target"] = float
- **Description**: Target fraction of off-policy experiences in the replay memory. (D in https://arxiv.org/abs/1807.05827)

Experience Replay / Off Policy / Annealing Rate

- **Usage**: e["Solver"]["Experience Replay"]["Off Policy"]["Annealing Rate"] = float
- **Description**: Annealing rate for Off Policy Cutoff Scale and Learning Rate. (A in https://arxiv.org/abs/1807.05827)

Experience Replay / Off Policy / REFER Beta

- **Usage**: e["Solver"]["Experience Replay"]["Off Policy"]["REFER Beta"] = float
- **Description**: Initial value for the penalisation coefficient for off-policiness. (beta in https://arxiv.org/abs/1807.05827)

Experiences Between Policy Updates

- **Usage**: e["Solver"]["Experiences Between Policy Updates"] = float
- **Description**: The number of experiences to receive before training/updating (real number, may be less than < 1.0, for more than one update per experience).

State Rescaling / Enabled

- **Usage**: e["Solver"]["State Rescaling"]["Enabled"] = *True/False*
- **Description**: Determines whether to normalize the states, such that they have mean 0 and standard deviation 1 (done only once after the initial exploration phase).

Reward / Rescaling / Enabled

- **Usage**: e["Solver"]["Reward"]["Rescaling"]["Enabled"] = *True/False*
- **Description**: Determines whether to normalize the rewards, such that they have mean 0 and standard deviation 1

Multi Agent Relationship

- **Usage**: e["Solver"]["Multi Agent Relationship"] = *string*
- **Description**: Specifies whether we are in an individual setting or collaborator setting.
- **Options**:
  - "*Individual*": Each Agent learns solely on his own.
  - "*Cooperation*": Rewards from each agent are averaged.
  - "*Competition*": Agent learns solely on his own and Experience Sharing is disabled.

Multi Agent Correlation

- **Usage**: e["Solver"]["Multi Agent Correlation"] = *True/False*
- **Description**: Specifies whether we take into account the dependencies of the agents or not.

Multi Agent Sampling

- **Usage**: e["Solver"]["Multi Agent Sampling"] = *string*
- **Description**: Specifies how to sample the minibatch.
- **Options**:
  - "*Tuple*": Sample expId and use same value for all agents.
  - "*Baseline*": Sample expId and agentId.
  - "*Experience*": Sample expId separately for all agents.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Episodes

- **Usage**: e["Solver"]["Max Episodes"] = *unsigned integer*
- **Description**: The solver will stop when the given number of episodes have been run.
- **Criteria**: `(_mode == "Training") && (_maxEpisodes > 0) && (_currentEpisode >= _maxEpisodes)`

Max Experiences

- **Usage**: e["Solver"]["Max Experiences"] = *unsigned integer*
- **Description**: The solver will stop when the given number of experiences have been gathered.
- **Criteria**: `(_mode == "Training") && (_maxExperiences > 0) && (_experienceCount >= _maxExperiences)`

Max Policy Updates

- **Usage**: e["Solver"]["Max Policy Updates"] = *unsigned integer*

- **Description**: The solver will stop when the given number of optimization steps have been performed.
- **Criteria**: `(_mode == "Training") && (_maxPolicyUpdates > 0) && (_policyUpdateCount >= _maxPolicyUpdates)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Concurrent Workers": 1,
"Discount Factor": 0.995,
"Episodes Per Generation": 1,
"Experience Replay": {
    "Off Policy": {
        "Annealing Rate": 0.0,
        "Cutoff Scale": 4.0,
        "REFER Beta": 0.3,
        "Target": 0.1
        },
    "Serialize": true
    },
"Importance Weight Truncation Level": 1.0,
"L2 Regularization": {
    "Enabled": false,
    "Importance": 0.0001
    },
"Mini Batch": {
    "Size": 256
    },
"Model Evaluation Count": 0,
"Multi Agent Correlation": false,
"Multi Agent Relationship": "Individual",
"Multi Agent Sampling": "Tuple",
"Normal Generator": {
    "Mean": 0.0,
    "Name": "Agent / Continuous / Normal Generator",
    "Standard Deviation": 1.0,
    "Type": "Univariate/Normal"
    },
"Policy": {
    "Distribution": "Normal"
```

(continues on next page)

(continued from previous page)

```
    },
"Reward": {
    "Rescaling": {
        "Enabled": false
        }
    },
"State Rescaling": {
    "Enabled": false
    },
"Termination Criteria": {
    "Max Episodes": 0,
    "Max Experiences": 0,
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000,
    "Max Policy Updates": 0
    },
"Testing": {
    "Best Policies": {    },
    "Current Policies": {    },
    "Sample Ids": []
    },
"Time Sequence Length": 1,
"Training": {
    "Average Depth": 100,
    "Best Policies": {    },
    "Current Policies": {    }
    },
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Name": "Agent / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Initial Exploration Noise": -1.0
}
```

1.29.6 Stochastic Simulation Methods (SSM)

Stochastic simulation methods solve problem of type *optimization*. A reaction consists of reactants (or reservoirs), products and an associated reaction rate. Implementation based on this project.

Sub-Categories:

Tau Leaping Algorithm

This models implements the TauLeaping algorithm, as published in Cao2005.

The tau-leaping algorithm speeds up the SSA algorithm by approxiamting the number of firings for each reaction channel during a chosen time increment (tau) as a Poisson variable. This algorithm avoids negative reactants using a simple trick and it is generally more accurate thant other variants using a Poisson procedure.

Usage

```
e["Solver"]["Type"] = "SSM/TauLeaping"
```

Results

These are the results produced by this solver:

Mean Trajectory

- **Usage**: e[“Results”][“Mean Trajectory”] = List of Lists of *real number*
- **Description**: Binned mean of trajectory, averaged over all simulation runs.

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment’s variables when this module is selected.

Configuration

These are settings required by this module.

Nc

- **Usage**: e[“Solver”][“Nc”] = *integer*
- **Description**: TODO.

Epsilon

- **Usage**: e[“Solver”][“Epsilon”] = *real number*
- **Description**: Error control parameter, larger epsilon leads to larger tau steps and errors.

Acceptance Factor

- **Usage**: e[“Solver”][“Acceptance Factor”] = *real number*
- **Description**: Multiplicator of inverse total propensity, to calculate acceptance criterion of tau step.

Num SSA Steps

- **Usage**: e["Solver"]["Num SSA Steps"] = *integer*
- **Description**: Number of intermediate SSA steps if leap step rejected.

Simulation Length

- **Usage**: e["Solver"]["Simulation Length"] = *real number*
- **Description**: Total duration of a stochastic reaction simulation.

Diagnostics / Num Bins

- **Usage**: e["Solver"]["Diagnostics"]["Num Bins"] = *unsigned integer*
- **Description**: Number of bins to calculate the mean trajectory at termination.

Simulations Per Generation

- **Usage**: e["Solver"]["Simulations Per Generation"] = *unsigned integer*
- **Description**: Number of trajectory simulations per Korali generation (checkpoints are generated between generations).

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Num Simulations

- **Usage**: e["Solver"]["Max Num Simulations"] = *unsigned integer*
- **Description**: Max number of trajectory simulations.
- **Criteria**: `_maxNumSimulations <= _completedSimulations`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Diagnostics": {
    "Num Bins": 100
    },
"Model Evaluation Count": 0,
"Poisson Generator": {
    "Mean": 1.0,
    "Type": "Univariate/Poisson"
    },
"Simulations Per Generation": 1,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000,
    "Max Num Simulations": 1
    },
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Type": "Univariate/Uniform"
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{    }
```

SSA (Stochastic Simulation Algorithm)

The Gillespie algorithm (1977) simulates trajectories of a stochastic equation system (reactions) for which the rates are known. If the number of reactants is high, the time steps become small and the algorithm stalls. This behaviour is alleviated in the tau-leaping algorithm.

Usage

```
e["Solver"]["Type"] = "SSM/SSA"
```

Results

These are the results produced by this solver:

Mean Trajectory

- **Usage**: e["Results"]["Mean Trajectory"] = List of Lists of *real number*
- **Description**: Binned mean of trajectory, averaged over all simulation runs.

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Configuration

These are settings required by this module.

Simulation Length

- **Usage**: e["Solver"]["Simulation Length"] = *real number*
- **Description**: Total duration of a stochastic reaction simulation.

Diagnostics / Num Bins

- **Usage**: e["Solver"]["Diagnostics"]["Num Bins"] = *unsigned integer*
- **Description**: Number of bins to calculate the mean trajectory at termination.

Simulations Per Generation

- **Usage**: e["Solver"]["Simulations Per Generation"] = *unsigned integer*
- **Description**: Number of trajectory simulations per Korali generation (checkpoints are generated between generations).

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Num Simulations

- **Usage**: e["Solver"]["Max Num Simulations"] = *unsigned integer*
- **Description**: Max number of trajectory simulations.
- **Criteria**: `_maxNumSimulations <= _completedSimulations`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Diagnostics": {
    "Num Bins": 100
    },
"Model Evaluation Count": 0,
"Simulations Per Generation": 1,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000,
    "Max Num Simulations": 1
    },
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": 0.0,
    "Type": "Univariate/Uniform"
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{    }
```

1.29.7 Deep Supervisor

Uses a *Neural Network* to solve a *Supervised Learning* problem.

It employs three Neural Networks:

- *Training Network* - Used to adjust the weights and biases with the help of a user-defined optimizer to minimize the loss function given in the supervised learning problem (e.g., Mean Square Error).
- *Validation Network* - Used to measure the improvement (in terms of loss function values) of the training network on the validation data given in the supervised learning problem.
- *Test Network* - The result of the optimisation procedure which can be used to evaluate the neural network on a test set using the test() function.

Uses a combination of a training and evaluation *Neural Networks* to solve a *Supervised Learning* problem. At each generation, it applies one or more optimization steps based on the loss function and the input/solutions received. The input and solutions may change in between generations.

Inference is fully openMP parallelizable, so that different openMP threads can infer from the learned parameters simultaneously. The training part should be done sequentially.

Usage

```
e["Solver"]["Type"] = "DeepSupervisor"
```

Results

These are the results produced by this solver:

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Configuration

These are settings required by this module.

Mode

- **Usage**: e["Solver"]["Mode"] = *string*
- **Description**: Specifies the operation mode for the learner.
- **Options**:
  - "*Training*": Trains a neural network for the supervised learning problem.
  - "*Testing*": Forwards the neural network given an input batch.

Neural Network / Hidden Layers

- **Usage**: e["Solver"]["Neural Network"]["Hidden Layers"] = knlohmann::json
- **Description**: Sets the configuration of the hidden layers for the neural network.

Neural Network / Output Activation

- **Usage**: e["Solver"]["Neural Network"]["Output Activation"] = knlohmann::json
- **Description**: Allows setting an aditional activation for the output layer.

Neural Network / Output Layer

- **Usage**: e["Solver"]["Neural Network"]["Output Layer"] = knlohmann::json
- **Description**: Sets any additional configuration (e.g., masks) for the output NN layer.

Neural Network / Engine

- **Usage**: e["Solver"]["Neural Network"]["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.

Neural Network / Optimizer

- **Usage**: e["Solver"]["Neural Network"]["Optimizer"] = *string*
- **Description**: Determines which optimizer algorithm to use to apply the gradients on the neural network's hyperparameters.

- **Options**:
  - "*Adam*": Uses the Adam algorithm.
  - "*AdaBelief*": Uses the AdaBelief algorithm.
  - "*MadGrad*": Uses the MadGrad algorithm.
  - "*AdaGrad*": Uses the AdaGrad algorithm.

Loss Function

- **Usage**: e["Solver"]["Loss Function"] = *string*
- **Description**: Function to calculate the difference (loss) between the NN inference and the exact solution and its gradients for optimization.
- **Options**:
  - "*Direct Gradient*": The given solution represents the gradients of the loss with respect to the network-output. Note that Korali uses the gradients to maximize the objective.
  - "*Mean Squared Error*": The loss is calculated as the negative mean of square errors, one per input in the batch. Note that Korali maximizes the negative MSE.

Learning Rate

- **Usage**: e["Solver"]["Learning Rate"] = float
- **Description**: Learning rate for the underlying ADAM optimizer.

L2 Regularization / Enabled

- **Usage**: e["Solver"]["L2 Regularization"]["Enabled"] = *True/False*
- **Description**: Regulates if l2 regularization will be applied to the neural network.

L2 Regularization / Importance

- **Usage**: e["Solver"]["L2 Regularization"]["Importance"] = *True/False*
- **Description**: Importance weight of l2 regularization.

Output Weights Scaling

- **Usage**: e["Solver"]["Output Weights Scaling"] = float
- **Description**: Specified by how much will the weights of the last linear transformation of the NN be scaled. A value of < 1.0 is useful for a more deterministic start.

Batch Concurrency

- **Usage**: e["Solver"]["Batch Concurrency"] = *unsigned integer*
- **Description**: Specifies in how many parts will the mini batch be split for concurrent processing. It must divide the training mini batch size perfectly.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Target Loss

- **Usage**: e["Solver"]["Target Loss"] = float
- **Description**: Specifies the maximum number of suboptimal generations.
- **Criteria**: `(_k->_currentGeneration > 1) && (_targetLoss > 0.0) && (_currentLoss <= _targetLoss)`

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Batch Concurrency": 1,
"Hyperparameters": [],
"L2 Regularization": {
    "Enabled": false,
    "Importance": 0.0001
    },
"Model Evaluation Count": 0,
"Neural Network": {
    "Output Activation": "Identity",
    "Output Layer": {    }
    },
"Output Weights Scaling": 1.0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000,
    "Target Loss": -1.0
    },
"Variable Count": 0
}
```

Variable Defaults

These following configuration will be assigned to each of the experiment variables by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{     }
```

1.29.8 Integrator

This solver performs the numerical integration of scalar functions $f : \mathbb{R}^D \rightarrow \mathbb{R}$ for the "Integration" problem.

Usage

```
e["Solver"]["Type"] = "Integrator"
```

Results

These are the results produced by this solver:

Integral

- **Usage**: e["Results"]["Integral"] = *real number*
- **Description**: Value of the integral.

Variable-Specific Settings

These are settings required by this module that are added to each of the experiment's variables when this module is selected.

Lower Bound

- **Usage**: e["Variables"][*index*]["Lower Bound"] = *real number*
- **Description**: Lower bound for integration.

Upper Bound

- **Usage**: e["Variables"][*index*]["Upper Bound"] = *real number*
- **Description**: Upper bound for integration.

Configuration

These are settings required by this module.

Executions Per Generation

- **Usage**: e["Solver"]["Executions Per Generation"] = *unsigned integer*
- **Description**: Specifies the number of model executions per generation. By default this setting is 0, meaning that all executions will be performed in the first generation. For values greater 0, executions will be split into batches and split int generations for intermediate output.

Termination Criteria

These are the customizable criteria that indicates whether the solver should continue or finish execution. Korali will stop when at least one of these conditions are met. The criteria is expressed in C++ since it is compiled and evaluated as seen here in the engine.

Max Model Evaluations

- **Usage**: e["Solver"]["Max Model Evaluations"] = *unsigned integer*
- **Description**: Specifies the maximum allowed evaluations of the computational model.
- **Criteria**: `_maxModelEvaluations <= _modelEvaluationCount`

Max Generations

- **Usage**: e["Solver"]["Max Generations"] = *unsigned integer*
- **Description**: Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.
- **Criteria**: `_k->_currentGeneration > _maxGenerations`

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Executions Per Generation": 100,
"Model Evaluation Count": 0,
"Termination Criteria": {
    "Max Generations": 10000000000,
    "Max Model Evaluations": 1000000000
    },
"Variable Count": 0
}
```

1.30 Conduits

Conduit modules specify how the computational models used to evaluate samples are to be executed, in particular regarding parallelism/concurrency. The conduit to use is specified at the Korali Engine-level, and is shared among all the experiments in the run.

To select a specific conduit, use the following syntax:

```
k = korali.Engine()
k["Conduit"]["Type"] = "Distributed"
```

For more information, see *Parallel Execution*.

Sub-Categories:

1.30.1 Concurrent Conduit

This concurrent conduit uses fork/join mechanisms to distribute sample evaluation among *n* concurrent workers, each running as separate process from the main application process. Communication among workers is realized via OS pipes.

Use this model if your application cannot be parallelized with MPI or linked to Korali in any way.

For example, pre-packaged (black-box) applications can be run using this conduit and then instantiating a new process per sample evaluation (see: *Concurrent Execution Example*).

For more information, see *Parallel Execution*.

Usage

```
k["Conduit"]["Type"] = "Concurrent"
```

Configuration

These are settings required by this module.

Concurrent Jobs

- **Usage**: e[“Conduit”][“Concurrent Jobs”] = *unsigned integer*
- **Description**: Specifies the number of worker processes (jobs) running concurrently.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Concurrent Jobs": 1
}
```

1.30.2 Distributed Conduit

This distributed conduit uses MPI to distribute sample evaluation among *n* workers. Each worker consists of *k* MPI ranks, where *k* is a configurable parameter. Communication among workers is realized via MPI messages.

This model is ideal for when your computational model can be directly linked with Korali and/or expects an MPI communicator itself.

For an example on how to create a MPI/Python Korali application, see: *MPI/Python Example*). For an example on how to create a MPI/C++ Korali application, see: *MPI/C++ Example*). For more information, see *Parallel Execution*.

Usage

```
k["Conduit"]["Type"] = "Distributed"
```

Configuration

These are settings required by this module.

Ranks Per Worker

- **Usage**: e["Conduit"]["Ranks Per Worker"] = *integer*
- **Description**: Specifies the number of MPI ranks per Korali worker.

Engine Ranks

- **Usage**: e["Conduit"]["Engine Ranks"] = *integer*
- **Description**: Specifies the number of MPI ranks for the Korali engine.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Engine Ranks": 1,
"Ranks Per Worker": 1
}
```

1.30.3 Sequential Conduit

This sequential conduit is the default conduit when using Korali. It evaluates samples running computational models sequentially (i.e., not concurrently or in parallel), under the same process as the calling application.

For more information, see *Parallel Execution*.

Usage

```
k["Conduit"]["Type"] = "Sequential"
```

Configuration

These are settings required by this module.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{      }
```

1.31 Distributions

In the subdirectories you can find the distribtuion modules for the generation of random variables and the evaluation of their densities and derived values.

Sub-Categories:

1.31.1 Special Distributions

This directory contains a collection of distribution objects which do not conform the implementation interface of the other categories of distributions (uniform or multivariate).

Sub-Categories:

Multinomial Distribution

Multinomial

The Multinomial distribution has the probability density function (PDF):

$$f(x_1, .., x_k; p_1, .., p_k) = \frac{n!}{x_1!..x_k!} p_1^{x_1} \cdots p_k^{x_k}$$

where $\sum x_i = n$ is the **number of trials** and $p_i > 0$ is an **event probability**.

Usage

```
e["Distribution"][*index*]["Type"] = "Specific/Multinomial"
```

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

1.31.2 Univariate Distributions

This directory contains a collection of univariate continuous probability distributions.

Sub-Categories:

Weibull Distribution

Weibull

The Weibull distribution has the probability density function:

$$f(x;k,\theta) = \begin{cases} \frac{k}{\theta}\left(\frac{x}{\theta}\right)^{k-1} \exp\left(-\left(\frac{x}{\theta}\right)^{k}\right), & x > 0\,, \\ 0, & \text{otherwise,} \end{cases}$$

where $\theta > 0$ is the **shape** and $k > 0$ the **scale** parameter. The mean of the Weibull distribution is given by $\theta\Gamma(1+1/k)$.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Weibull"
```

Distribution Configuration

These are settings required by this distribution.

Shape

- **Usage**: e["Shape"] = *real number*
- **Description**: The shape of the Weibull distribution.

Scale

- **Usage**: e["Scale"] = *real number*
- **Description**: The scale of the Weibull distribution.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Beta Distribution

Beta

The Beta distribution has the probability density function (PDF):

$$f(x;k,\theta) = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{B(\alpha,\beta)} \quad \text{for } x \in [0,1] \text{ and } \alpha,\beta > 0$$

where α and β are the **shape** parameters.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Beta"
```

Distribution Configuration

These are settings required by this distribution.

Alpha

- **Usage**: e["Alpha"] = *real number*
- **Description**: Shape parameter of the beta distribution.

Beta

- **Usage**: e["Beta"] = *real number*
- **Description**: Shape parameter of the beta distribution.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Ratio of Two Uniform Distribution

UniformRatio

The UniformRatio distribution is the distribution of $Z = X/Y$, where X and Y are two uniformly distributed random variables. The probability density function is:

$$f(z; minX, maxX, minY, maxY) = \begin{cases} (min(maxY, maxX/z)^2 - max(minY, minX/z)^2) * C & z \in [minX/maxY, maxX/min \\ 0, & \text{otherwise,} \end{cases}$$

where $minX$ and $maxX$ are the bounds of the random variable X, and $minY$ and $maxY$ are the bounds of the random variable Y. C is a normalization constant.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Uniformratio"
```

Distribution Configuration

These are settings required by this distribution.

Minimum X

- **Usage**: e["Minimum X"] = *real number*
- **Description**: Lower bound of the first (dividend) uniform distribution.

Maximum X

- **Usage**: e["Maximum X"] = *real number*
- **Description**: Upper bound of the first (divident) uniform distribution.

Minimum Y

- **Usage**: e["Minimum Y"] = *real number*
- **Description**: Lower bound of the second (divisor) uniform distribution.

Maximum Y

- **Usage**: e["Maximum Y"] = *real number*
- **Description**: Upper bound of the second (divisor) uniform distribution.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

LogNormal Distribution

Log-normal

The log-normal distribution has the probability density function (PDF):

$$f(x \mid \mu, \sigma^2) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right), \quad x > 0,$$

where $\mu \in \mathbb{R}$ and $\sigma > 0$ are the parameters associated with the Normal distribution. The **mean** of the PDF is $\exp(\mu + \frac{\sigma^2}{2})$.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/LogNormal"
```

Distribution Configuration

These are settings required by this distribution.

Mu

- **Usage**: e["Mu"] = *real number*
- **Description**: Check: https://en.wikipedia.org/wiki/Log-normal_distribution

Sigma

- **Usage**: e["Sigma"] = *real number*
- **Description**: Check: https://en.wikipedia.org/wiki/Log-normal_distribution

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Normal Distribution

Normal

The normal distribution has the probability density function (PDF):

$$f(x \mid \mu, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$$

where μ is the **mean** and $\sigma > 0$ is the **standard deviation** of the PDF.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Normal"
```

Distribution Configuration

These are settings required by this distribution.

Mean

- **Usage**: e["Mean"] = *real number*
- **Description**: The mean of the Normal (Gaussian) distribution.

Standard Deviation

- **Usage**: e["Standard Deviation"] = *real number*
- **Description**: The standard deviation of the Normal (Gaussian) distribution.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Gamma Distribution

Gamma

The Gamma distribution has the probability density function (PDF):

$$f(x;k,\theta) = \frac{x^{k-1}\, e^{-\frac{x}{\theta}}}{\theta^k\, \Gamma(k)}\,, \quad x > 0\,,$$

where $\theta > 0$ is the **scale** and $k > 0$ the **shape** parameter. The mean of the Gamma distribution is given by θk.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Gamma"
```

Distribution Configuration

These are settings required by this distribution.

Shape

- **Usage**: e["Shape"] = *real number*
- **Description**: The shape parameter of the Gamma distribution, it controls the mean and skewness.

Scale

- **Usage**: e["Scale"] = *real number*
- **Description**: The scale parameter of the Gamma distribution, it controls the mean.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Inverse Gamma Distribution

Inverse Gamma

The inverse Gamma distribution has the probability density function (PDF):

$$f(x;k,\theta) = \frac{1}{x}^{\alpha+1} \frac{\beta^\alpha}{\Gamma(\alpha)} \exp(-\beta/x)\,, \quad x > 0\,,$$

where $\alpha > 0$ is the **shape** parameter, and $\beta > 0$ is the **scale** parameter.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Igamma"
```

Distribution Configuration

These are settings required by this distribution.

Shape

- **Usage**: e[“Shape”] = *real number*
- **Description**: The shape parameter of the inverse gamma distribution.

Scale

- **Usage**: e[“Scale”] = *real number*
- **Description**: The scale parameter of the inverse gamma distribution.

Configuration

These are settings required by this module.

Name

- **Usage**: e[“Name”] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e[“Random Seed”] = *unsigned integer*

- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Geometric Distribution

Geometric

The Geometric distribution has the probability density function (PDF):

$$f(k;p) = (1-p)^k p$$

where $p \in [0,1]$ is the **failure probability** and $k \in \mathbb{Z}^+$ the number of **failures**.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Geometric"
```

Distribution Configuration

These are settings required by this distribution.

Success Probability

- **Usage**: e["Success Probability"] = *real number*
- **Description**: Probability of success of an individual trial.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Laplace Distribution

Laplace

The Laplace distribution has the probability density function (PDF):

$$f(x;\mu,b) = \frac{1}{2b}\exp(-\frac{|x-\mu|}{b})$$

where μ is a the **mean** parameter and $b > 0$ is the **scale** parameter.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Laplace"
```

Distribution Configuration

These are settings required by this distribution.

Mean

- **Usage**: e["Mean"] = *real number*
- **Description**: The mean of the Laplace distribution.

Width

- **Usage**: e["Width"] = *real number*
- **Description**: The scale of the Laplace distribution, it controls the variance.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Truncated Normal Distribution

Truncated normal

The truncated normal distribution has the probability density function:

$$f(x \mid \mu, \sigma^2) = \begin{cases} \left(2\pi\sigma^2\right)^{-1/2} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right), & x \in [a, b], \\ 0, & \text{otherwise}, \end{cases}$$

where μ and $\sigma > 0$ are the parameter associated with the normal distribution.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/TruncatedNormal"
```

Distribution Configuration

These are settings required by this distribution.

Mu

- **Usage**: e["Mu"] = *real number*
- **Description**: The mean of the untruncated Normal distribution.

Sigma

- **Usage**: e["Sigma"] = *real number*
- **Description**: The standard deviation of the untruncated Normal distribution.

Minimum

- **Usage**: e["Minimum"] = *real number*
- **Description**: The lower bound of the truncated Normal distribution.

Maximum

- **Usage**: e["Maximum"] = *real number*
- **Description**: The upper bound of the truncated Normal distribution.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Uniform Distribution

Uniform

The Uniform distribution has the probability density function:

$$f(x; a, b) = \begin{cases} \frac{1}{b-a}\,, & x \in [a, b]\,, \\ 0, & \text{otherwise,} \end{cases}$$

where b is the **upper bound** and $a < b$ the **lower bound**.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Uniform"
```

Distribution Configuration

These are settings required by this distribution.

Minimum

- **Usage**: e["Minimum"] = *real number*
- **Description**: The lower bound of the uniform distribution.

Maximum

- **Usage**: e["Maximum"] = *real number*
- **Description**: The upper bound of the uniform distribution.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Cauchy Distribution

Cauchy

The Cauchy distribution has the probability density function (PDF):

$$f(x; x_0, \gamma) = \frac{1}{\pi\gamma}\left(\frac{\gamma^2}{(x-x_0)^2+\gamma^2}\right)$$

where x_0 is the location parameter, specifying the **mode** of the distribution, and γ is the **scale** parameter which specifies the half-width at half-maximum (HWHM).

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Cauchy"
```

Distribution Configuration

These are settings required by this distribution.

Location

- **Usage**: e["Location"] = *real number*
- **Description**: Specifies the location of the peak of the distribution.

Scale

- **Usage**: e["Scale"] = *real number*
- **Description**: Specifies the half-width at half-maximum (HWHM)

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*

- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Posson Distribution

Poisson

The Poisson distribution has the discrete probability density function (PDF):

$$f(x;\mu) = \frac{\mu^k}{k!} exp(-\mu)$$

where $x \in \mathbb{N}_0$ and $\mu > 0$ is the **mean** and the **variance**.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Poisson"
```

Distribution Configuration

These are settings required by this distribution.

Mean

- **Usage**: e["Mean"] = *real number*
- **Description**: The mean and variance of the distribution.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

Exponential Distribution

Exponential

The Exponential distribution has the probability density function (PDF):

$$f(x;\beta) = \begin{cases} \frac{1}{\beta} e^{-\frac{x}{\beta}} & x \geq 0, \\ 0 & x < 0, \end{cases}$$

where $\beta > 0$ is the **mean** and the **standard deviation**.

Usage

```
e["Distribution"][*index*]["Type"] = "Univariate/Exponential"
```

Distribution Configuration

These are settings required by this distribution.

Location

- **Usage**: e["Location"] = *real number*
- **Description**: Shift for the exponential distribution.

Mean

- **Usage**: e["Mean"] = *real number*
- **Description**: Mean and standard deviation of the (unshifted) exponential distribution.

Configuration

These are settings required by this module.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Name": "",
"Random Seed": 0,
"Range": ""
}
```

1.31.3 Multivariate Distributions

This directory contains a collection of multivariate probability distributions, that define a joint distribution over multiple variables.

Sub-Categories:

Multivariate Normal Distribution

The multivariate Normal distribution has the probability density function (PDF):

$$f(x \mid \mu, \Sigma) = (2\pi)^{-\frac{k}{2}} |\Sigma|^{-\frac{1}{2}} \exp\left(-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)\right)$$

where $\mu \in \mathbb{R}^k$ is the **mean vector** and $\Sigma \in \mathbb{R}^{k \times k}$ is a positive definite **covariance matrix**.

Usage

```
e["Distribution"][*index*]["Type"] = "Multivariate/Normal"
```

Configuration

These are settings required by this module.

Mean Vector

- **Usage**: e["Mean Vector"] = List of *real number*
- **Description**: Means of the variables.

Sigma

- **Usage**: e["Sigma"] = List of *real number*
- **Description**: Cholesky Decomposition of the covariance matrix.

Name

- **Usage**: e["Name"] = *string*
- **Description**: Defines the name of the distribution.

Random Seed

- **Usage**: e["Random Seed"] = *unsigned integer*
- **Description**: Defines the random seed of the distribution.

Range

- **Usage**: e["Range"] = gsl_rng
- **Description**: Stores the current state of the distribution in hexadecimal notation.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Mean Vector": [],
"Name": "",
"Random Seed": 0,
"Range": "",
"Sigma": []
}
```

1.32 Neural Network

This module combines the given *layers* to a Neural Network. The Neural Network can be used, for example, as a function approximator in a *Supervised Learning* problem.

The user can choose between different backends to perform the computations. Besides the Korali lightweight implementation, we support

- Intel's oneDNN library.
- Nvidia's cuDNN library.

This module enables two *Operation* types:

- **Training**: In this operation, the weights and biases are optimized to minimize a given loss function. The initial guess is chosen according to the specified *Weight Initialization*.
- **Inference**: In this operation, weights and biases are either provided by the user, or obtained by the training operation. This configuration is only used to compute the output for a given input.

Sub-Categories:

1.32.1 Layers

This module contains the descriptions of the layers to form a *Neural Network*.

We support

- Intel's oneDNN library.
- Nvidia's cuDNN library.

Each layer consists of a vector $\mathbf{z}^l \in \mathbb{R}^{n_l}$ with *Node Count* elements. Each layer is connected to the previous layer by *Weights*, a *Bias* and an activation function *Activation Function*.

Sub-Categories:

Output Layer

Specialization of the Layer for Output.

Usage

```
eLayer/Output"
```

Configuration

These are settings required by this module.

Transformation Mask

- **Usage**: e["Transformation Mask"] = List of *string*
- **Description**: Indicates a transformation to be performed to the output at the last layer of the neural network. [Order of application on forward propagation: 1/3]

Scale

- **Usage**: e["Scale"] = List of float
- **Description**: Gives a scaling factor for each of the output values of the NN. [Order of application on forward propagation 2/3]

Shift

- **Usage**: e["Shift"] = List of float
- **Description**: Shifts the output of the NN by the values given. [Order of application on forward propagation 3/3]

Output Channels

- **Usage**: e["Output Channels"] = *unsigned integer*
- **Description**: Indicates the size of the output vector produced by the layer.

Weight Scaling

- **Usage**: e["Weight Scaling"] = float
- **Description**: Factor that is mutliplied by the layers' weights.

Engine

- **Usage**: e["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.
- **Options**:
  - "*Korali*": Uses Korali's lightweight NN support. (CPU Sequential - Does not require installing third party software other than Eigen)
  - "*OneDNN*": Uses oneDNN as NN support. (CPU Sequential/Parallel - Requires installing oneDNN)
  - "*CuDNN*": Uses cuDNN as NN support. (GPU - Requires installing cuDNN)

Mode

- **Usage**: e["Mode"] = *string*
- **Description**: Specifies the execution mode of the Neural Network.
- **Options**:
  - "*Training*": Use for training. Stores data during forward propagation and allows backward propagation.
  - "*Inference*": Use for inference only. Only runs forward propagation. Faster for inference.

Layers

- **Usage**: e["Layers"] = knlohmann::json
- **Description**: Complete description of the NN's layers.

Timestep Count

- **Usage**: e["Timestep Count"] = *unsigned integer*
- **Description**: Provides the sequence length for the input/output data.

Batch Sizes

- **Usage**: e["Batch Sizes"] = List of *unsigned integer*
- **Description**: Specifies the batch sizes.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Batch Sizes": [],
"Engine": "Korali",
"Input Values": [],
"Output Channels": 0,
"Scale": [],
"Shift": [],
"Transformation Mask": [],
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": -1.0,
    "Name": "Neural Network / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Weight Scaling": 1.0
}
```

Linear Layer

Specialization of the Layer for Linear Maps. The entries of the previous layer $\mathbf{z}^{l-1} \in \mathbb{R}^{n_{l-1}}$ are multiplied by a weight matrix $W \in \mathbb{R}^{n_l \times n_{l-1}}$ and a bias $\mathbf{b}^l \in \mathbb{R}^{n_l}$.

$$\mathbf{z}^l = W^l \mathbf{z}^{l-1} + \mathbf{b}^l$$

If we denote the components of the vectors by z_i^{l-1} and z_j^l, b_j^l and for the matrix by W_{ij}^l for $i = 1, \ldots, n_{l-1}$ and $j = 1, \ldots, n_l$, this operation can be written as

$$z_j^l = W_{ij}^l z_i^{l-1} + b_j^l$$

Usage

```
eLayer/Linear"
```

Configuration

These are settings required by this module.

Output Channels

- **Usage**: e["Output Channels"] = *unsigned integer*
- **Description**: Indicates the size of the output vector produced by the layer.

Weight Scaling

- **Usage**: e["Weight Scaling"] = float
- **Description**: Factor that is mutliplied by the layers' weights.

Engine

- **Usage**: e["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.
- **Options**:
  - "*Korali*": Uses Korali's lightweight NN support. (CPU Sequential - Does not require installing third party software other than Eigen)
  - "*OneDNN*": Uses oneDNN as NN support. (CPU Sequential/Parallel - Requires installing oneDNN)
  - "*CuDNN*": Uses cuDNN as NN support. (GPU - Requires installing cuDNN)

Mode

- **Usage**: e["Mode"] = *string*
- **Description**: Specifies the execution mode of the Neural Network.
- **Options**:
  - "*Training*": Use for training. Stores data during forward propagation and allows backward propagation.
  - "*Inference*": Use for inference only. Only runs forward propagation. Faster for inference.

Layers

- **Usage**: e["Layers"] = knlohmann::json
- **Description**: Complete description of the NN's layers.

Timestep Count

- **Usage**: e["Timestep Count"] = *unsigned integer*
- **Description**: Provides the sequence length for the input/output data.

Batch Sizes

- **Usage**: e["Batch Sizes"] = List of *unsigned integer*
- **Description**: Specifies the batch sizes.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Batch Sizes": [],
"Engine": "Korali",
"Input Values": [],
"Output Channels": 0,
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": -1.0,
    "Name": "Neural Network / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Weight Scaling": 1.0
}
```

Activation Layer

Specialization of the Layer for activation function. The activation function φ is applied (either element-wise or layer-wise).

$$\mathbf{z} = \varphi(\mathbf{x})$$

Usage

```
eLayer/Activation"
```

Configuration

These are settings required by this module.

Function

- **Usage**: e["Function"] = *string*
- **Description**: Indicates the activation function for the weighted inputs to the current layer.
- **Options**:
  - "*Elementwise/Clip*": Clips the input (s) element-wise into the alpha < s < beta range.
  - "*Elementwise/Linear*": Applies a linear transformation element-wise.
  - "*Elementwise/Log*": Applies the element-wise log function.
  - "*Elementwise/Logistic*": Applies an element-wise logistic (sigmoid) function.
  - "*Elementwise/ReLU*": Applies an element-wise rectifier linear unit function.
  - "*Elementwise/SoftReLU*": Applies an element-wise soft rectifier function (currently only supported with OneDNN and the Eigen library).
  - "*Elementwise/SoftSign*": Applies an element-wise soft sign function (currently only supported with the Eigen library).
  - "*Elementwise/Tanh*": Applies tanh element-wise.
  - "*Softmax*": Applies the layer-wide softmax operation.

Alpha

- **Usage**: e["Alpha"] = float
- **Description**: First (alpha) argument to the activation function, as detailed in https://oneapi-src.github.io/oneDNN/dev_guide_eltwise.html

Beta

- **Usage**: e["Beta"] = float
- **Description**: Second (beta) argument to the activation function, as detailed in https://oneapi-src.github.io/oneDNN/dev_guide_eltwise.html

Output Channels

- **Usage**: e["Output Channels"] = *unsigned integer*
- **Description**: Indicates the size of the output vector produced by the layer.

Weight Scaling

- **Usage**: e["Weight Scaling"] = float
- **Description**: Factor that is mutliplied by the layers' weights.

Engine

- **Usage**: e["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.
- **Options**:
  - "*Korali*": Uses Korali's lightweight NN support. (CPU Sequential - Does not require installing third party software other than Eigen)
  - "*OneDNN*": Uses oneDNN as NN support. (CPU Sequential/Parallel - Requires installing oneDNN)
  - "*CuDNN*": Uses cuDNN as NN support. (GPU - Requires installing cuDNN)

Mode

- **Usage**: e["Mode"] = *string*
- **Description**: Specifies the execution mode of the Neural Network.
- **Options**:
  - "*Training*": Use for training. Stores data during forward propagation and allows backward propagation.
  - "*Inference*": Use for inference only. Only runs forward propagation. Faster for inference.

Layers

- **Usage**: e["Layers"] = knlohmann::json
- **Description**: Complete description of the NN's layers.

Timestep Count

- **Usage**: e["Timestep Count"] = *unsigned integer*
- **Description**: Provides the sequence length for the input/output data.

Batch Sizes

- **Usage**: e["Batch Sizes"] = List of *unsigned integer*
- **Description**: Specifies the batch sizes.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Alpha": 1.0,
"Batch Sizes": [],
"Beta": 0.0,
"Engine": "Korali",
"Input Values": [],
"Output Channels": 0,
"Uniform Generator": {
```

(continues on next page)

(continued from previous page)

```
    "Maximum": 1.0,
    "Minimum": -1.0,
    "Name": "Neural Network / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Weight Scaling": 1.0
}
```

Pooling Layer

Specialization of the Layer for Pooling in Convolutional Neural Networks. The input image of size $IH \times IW$ is padded using PT zeros from the top, PL left, PB bottom and PR rights, respectively. The pooling kernel then beforms the selected function over a Kernel with size $KH \times KW$ and a stride of SV in vertical and SH in horizontal direction respectively.

The dimension $OH \times OW$ of the resulting pooled image can be computed as

$$OH = \lfloor \frac{IH - (KH - (PR + PL))}{SH} \rfloor + 1 OW = \lfloor \frac{IW - (KW - (PT + PB))}{SV} \rfloor + 1$$

Note that the _outputChannels must be specified such that it equals $OH \cdot OW \cdot IC$, where IC is the number of channels in the original image. This guarantees the the image has the same number of channels in the output.

Usage

```
eLayer/Pooling"
```

Configuration

These are settings required by this module.

Function

- **Usage**: e["Function"] = *string*
- **Description**: Indicates the pooling function to apply.
- **Options**:
  - "*Max*": Selects the maximum value in the kernel.
  - "*Inclusive Average*": Selects the average of the values in the kernel, including padding.
  - "*Exclusive Average*": Selects the average of the values in the kernel, excluding padding.

Image Height

- **Usage**: e["Image Height"] = s*unsigned integer*
- **Description**: Height of the incoming 2D image.

Image Width

- **Usage**: e["Image Width"] = s*unsigned integer*
- **Description**: Width of the incoming 2D image.

Kernel Height

- **Usage**: e["Kernel Height"] = s*unsigned integer*
- **Description**: Height of the incoming 2D image.

Kernel Width

- **Usage**: e["Kernel Width"] = s*unsigned integer*
- **Description**: Width of the incoming 2D image.

Vertical Stride

- **Usage**: e["Vertical Stride"] = s*unsigned integer*
- **Description**: Strides for the image on the vertical dimension.

Horizontal Stride

- **Usage**: e["Horizontal Stride"] = s*unsigned integer*
- **Description**: Strides for the image on the horizontal dimension.

Padding Left

- **Usage**: e["Padding Left"] = s*unsigned integer*
- **Description**: Paddings for the image left side.

Padding Right

- **Usage**: e["Padding Right"] = s*unsigned integer*
- **Description**: Paddings for the image right side.

Padding Top

- **Usage**: e["Padding Top"] = s*unsigned integer*
- **Description**: Paddings for the image top side.

Padding Bottom

- **Usage**: e["Padding Bottom"] = s*unsigned integer*
- **Description**: Paddings for the image Bottom side.

Output Channels

- **Usage**: e["Output Channels"] = *unsigned integer*
- **Description**: Indicates the size of the output vector produced by the layer.

Weight Scaling

- **Usage**: e["Weight Scaling"] = float
- **Description**: Factor that is mutliplied by the layers' weights.

Engine

- **Usage**: e["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.
- **Options**:
  - "*Korali*": Uses Korali's lightweight NN support. (CPU Sequential - Does not require installing third party software other than Eigen)
  - "*OneDNN*": Uses oneDNN as NN support. (CPU Sequential/Parallel - Requires installing oneDNN)

  - "*CuDNN*": Uses cuDNN as NN support. (GPU - Requires installing cuDNN)

Mode

- **Usage**: e["Mode"] = *string*
- **Description**: Specifies the execution mode of the Neural Network.
- **Options**:
  - "*Training*": Use for training. Stores data during forward propagation and allows backward propagation.
  - "*Inference*": Use for inference only. Only runs forward propagation. Faster for inference.

Layers

- **Usage**: e["Layers"] = knlohmann::json
- **Description**: Complete description of the NN's layers.

Timestep Count

- **Usage**: e["Timestep Count"] = *unsigned integer*
- **Description**: Provides the sequence length for the input/output data.

Batch Sizes

- **Usage**: e["Batch Sizes"] = List of *unsigned integer*
- **Description**: Specifies the batch sizes.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Batch Sizes": [],
"Engine": "Korali",
"Input Values": [],
"Output Channels": 0,
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": -1.0,
    "Name": "Neural Network / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Weight Scaling": 1.0
}
```

Deconvolution Layer

Specialization of the Layer for Deconvolution in Convolutional Neural Networks. Should contain the same arguments for kernel/padding/stride as the respective Convolution layer that is inverted. The resulting image-size is given as $IH \times IW$. The number of _outputChannels=:math:*IHcdot IWcdot IC* must be specified such that the number of input channels IC takes the wished value.

Usage

```
eLayer/Deconvolution"
```

Configuration

These are settings required by this module.

Image Height

- **Usage**: e["Image Height"] = s*unsigned integer*
- **Description**: Height of the incoming 2D image.

Image Width

- **Usage**: e["Image Width"] = s*unsigned integer*
- **Description**: Width of the incoming 2D image.

Kernel Height

- **Usage**: e["Kernel Height"] = s*unsigned integer*
- **Description**: Height of the incoming 2D image.

Kernel Width

- **Usage**: e["Kernel Width"] = s*unsigned integer*
- **Description**: Width of the incoming 2D image.

Vertical Stride

- **Usage**: e["Vertical Stride"] = s*unsigned integer*
- **Description**: Strides for the image on the vertical dimension.

Horizontal Stride

- **Usage**: e["Horizontal Stride"] = s*unsigned integer*
- **Description**: Strides for the image on the horizontal dimension.

Padding Left

- **Usage**: e["Padding Left"] = s*unsigned integer*
- **Description**: Paddings for the image left side.

Padding Right

- **Usage**: e["Padding Right"] = s*unsigned integer*
- **Description**: Paddings for the image right side.

Padding Top

- **Usage**: e["Padding Top"] = s*unsigned integer*
- **Description**: Paddings for the image top side.

Padding Bottom

- **Usage**: e["Padding Bottom"] = s*unsigned integer*
- **Description**: Paddings for the image Bottom side.

Output Channels

- **Usage**: e["Output Channels"] = *unsigned integer*
- **Description**: Indicates the size of the output vector produced by the layer.

Weight Scaling

- **Usage**: e["Weight Scaling"] = float
- **Description**: Factor that is mutliplied by the layers' weights.

Engine

- **Usage**: e["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.
- **Options**:
  - "*Korali*": Uses Korali's lightweight NN support. (CPU Sequential - Does not require installing third party software other than Eigen)
  - "*OneDNN*": Uses oneDNN as NN support. (CPU Sequential/Parallel - Requires installing oneDNN)
  - "*CuDNN*": Uses cuDNN as NN support. (GPU - Requires installing cuDNN)

Mode

- **Usage**: e["Mode"] = *string*
- **Description**: Specifies the execution mode of the Neural Network.
- **Options**:
  - "*Training*": Use for training. Stores data during forward propagation and allows backward propagation.
  - "*Inference*": Use for inference only. Only runs forward propagation. Faster for inference.

Layers

- **Usage**: e["Layers"] = knlohmann::json
- **Description**: Complete description of the NN's layers.

Timestep Count

- **Usage**: e["Timestep Count"] = *unsigned integer*
- **Description**: Provides the sequence length for the input/output data.

Batch Sizes

- **Usage**: e["Batch Sizes"] = List of *unsigned integer*
- **Description**: Specifies the batch sizes.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Batch Sizes": [],
"Engine": "Korali",
"Input Values": [],
"Output Channels": 0,
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": -1.0,
    "Name": "Neural Network / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Weight Scaling": 1.0
}
```

Recurrent Layer

Specialization of the Layer for Recurrent Neural Networks with further specialization for Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU).

Sub-Categories:

LSTM Layer

Specialization of the Recurrent Layer for Long Short Term Memory (LSTM). The input sequence entries $t \in \{0, \dots, T\}$ denoted by $\mathbf{x}^t \in \mathbb{R}^{n_{l-1}}$ is processed to the output state $\mathbf{h}^{t-1} \in \mathbb{R}^{n_l}$ by

$$\begin{aligned} \mathbf{f}_t &= \sigma_g(W_f\mathbf{x}^t + U_f\mathbf{h}^{t-1} + \mathbf{b}_f)\mathbf{i}_t \\ = \sigma_g(W_i\mathbf{x}^t + U_i\mathbf{h}^{t-1} + \mathbf{b}_i)\mathbf{o}_t &= \sigma_g(W_o\mathbf{x}^t + U_o\mathbf{h}^{t-1} + \mathbf{b}_o)\tilde{\mathbf{c}}_t \\ = \sigma_c(W_c\mathbf{x}^t + U_c\mathbf{h}^{t-1} + \mathbf{b}_c)\mathbf{c}_t &= f_t \circ \mathbf{c}_{t-1} + \mathbf{i}_t \circ \tilde{\mathbf{c}}_t \\ \mathbf{h}_t &= \mathbf{o}_t \circ \sigma_h(\mathbf{c}_t) \end{aligned}$$

We note that for $t = 0$ the needed vectors $\mathbf{h}_{t-1}$ and $\mathbf{c}_{t-1}$ are zero vectors. From the input and the hidden state we compute the forget $\mathbf{f}_t$, the input $\mathbf{i}_t$, and output $\mathbf{o}_t$ activation vector using the respective Weights $W_f, W_i, W_o, U_f, U_i, U_o$ and Biases $\mathbf{b}_f, \mathbf{b}_i, \mathbf{b}_o$. These are used to compute the cell $\tilde{\mathbf{c}}_t$ activation vector via the Weights W_c and Bias $\mathbf{b}_c$. The cell state $\mathbf{c}_t$ is now computed by combining the cell state with the forget and input activation vectors. The output state is obtained by combining the cell state with the output activation vector. The used component-wise non-linearitites are sigmoid for σ_g and hyperbolic tangent functions for σ_c and σ_h. As an illustration we attach a visual representation of the data-flow through an LSTM layer.

Usage

```
eLayer/Recurrent/Lstm"
```

Configuration

These are settings required by this module.

Depth

- **Usage**: e["Depth"] = *unsigned integer*
- **Description**: The number of copies of this layer. This has a better performance than just defining many of these layers manually since it is optimized by the underlying engine.

Output Channels

- **Usage**: e["Output Channels"] = *unsigned integer*
- **Description**: Indicates the size of the output vector produced by the layer.

Weight Scaling

- **Usage**: e["Weight Scaling"] = float
- **Description**: Factor that is mutliplied by the layers' weights.

Engine

- **Usage**: e["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.
- **Options**:
  - "*Korali*": Uses Korali's lightweight NN support. (CPU Sequential - Does not require installing third party software other than Eigen)
  - "*OneDNN*": Uses oneDNN as NN support. (CPU Sequential/Parallel - Requires installing oneDNN)
  - "*CuDNN*": Uses cuDNN as NN support. (GPU - Requires installing cuDNN)

Mode

- **Usage**: e["Mode"] = *string*
- **Description**: Specifies the execution mode of the Neural Network.
- **Options**:
  - "*Training*": Use for training. Stores data during forward propagation and allows backward propagation.
  - "*Inference*": Use for inference only. Only runs forward propagation. Faster for inference.

Layers

- **Usage**: e["Layers"] = knlohmann::json
- **Description**: Complete description of the NN's layers.

Timestep Count

- **Usage**: e["Timestep Count"] = *unsigned integer*
- **Description**: Provides the sequence length for the input/output data.

Batch Sizes

- **Usage**: e["Batch Sizes"] = List of *unsigned integer*
- **Description**: Specifies the batch sizes.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Batch Sizes": [],
"Depth": 1,
"Engine": "Korali",
"Input Values": [],
"Output Channels": 0,
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": -1.0,
    "Name": "Neural Network / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Weight Scaling": 1.0
}
```

GRU Layer

Specialization of the Recurrent Layer for Gated Recurrent Units (GRU). The input sequence entry $t \in \{0, \dots, T\}$ denoted by $\mathbf{z}^t \in \mathbb{R}^{n_{l-1}}$ is processed to the output state $\mathbf{h}^t \in \mathbb{R}^{n_l}$ by

$$
\begin{aligned}
\mathbf{z}_t &= \sigma_g(W_z \mathbf{x}_t + U_z \mathbf{h}_{t-1} + \mathbf{b}_z) \\
\mathbf{r}_t &= \sigma_g(W_r \mathbf{x}_t + U_r \mathbf{h}_{t-1} + \mathbf{b}_r) \\
\hat{\mathbf{h}}_t &= \phi_h(W_h \mathbf{x}_t + U_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h) \\
\mathbf{h}_t &= (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \hat{\mathbf{h}}_t
\end{aligned}
$$

We note that for $t = 0$ the needed vector $\mathbf{h}_{t-1}$ is a zero vector. From the input and the hidden state we compute the gate $\mathbf{z}_t$, and reset $\mathbf{r}_t$ vector using the respective Weights W_z, W_r, U_z, U_r and Biases $\mathbf{b}_z, \mathbf{b}_r$. These are used to compute the output state via the Weights W_h, U_h and Bias $\mathbf{b}_h$. The used component-wise non-linearitites are sigmoid for σ_g and hyperbolic tangent functions for ϕ_h. As an illustration we attach a visual representation of the data-flow through an LSTM layer.

Usage

```
eLayer/Recurrent/Gru"
```

Configuration

These are settings required by this module.

Depth

- **Usage**: e["Depth"] = *unsigned integer*
- **Description**: The number of copies of this layer. This has a better performance than just defining many of these layers manually since it is optimized by the underlying engine.

Output Channels

- **Usage**: e["Output Channels"] = *unsigned integer*
- **Description**: Indicates the size of the output vector produced by the layer.

Weight Scaling

- **Usage**: e["Weight Scaling"] = float
- **Description**: Factor that is mutliplied by the layers' weights.

Engine

- **Usage**: e["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.
- **Options**:
  - "*Korali*": Uses Korali's lightweight NN support. (CPU Sequential - Does not require installing third party software other than Eigen)
  - "*OneDNN*": Uses oneDNN as NN support. (CPU Sequential/Parallel - Requires installing oneDNN)
  - "*CuDNN*": Uses cuDNN as NN support. (GPU - Requires installing cuDNN)

Mode

- **Usage**: e["Mode"] = *string*
- **Description**: Specifies the execution mode of the Neural Network.
- **Options**:
  - "*Training*": Use for training. Stores data during forward propagation and allows backward propagation.
  - "*Inference*": Use for inference only. Only runs forward propagation. Faster for inference.

Layers

- **Usage**: e["Layers"] = knlohmann::json
- **Description**: Complete description of the NN's layers.

Timestep Count

- **Usage**: e["Timestep Count"] = *unsigned integer*
- **Description**: Provides the sequence length for the input/output data.

Batch Sizes

- **Usage**: e["Batch Sizes"] = List of *unsigned integer*
- **Description**: Specifies the batch sizes.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Batch Sizes": [],
"Depth": 1,
"Engine": "Korali",
"Input Values": [],
"Output Channels": 0,
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": -1.0,
    "Name": "Neural Network / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Weight Scaling": 1.0
}
```

Input Layer

Specialization of the Layer for Input.

Usage

```
eLayer/Input"
```

Configuration

These are settings required by this module.

Output Channels

- **Usage**: e["Output Channels"] = *unsigned integer*
- **Description**: Indicates the size of the output vector produced by the layer.

Weight Scaling

- **Usage**: e["Weight Scaling"] = float
- **Description**: Factor that is mutliplied by the layers' weights.

Engine

- **Usage**: e["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.
- **Options**:
  - "*Korali*": Uses Korali's lightweight NN support. (CPU Sequential - Does not require installing third party software other than Eigen)
  - "*OneDNN*": Uses oneDNN as NN support. (CPU Sequential/Parallel - Requires installing oneDNN)
  - "*CuDNN*": Uses cuDNN as NN support. (GPU - Requires installing cuDNN)

Mode

- **Usage**: e["Mode"] = *string*
- **Description**: Specifies the execution mode of the Neural Network.
- **Options**:
  - "*Training*": Use for training. Stores data during forward propagation and allows backward propagation.
  - "*Inference*": Use for inference only. Only runs forward propagation. Faster for inference.

Layers

- **Usage**: e["Layers"] = knlohmann::json
- **Description**: Complete description of the NN's layers.

Timestep Count

- **Usage**: e["Timestep Count"] = *unsigned integer*
- **Description**: Provides the sequence length for the input/output data.

Batch Sizes

- **Usage**: e["Batch Sizes"] = List of *unsigned integer*
- **Description**: Specifies the batch sizes.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Batch Sizes": [],
"Engine": "Korali",
"Input Values": [],
"Output Channels": 0,
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": -1.0,
    "Name": "Neural Network / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Weight Scaling": 1.0
}
```

Convolutional Layer

Specialization of the Layer for Convolution in Convolutional Neural Networks. The input image of size $IH \times IW$ with IC channels is padded using PT zeros from the top, PL left, PB bottom and PR rights, respectively. The trainable convolution has size $KH \times KW$ and OC channels and performs the operation using a stride of SV in vertical and SH in horizontal direction respectively.

The dimension $OC \times OH \times OW$ of the resulting convoluted image can be computed as

$$OH = \lfloor \frac{IH - (KH - (PR + PL))}{SH} \rfloor + 1 OW = \lfloor \frac{IW - (KW - (PT + PB))}{SV} \rfloor + 1 OC = \frac{}{OH \cdot OW}$$

We note that the _outputChannels must be specified such that OC takes the wished value.

Usage

```
eLayer/Convolution"
```

Configuration

These are settings required by this module.

Image Height

- **Usage**: e[“Image Height”] = s*unsigned integer*
- **Description**: Height of the incoming 2D image.

Image Width

- **Usage**: e[“Image Width”] = s*unsigned integer*
- **Description**: Width of the incoming 2D image.

Kernel Height

- **Usage**: e[“Kernel Height”] = s*unsigned integer*
- **Description**: Height of the incoming 2D image.

Kernel Width

- **Usage**: e[“Kernel Width”] = s*unsigned integer*
- **Description**: Width of the incoming 2D image.

Vertical Stride

- **Usage**: e[“Vertical Stride”] = s*unsigned integer*
- **Description**: Strides for the image on the vertical dimension.

Horizontal Stride

- **Usage**: e[“Horizontal Stride”] = s*unsigned integer*
- **Description**: Strides for the image on the horizontal dimension.

Padding Left

- **Usage**: e[“Padding Left”] = s*unsigned integer*
- **Description**: Paddings for the image left side.

Padding Right

- **Usage**: e[“Padding Right”] = s*unsigned integer*
- **Description**: Paddings for the image right side.

Padding Top

- **Usage**: e[“Padding Top”] = s*unsigned integer*
- **Description**: Paddings for the image top side.

Padding Bottom

- **Usage**: e["Padding Bottom"] = s*unsigned integer*
- **Description**: Paddings for the image Bottom side.

Output Channels

- **Usage**: e["Output Channels"] = *unsigned integer*
- **Description**: Indicates the size of the output vector produced by the layer.

Weight Scaling

- **Usage**: e["Weight Scaling"] = float
- **Description**: Factor that is mutliplied by the layers' weights.

Engine

- **Usage**: e["Engine"] = *string*
- **Description**: Specifies which Neural Network backend engine to use.
- **Options**:
  - "*Korali*": Uses Korali's lightweight NN support. (CPU Sequential - Does not require installing third party software other than Eigen)
  - "*OneDNN*": Uses oneDNN as NN support. (CPU Sequential/Parallel - Requires installing oneDNN)
  - "*CuDNN*": Uses cuDNN as NN support. (GPU - Requires installing cuDNN)

Mode

- **Usage**: e["Mode"] = *string*
- **Description**: Specifies the execution mode of the Neural Network.
- **Options**:
  - "*Training*": Use for training. Stores data during forward propagation and allows backward propagation.
  - "*Inference*": Use for inference only. Only runs forward propagation. Faster for inference.

Layers

- **Usage**: e["Layers"] = knlohmann::json
- **Description**: Complete description of the NN's layers.

Timestep Count

- **Usage**: e["Timestep Count"] = *unsigned integer*
- **Description**: Provides the sequence length for the input/output data.

Batch Sizes

- **Usage**: e["Batch Sizes"] = List of *unsigned integer*
- **Description**: Specifies the batch sizes.

Default Configuration

These following configuration will be assigned by default. Any settings defined by the user will override the given settings specified in these defaults.

```
{
"Batch Sizes": [],
"Engine": "Korali",
"Input Values": [],
"Output Channels": 0,
"Uniform Generator": {
    "Maximum": 1.0,
    "Minimum": -1.0,
    "Name": "Neural Network / Uniform Generator",
    "Type": "Univariate/Uniform"
    },
"Weight Scaling": 1.0
}
```

1.33 Extending Korali

Korali can be extended with new algorithms and problems by adding modules. Modules are plug-and-play packages consisting of base C++ code (*.cpp.base*, *.hpp.base* files), configuration (*.config* file), and documentation (*README.rst* file).

To create a new Korali module, you need to create a new folder inside the */source/modules/* folder. You will find examples of existing modules that you can use as basis for your new module. After creating the module, you need to add it to the list of selectable modules in the *source/modules/module.cpp* file.

1.33.1 Base Code

The base module code files (*.base*) contain the definition of the Module's C++ class and its component functions. The base files contain certain markers that trigger the automatic generation of code upon building. These markers should not be removed. Building Korali will result in end *.hpp* and *.cpp* files. These files should be included in the git repository as generated and never manually modified (any changes will be overwritten in the next build).

1.33.2 Documentation

In the *README.rst* file, the author of the module must add a detailed description of the module's purpose and rationale, including any publications that can serve as further explanation.

1.33.3 Configuration

After creating the new folder, you need to create a configuration file with extension *.config* that describes the internal aspects of the module. We explain below the purpose of each of the category contained therein.

Module Data

This category contains basic information about the Module (e.g., class name and namespace). This information should be accurately defined for the build to work.

Configuration Settings

Each of the entries in this category represents a user-configurable aspect of the module. Upon building Korali, they become public fields in the module's C++ class.

Variables Configuration

Each of the entries in this category represents a parameter of the Korali experiment's variables. These settings are user-configurable, and they are accessed by prefixing ["Variables"][numerical_index]. Upon building Korali, they become fields in the korali::Variable class file.

Internal Settings

Each of the entries in this category represents a developer-only aspect of the module. These settings are not user-configurable, but their purpose is to automatize the serialization/deserialization of the internal state of the module. Upon building Korali, they become fields in the module's C++ class file.

Module Defaults

Defines the default configuration of all the parameters that describe the module. They are overwritten by any user-specified values.

Variable Defaults

Defines the default configuration for parameters of the korali::Variable class. They are applied to all the variables defined in the experiment. They are overwritten by any user-specified values.

(Solvers Only) Termination Criteria

Each of the entries in this category determines a criterion that Korali checks at each generation to determine whether to continue or finish execution. These settings are user-configurable, but they are accessed by prefixing ["Solver"]["Termination Criteria"]. Upon building Korali, they become fields in the module's C++ class with prefix _terminationCriteria.

(Problems Only) Compatible Solvers

List of all the solvers that can be used with the current problem type. Solvers are specified by name and must produce at least one result.

(Problems Only) Results

List of all the results that can be obtained by running this problem. Each result specifies which solvers can produce them. It is the task of the developer to make sure that the promised results are indeed produced by the solvers here specified.

(Problems Only) Available Operations

Lists all the operations that the problem can perform on a sample. Each operation links to an actual C++ method in the class.

1.34 Code Documentation

class *korali*::*neuralNetwork*::*layer*::**Activation** : **public** *korali*::*neuralNetwork*::*Layer*

#include <activation.hpp> Class declaration for module: *Activation*.

Public Functions

virtual void **getConfiguration**(knlohmann::json *&js*) **override**

Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json *&js*) **override**

Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json *&js*) **override**

Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**

Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**

Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **createForwardPipeline**() **override**

Initializes the layer's internal memory structures for the forward pipeline.

virtual void **createBackwardPipeline**() **override**

Initializes the internal memory structures for the backward pipeline.

virtual void **forwardData**(**const** size_t *t*) **override**

Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

virtual void **backwardData**(**const** size_t *t*) **override**

Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

Public Members

std::string **_function**
Indicates the activation function for the weighted inputs to the current layer.

float **_alpha**
First (alpha) argument to the activation function, as detailed in https://oneapi-src.github.io/oneDNN/dev_guide_eltwise.html.

float **_beta**
Second (beta) argument to the activation function, as detailed in https://oneapi-src.github.io/oneDNN/dev_guide_eltwise.html.

class *korali*::*solver*::*optimizer*::**AdaBelief** : **public** *korali*::*solver*::*Optimizer*
#include <AdaBelief.hpp> Class declaration for module: *AdaBelief*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **processResult**(double *evaluation*, *std*::vector<double> &*gradient*)
Takes a sample evaluation and its gradient and calculates the next set of parameters.

Parameters

- **evaluation** – The value of the objective function at the current set of parameters
- **gradient** – The gradient of the objective function at the current set of parameters

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

virtual void **runGeneration**() **override**
Runs the current generation.

virtual void **printGenerationBefore**() **override**
Prints solver information before the execution of the current generation.

virtual void **printGenerationAfter**() **override**
Prints solver information after the execution of the current generation.

Public Members

double **_beta1**
Smoothing factor for momentum update.

double **_beta2**
Smoothing for gradient update.

double **_eta**
Learning Rate (Step Size)

double **_epsilon**
Term to facilitate numerical stability.

std::vector<double> **_currentVariable**
[Internal Use] Current value of parameters.

std::vector<double> **_gradient**
[Internal Use] Gradient of Function with respect to Parameters.

std::vector<double> **_bestEverGradient**
[Internal Use] Gradient of function with respect to Best Ever Variables.

double **_gradientNorm**
[Internal Use] Norm of gradient of function with respect to Parameters.

std::vector<double> **_firstMoment**
[Internal Use] Estimate of first moment of Gradient.

std::vector<double> **_biasCorrectedFirstMoment**
[Internal Use] Bias corrected estimate of first moment of Gradient.

std::vector<double> **_secondCentralMoment**
[Internal Use] Previous estimate of second moment of Gradient.

std::vector<double> **_biasCorrectedSecondCentralMoment**
[Internal Use] Bias corrected estimate of second moment of Gradient.

double **_minGradientNorm**
[Termination Criteria] Specifies the minimal norm for the gradient of function with respect to Parameters.

double **_maxGradientNorm**
[Termination Criteria] Specifies the minimal norm for the gradient of function with respect to Parameters.

class *korali*::*solver*::*optimizer*::**Adam** : **public** *korali*::*solver*::*Optimizer*
#include <Adam.hpp> Class declaration for module: *Adam*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **processResult**(double *evaluation*, *std*::vector<double> &*gradient*)
Takes a sample evaluation and its gradient and calculates the next set of parameters.

Parameters

- **evaluation** – The value of the objective function at the current set of parameters
- **gradient** – The gradient of the objective function at the current set of parameters

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

virtual void **runGeneration**() **override**
Runs the current generation.

virtual void **printGenerationBefore**() **override**
Prints solver information before the execution of the current generation.

virtual void **printGenerationAfter**() **override**
Prints solver information after the execution of the current generation.

Public Members

double **_beta1**
Smoothing factor for momentum update.

double **_beta2**
Smoothing factor for gradient update.

double **_eta**
Learning Rate (Step Size)

double **_epsilon**
Term to facilitate numerical stability.

std::vector<double> **_currentVariable**
[Internal Use] Current value of parameters.

std::vector<double> **_gradient**
[Internal Use] Gradient of Function with respect to Parameters.

std::vector<double> **_bestEverGradient**
[Internal Use] Gradient of function with respect to Best Ever Variables.

std::vector<double> **_squaredGradient**
[Internal Use] Square of gradient of function with respect to Parameters.

double **_gradientNorm**
[Internal Use] Norm of gradient of function with respect to Parameters.

std::vector<double> **_firstMoment**
[Internal Use] Estimate of first moment of Gradient.

std::vector<double> **_biasCorrectedFirstMoment**
[Internal Use] Bias corrected estimate of first moment of Gradient.

std::vector<double> **_secondMoment**
[Internal Use] Old estimate of second moment of Gradient.

std::vector<double> **_biasCorrectedSecondMoment**
[Internal Use] Bias corrected estimate of second moment of Gradient.

double **_minGradientNorm**
[Termination Criteria] Specifies the minimal norm for the gradient of function with respect to Parameters.

double **_maxGradientNorm**
[Termination Criteria] Specifies the minimal norm for the gradient of function with respect to Parameters.

class *korali*::*solver*::**Agent** : **public** *korali*::*Solver*
#include <agent.hpp> Class declaration for module: *Agent*.

Subclassed by *korali::solver::agent::Continuous*, *korali::solver::agent::Discrete*

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **normalizeStateActionNeuralNetwork**(*NeuralNetwork* **neuralNetwork*, size_t *miniBatchSize*, size_t *normalizationSteps*)

Mini-batch based normalization routine for Neural Networks with state and action inputs (typically critics)

Parameters

- **neuralNetwork** – Neural Network to normalize
- **miniBatchSize** – Number of entries in the normalization minibatch
- **normalizationSteps** – How many normalization steps to perform (and grab the average)

void **normalizeStateNeuralNetwork**(*NeuralNetwork* **neuralNetwork*, size_t *miniBatchSize*, size_t *normalizationSteps*)

Mini-batch based normalization routine for Neural Networks with state inputs only (typically policy)

Parameters

- **neuralNetwork** – Neural Network to normalize
- **miniBatchSize** – Number of entries in the normalization minibatch
- **normalizationSteps** – How many normalization steps to perform (and grab the average)

void **averageRewardsAcrossAgents**(knlohmann::json *&message*)

Average rewards across agents per experience in multi agent framework.

Parameters **message** – A json object containing all experiences from all agents.

void **processEpisode**(knlohmann::json *&episode*)

Additional post-processing of episode after episode terminated.

Parameters **episode** – A vector of experiences pertaining to the episode.

std::vector<*std*::pair<size_t, size_t>> **generateMiniBatch**()

Generates an experience mini batch from the replay memory.

Returns A vector of pairs with the indexes to the experiences and agents in the mini batch

std::vector<*std*::vector<*std*::vector<float>>> **getMiniBatchStateSequence**(**const** *std*::vector<*std*::pair<size_t, size_t>> *&miniBatch*)

Gets a vector of states corresponding of time sequence corresponding to the provided last experience index.

Parameters **miniBatch** – Indexes to the latest experiences in a batch of sequences

Returns The time step vector of states

void **updateExperienceMetadata**(**const** *std*::vector<*std*::pair<size_t, size_t>> *&miniBatch*, **const** *std*::vector<*policy_t*> *&policyData*)

Updates the state value, retrace, importance weight and other metadata for a given minibatch of experiences.

Parameters

- **miniBatch** – The mini batch of experience ids to update
- **policyData** – The policy to use to evaluate the experiences

void **resetTimeSequence**()

Resets time sequence within the agent, to forget past actions from other episodes.

virtual float **calculateStateValue**(**const** *std*::vector<*std*::vector<float>> &*stateSequence*, size_t *policyIdx* = 0) = 0

Function to pass a state time series through the NN and calculates the action probabilities, along with any additional information.

Parameters

- **stateSequence** – The batch of state time series (Format: BxTxS, B is batch size, T is the time series lenght, and S is the state size)
- **policyIdx** – The index for the policy for which the state-value is computed

Returns A JSON object containing the information produced by the policies given the current state series

virtual void **runPolicy**(**const** *std*::vector<*std*::vector<*std*::vector<float>>> &*stateSequenceBatch*, *std*::vector<*policy_t*> &*policy*, size_t *policyIdx* = 0) = 0

Function to pass a state time series through the NN and calculates the action probabilities, along with any additional information.

Parameters

- **stateSequenceBatch** – The batch of state time series (Format: BxTxS, B is batch size, T is the time series lenght, and S is the state size)
- **policy** – Vector with policy objects that is filled after forwarding the policy
- **policyIdx** – The index for the policy for which the state-value is computed

size_t **getTimeSequenceStartExpId**(size_t *expId*)

Calculates the starting experience index of the time sequence for the selected experience.

Parameters **expId** – The index of the latest experience in the sequence

Returns The starting time sequence index

std::vector<*std*::vector<float>> **getTruncatedStateSequence**(size_t *expId*, size_t *agentId*)

Gets a vector of states corresponding of time sequence corresponding to the provided second-to-last experience index for which a truncated state exists.

Parameters

- **expId** – The index of the second-to-latest experience in the sequence
- **agentId** – The index of the agent

Returns The time step vector of states, including the truncated state

virtual float **calculateImportanceWeight**(**const** *std*::vector<float> &*action*, **const** *policy_t* &*curPolicy*, **const** *policy_t* &*oldPolicy*) = 0

Calculates importance weight of current action from old and new policies.

Parameters

- **action** – The action taken
- **curPolicy** – The current policy
- **oldPolicy** – The old policy, the one used for take the action in the first place

Returns The importance weight

void **attendWorker**(**const** size_t *workerId*)

Listens to incoming experience from the given agent, sends back policy or terminates the episode depending on what's needed.

Parameters **workerId** – The worker's ID

void **serializeExperienceReplay**()
Serializes the experience replay into a JSON compatible format.

void **deserializeExperienceReplay**()
Deserializes a JSON object into the experience replay.

void **trainingGeneration**()
Runs a generation when running in training mode.

void **testingGeneration**()
Runs a generation when running in testing mode.

void **rescaleStates**()
Rescales states to have a zero mean and unit variance.

inline float **getScaledReward**(**const** float *reward*)
Rescales a given reward by the square root of the sum of squarred rewards.

Parameters **reward** – the input reward to rescale

Returns The normalized reward

virtual void **trainPolicy**() = 0
Trains the policy, based on the new experiences.

virtual knlohmann::json **getPolicy**() = 0
Obtains the policy hyperaparamters from the learner for the agent to generate new actions.

Returns The current policy hyperparameters

virtual void **setPolicy**(**const** knlohmann::json &*hyperparameters*) = 0
Updates the agent's hyperparameters.

Parameters **hyperparameters** – The hyperparameters to update the agent.

virtual void **initializeAgent**() = 0
Initializes the internal state of the policy.

virtual void **printInformation**() = 0
Prints information about the training policy.

virtual void **getAction**(*korali*::*Sample* &*sample*) = 0
Gathers the next action either from the policy or randomly.

Parameters **sample** – *Sample* on which the action and metadata will be stored

virtual void **runGeneration**() **override**
Runs the current generation.

virtual void **printGenerationAfter**() **override**
Prints solver information after the execution of the current generation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

Public Members

std::string **_mode**
Specifies the operation mode for the agent.

std::vector<size_t> **_testingSampleIds**
A vector with the identifiers for the samples to test the hyperparameters with.

knlohmann::json **_testingCurrentPolicies**
The current hyperparameters of the policies to test.

size_t **_trainingAverageDepth**
Specifies the depth of the running training average to report.

size_t **_concurrentWorkers**
Indicates the number of concurrent environments to use to collect experiences.

size_t **_episodesPerGeneration**
Number of reinforcement learning episodes per *Korali* generation (checkpoints are generated between generations).

size_t **_miniBatchSize**
The number of experiences to randomly select to train the neural network(s) with.

size_t **_timeSequenceLength**
Indicates the number of contiguous experiences to pass to the NN for learning. This is only useful when using recurrent NNs.

float **_learningRate**
The initial learning rate to use for the NN hyperparameter optimization.

int **_l2RegularizationEnabled**
Boolean to determine if l2 regularization will be applied to the neural networks.

float **_l2RegularizationImportance**
Coefficient for l2 regularization.

knlohmann::json **_neuralNetworkHiddenLayers**
Indicates the configuration of the hidden neural network layers.

std::string **_neuralNetworkOptimizer**
Indicates the optimizer algorithm to update the NN hyperparameters.

std::string **_neuralNetworkEngine**
Specifies which Neural Network backend to use.

float **_discountFactor**
Represents the discount factor to weight future experiences.

float **_importanceWeightTruncationLevel**
Represents the discount factor to weight future experiences.

int **_experienceReplaySerialize**
Indicates whether to serialize and store the experience replay after each generation. Disabling will reduce I/O overheads but will disable the checkpoint/resume function.

size_t **_experienceReplayStartSize**
The minimum number of experiences before learning starts.

size_t **_experienceReplayMaximumSize**
The size of the replay memory. If this number is exceeded, experiences are deleted.

float **_experienceReplayOffPolicyCutoffScale**
Initial Cut-Off to classify experiences as on- or off-policy. (c_max in https://arxiv.org/abs/1807.05827)

float **_experienceReplayOffPolicyTarget**
Target fraction of off-policy experiences in the replay memory. (D in https://arxiv.org/abs/1807.05827)

float **_experienceReplayOffPolicyAnnealingRate**
Annealing rate for Off Policy Cutoff Scale and Learning Rate. (A in https://arxiv.org/abs/1807.05827)

float **_experienceReplayOffPolicyREFERBeta**
Initial value for the penalisation coefficient for off-policiness. (beta in https://arxiv.org/abs/1807.05827)

float **_experiencesBetweenPolicyUpdates**
The number of experiences to receive before training/updating (real number, may be less than < 1.0, for more than one update per experience).

int **_stateRescalingEnabled**
Determines whether to normalize the states, such that they have mean 0 and standard deviation 1 (done only once after the initial exploration phase).

int **_rewardRescalingEnabled**
Determines whether to normalize the rewards, such that they have mean 0 and standard deviation 1.

std::string **_multiAgentRelationship**
Specifies whether we are in an individual setting or collaborator setting.

int **_multiAgentCorrelation**
Specifies whether we take into account the dependencies of the agents or not.

std::string **_multiAgentSampling**
Specifies how to sample the minibatch.

size_t **_policyParameterCount**
[Internal Use] Stores the number of parameters that determine the probability distribution for the current state sequence.

std::vector<float> **_actionLowerBounds**
[Internal Use] Lower bounds for actions.

std::vector<float> **_actionUpperBounds**
[Internal Use] Upper bounds for actions.

size_t **_currentEpisode**
[Internal Use] Indicates the current episode being processed.

std::vector<*std*::vector<float>> **_trainingRewardHistory**
[Internal Use] Keeps a history of all training episode rewards.

std::vector<size_t> **_trainingExperienceHistory**
[Internal Use] Keeps a history of all training episode experience counts.

std::vector<float> **_testingAverageRewardHistory**
[Internal Use] Keeps a history of all training episode rewards.

std::vector<float> **_trainingAverageReward**
[Internal Use] Contains a running average of the training episode rewards.

std::vector<float> **_trainingLastReward**
[Internal Use] Remembers the cumulative sum of rewards for the last training episode.

std::vector<float> **_trainingBestReward**
[Internal Use] Remembers the best cumulative sum of rewards found so far in any episodes.

std::vector<size_t> **_trainingBestEpisodeId**
[Internal Use] Remembers the episode that obtained the maximum cumulative sum of rewards found so far.

knlohmann::json **_trainingCurrentPolicies**
[Internal Use] Stores the current training policies configuration.

knlohmann::json **_trainingBestPolicies**
[Internal Use] Stores the best training policies configuration found so far.

std::vector<float> **_testingReward**
[Internal Use] The cumulative sum of rewards obtained when evaluating the testing samples.

float **_testingBestReward**
[Internal Use] Remembers the best cumulative sum of rewards from latest testing episodes, if any.

float **_testingWorstReward**
[Internal Use] Remembers the worst cumulative sum of rewards from latest testing episodes, if any.

size_t **_testingBestEpisodeId**
[Internal Use] Remembers the episode Id that obtained the maximum cumulative sum of rewards found so far during testing.

size_t **_testingCandidateCount**
[Internal Use] Remembers the number of candidate policies tested so far.

float **_testingAverageReward**
[Internal Use] Remembers the average cumulative sum of rewards from latest testing episodes, if any.

float **_testingBestAverageReward**
[Internal Use] Remembers the best cumulative sum of rewards found so far from testing episodes.

knlohmann::json **_testingBestPolicies**
[Internal Use] Stores the best testing policies configuration found so far.

std::vector<size_t> **_experienceReplayOffPolicyCount**
[Internal Use] Number of off-policy experiences in the experience replay.

std::vector<float> **_experienceReplayOffPolicyRatio**
[Internal Use] Current off policy ratio in the experience replay.

float **_experienceReplayOffPolicyCurrentCutoff**
[Internal Use] Indicates the current cutoff to classify experiences as on- or off-policy.

std::vector<float> **_experienceReplayOffPolicyREFERCurrentBeta**
[Internal Use] Vector of the current penalisation coefficient for off-policiness for each agent.

float **_currentLearningRate**
[Internal Use] The current learning rate to use for the NN hyperparameter optimization.

size_t **_policyUpdateCount**
[Internal Use] Keeps track of the number of policy updates that have been performed.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Uniform random number generator.

size_t **_experienceCount**
[Internal Use] Count of the number of experiences produced so far.

float **_rewardRescalingSigma**
[Internal Use] Contains the standard deviation of the rewards. They will be scaled by this value in order to normalize the reward distribution in the RM.

float **_rewardRescalingSumSquaredRewards**
[Internal Use] Sum of squared rewards in experience replay.

std::vector<*std*::vector<float>> **_stateRescalingMeans**
[Internal Use] Contains the mean of the states. They will be shifted by this value in order to normalize the state distribution in the RM.

std::vector<*std*::vector<float>> **_stateRescalingSigmas**
[Internal Use] Contains the standard deviations of the states. They will be scaled by this value in order to normalize the state distribution in the RM.

size_t **_effectiveMinibatchSize**
[Internal Use] Effective Minibatch Size in the context of MARL.

size_t **_maxEpisodes**
[Termination Criteria] The solver will stop when the given number of episodes have been run.

size_t **_maxExperiences**
[Termination Criteria] The solver will stop when the given number of experiences have been gathered.

size_t **_maxPolicyUpdates**
[Termination Criteria] The solver will stop when the given number of optimization steps have been performed.

std::vector<*Sample*> **_workers**
Array of workers collecting new experiences.

std::vector<bool> **_isWorkerRunning**
Keeps track of the workers.

std::vector<*solver*::*DeepSupervisor**> **_criticPolicyLearner**
Pointer to training the actor network.

std::vector<*korali*::*Experiment*> **_criticPolicyExperiment**
Korali experiment for obtaining the agent's action.

std::vector<*problem*::*SupervisedLearning**> **_criticPolicyProblem**
Pointer to actor's experiment problem.

size_t **_sessionExperienceCount**
Session-specific experience count. This is useful in case of restart: counters from the old session won't count.

size_t **_sessionEpisodeCount**
Session-specific episode count. This is useful in case of restart: counters from the old session won't count.

size_t **_sessionGeneration**
Session-specific generation count. This is useful in case of restart: counters from the old session won't count.

size_t **_sessionPolicyUpdateCount**
Session-specific policy update count. This is useful in case of restart: counters from the old session won't count.

size_t **_sessionExperiencesUntilStartSize**
Session-specific counter that keeps track of how many experiences need to be obtained this session to reach the start training threshold.

cBuffer<*std*::vector<*std*::vector<float>>> **_stateBuffer**
Stores the state of the experience.

cBuffer<*std*::vector<*std*::vector<float>>> **_actionBuffer**
Stores the action taken by the agent.

std::vector<*cBuffer*<*std*::vector<float>>> **_stateTimeSequence**
Stores the current sequence of states observed by the agent (limited to time sequence length defined by the user)

cBuffer<size_t> **_episodeIdBuffer**
Episode that experience belongs to.

cBuffer<size_t> **_episodePosBuffer**
Position within the episode of this experience.

cBuffer<*std*::vector<float>> **_importanceWeightBuffer**
Contains the latest calculation of the experience's importance weight.

cBuffer<float> **_truncatedImportanceWeightBufferContiguous**
Contains the latest calculation of the experience's truncated importance weight (for cache optimzed update of retV in updateExperienceMetadata)

cBuffer<float> **_productImportanceWeightBuffer**
Contains the latest calculation of the product of the product of the experience's importance weights.

cBuffer<*std*::vector<*policy_t*>> **_curPolicyBuffer**
Contains the most current policy information given the experience state.

cBuffer<*std*::vector<*policy_t*>> **_expPolicyBuffer**
Contains the policy information produced at the moment of the action was taken.

cBuffer<*std*::vector<char>> **_isOnPolicyBuffer**
Indicates whether the experience is on policy, given the specified off-policiness criteria.

cBuffer<*termination_t*> **_terminationBuffer**
Specifies whether the experience is terminal (truncated or normal) or not.

cBuffer<float> **_retraceValueBufferContiguous**
Contains the result of the retrace (Vtbc) function for the currrent experience (for cache optimzed update of retV in updateExperienceMetadata)

cBuffer<*std*::vector<float>> **_truncatedStateValueBuffer**
If this is a truncated terminal experience, this contains the state value for that state.

cBuffer<*std*::vector<*std*::vector<float>>> **_truncatedStateBuffer**
If this is a truncated terminal experience, the truncated state is also saved here.

cBuffer<float> **_rewardBufferContiguous**
Contains the rewards of every experience (for cache optimzed update of retV in updateExperienceMetadata)

cBuffer<float> **_stateValueBufferContiguous**
Contains the state value evaluation for every experience (for cache optimzed update of retV in updateExperienceMetadata)

float **_priorityAnnealingRate**
Stores the priority annealing rate.

float **_importanceWeightAnnealingRate**
Stores the importance weight annealing factor.

problem::*ReinforcementLearning* ***_problem**
Storage for the pointer to the learning problem.

double **_sessionRunningTime**
[Profiling] Measures the amount of time taken by the generation

double **_sessionSerializationTime**
[Profiling] Measures the amount of time taken by ER serialization

double **_sessionWorkerComputationTime**
[Profiling] Stores the computation time per episode taken by Workers

double **_sessionWorkerCommunicationTime**
[Profiling] Measures the average communication time per episode taken by Workers

double **_sessionPolicyEvaluationTime**
[Profiling] Measures the average policy evaluation time per episode taken by Workers

double **_sessionPolicyUpdateTime**
[Profiling] Measures the time taken to update the policy in the current generation

double **_sessionWorkerAttendingTime**
[Profiling] Measures the time taken to update the attend the agent's state

double **_generationRunningTime**
[Profiling] Measures the amount of time taken by the generation

double **_generationSerializationTime**
[Profiling] Measures the amount of time taken by ER serialization

double **_generationWorkerComputationTime**
[Profiling] Stores the computation time per episode taken by worker

double **_generationWorkerCommunicationTime**
[Profiling] Measures the average communication time per episode taken by Workers

double **_generationPolicyEvaluationTime**
[Profiling] Measures the average policy evaluation time per episode taken by Workers

double **_generationPolicyUpdateTime**
[Profiling] Measures the time taken to update the policy in the current generation

double **_generationWorkerAttendingTime**
[Profiling] Measures the time taken to update the attend the agent's state

class *korali*::*problem*::**Bayesian** : **public** *korali*::*Problem*
#include <bayesian.hpp> Class declaration for module: *Bayesian*.

Subclassed by *korali::problem::bayesian::Custom*, *korali::problem::bayesian::Reference*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **evaluate**(*korali*::*Sample* &*sample*)
Produces a generic evaluation from the Posterior distribution of the sample, for optimization with CMAES, DEA, storing it in and stores it in sample["F(x)"].

Parameters **sample** – A *Korali Sample*

void **evaluateLogPrior**(*korali*::*Sample* &*sample*)
Evaluates the log prior of the given sample, and stores it in sample["Log Prior"].

Parameters **sample** – A *Korali Sample*

virtual void **evaluateLoglikelihood**(*korali*::*Sample* &*sample*) = 0
Evaluates the log likelihood of the given sample, and stores it in sample["Log Likelihood"].

Parameters **sample** – A *Korali Sample*

void **evaluateLogPosterior**(*korali*::*Sample* &*sample*)
Evaluates the log posterior of the given sample, and stores it in sample["Log Posterior"].

Parameters **sample** – A *Korali Sample*

virtual void **evaluateGradient**(*korali*::*Sample* &*sample*)
Evaluates the gradient of the objective w.r.t. to the variables, and stores it in sample["Gradient"].

Parameters **sample** – A *Korali Sample*

void **evaluateLogPriorGradient**(*korali*::*Sample* &*sample*)
Evaluates the gradient of the logPrior w.r.t. to the variables, and stores it in sample["logPrior Gradient"].

Parameters **sample** – A *Korali Sample*

inline virtual void **evaluateLoglikelihoodGradient**(*korali*::*Sample* &*sample*)
Evaluates the gradient of the logLikelikood w.r.t. to the variables, and stores it in sample["logLikelihood Gradient"].

Parameters **sample** – A *Korali Sample*

virtual void **evaluateHessian**(*korali*::*Sample* &*sample*)
Evaluates the hessian of the objective w.r.t. to the variables, and stores it in sample["Hessian"].

Parameters **sample** – A *Korali Sample*

void **evaluateLogPriorHessian**(*korali*::*Sample* &*sample*)
Evaluates the gradient of the logPrior w.r.t. to the variables, and stores it in sample["logPrior Hessian"].

Parameters **sample** – A *Korali Sample*

inline virtual void **evaluateLogLikelihoodHessian**(*korali*::*Sample* &*sample*)
Evaluates the gradient of the logLikelikood w.r.t. to the variables, and stores it in sample["logLikelihood Hessian"].

Parameters **sample** – A *Korali Sample*

inline virtual void **evaluateFisherInformation**(*korali*::*Sample* &*sample*)
Evaluates the empirical Fisher information.

Parameters **sample** – A *Korali Sample*

class *korali*::*distribution*::*univariate*::**Beta** : **public** *korali*::*distribution*::*Univariate*
#include <beta.hpp> Class declaration for module: *Beta*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian** (double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber** () **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_alpha**
[Conditional Variable Value] Shape parameter of the beta distribution.

std::string **_alphaConditional**
[Conditional Variable Reference] Shape parameter of the beta distribution.

double **_beta**
[Conditional Variable Value] Shape parameter of the beta distribution.

std::string **_betaConditional**
[Conditional Variable Reference] Shape parameter of the beta distribution.

template<typename **valType**, typename **timerType**>
struct *korali*::**cacheElement_t**
#include <kcache.hpp> Struct that defines an element present in *Korali*'s cache structure.

Public Members

valType **value**
Value of the element.

timerType **time**
Time when the element was last updated.

class *korali*::*distribution*::*univariate*::**Cauchy** : **public** *korali*::*distribution*::*Univariate*
#include <cauchy.hpp> Class declaration for module: *Cauchy*.

Public Functions

virtual void **getConfiguration** (knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration** (knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults** (knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_location**
[Conditional Variable Value] Specifies the location of the peak of the distribution.

std::string **_locationConditional**
[Conditional Variable Reference] Specifies the location of the peak of the distribution.

double **_scale**
[Conditional Variable Value] Specifies the half-width at half-maximum (HWHM)

std::string **_scaleConditional**
[Conditional Variable Reference] Specifies the half-width at half-maximum (HWHM)

template<typename **T**>
class *korali*::**cBuffer**
#include <cbuffer.hpp> This class defines a circular buffer with overwrite policy on add.

Public Functions

inline cBuffer()
Default constructor.

inline cBuffer(size_t *size*)
Constructor with a specific size.

Parameters **size** – The buffer size

inline size_t **size**()
Returns the current number of elements in the buffer.

Returns The number of elements

inline void **resize**(size_t *maxSize*)
Returns the current number of elements in the buffer.

Parameters **maxSize** – The buffer size

inline void **add**(**const** *T* &*v*)
Adds an element to the buffer.

Parameters **v** – The element to add

inline *std*::vector<*T*> **getVector**()
Returns the elements of the buffer in a vector format.

Returns The vector with the circular buffer elements

inline void **clear**()
Eliminates all contents of the buffer.

inline *T* &**operator[]**(size_t *pos*) **const**
Accesses an element at the required position.

Parameters **pos** – The access position

Returns The element corresponding to the position

Private Members

size_t **_maxSize**
Size of buffer container.

size_t **_size**
Number of elements already added.

std::unique_ptr<*T*[]> **_data**
Container for data.

size_t **_start**
Position of the start of the buffer.

size_t **_end**
Position of the end of the buffer.

class *korali*::*solver*::*optimizer*::**CMAES** : **public** *korali*::*solver*::*Optimizer*
#include <CMAES.hpp> Class declaration for module: *CMAES*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **prepareGeneration**()
Prepares generation for the next set of evaluations.

void **sampleSingle**(size_t *sampleIdx*, **const** *std*::vector<double> &*randomNumbers*)
Evaluates a single sample.

Parameters

- **sampleIdx** – Index of the sample to evaluate
- **randomNumbers** – Random numbers to generate sample

void **adaptC**(int *hsig*)
Adapts the covariance matrix.

Parameters **hsig** – Sign

void **updateSigma**()
Updates scaling factor of covariance matrix.

void **updateDistribution**()
Updates mean and covariance of Gaussian proposal distribution.

void **updateEigensystem**(**const** *std*::vector<double> &*M*)
Updates the system of eigenvalues and eigenvectors.

Parameters **M** – Input matrix

void **numericalErrorTreatment**()
Method that checks potential numerical issues and does correction. Not yet implemented.

void **eigen**(size_t *N*, **const** *std*::vector<double> &*C*, *std*::vector<double> &*diag*, *std*::vector<double> &*Q*) **const**
Function for eigenvalue decomposition.

Parameters

- **N** – Matrix size
- **C** – Input matrix

- **diag** – Sorted eigenvalues
- **Q** – eingenvectors of C

void **sort_index**(**const** *std*::vector<double> &*vec*, *std*::vector<size_t> &*_sortingIndex*, size_t *N*) **const**
Descending sort of vector elements, stores ordering in _sortingIndex.

Parameters

- **_sortingIndex** – Ordering of elements in vector
- **vec** – Vector to sort
- **N** – Number of current samples.

void **initMuWeights**(size_t *numsamples*)
Initializes the weights of the mu vector.

Parameters **numsamples** – Length of mu vector

void **initCovariance**()
Initialize Covariance Matrix and Cholesky Decomposition.

void **checkMeanAndSetRegime**()
Check if mean of proposal distribution is inside of valid domain (does not violate constraints), if yes, re-initialize internal vars. Method for CCMA-ES.

void **updateConstraints**()
Update constraint evaluationsa. Method for CCMA-ES.

void **updateViabilityBoundaries**()
Update viability boundaries. Method for CCMA-ES.

void **handleConstraints**()
Process samples that violate constraints. Method for CCMA-ES.

void **reEvaluateConstraints**()
Reevaluate constraint evaluations. Called in handleConstraints. Method for CCMA-ES.

void **updateDiscreteMutationMatrix**()
Update mutation matrix for discrete variables. Method for discrete/integer optimization.

void **discretize**(*std*::vector<double> &*sample*)
Discretize variables to given granularity using arithmetic rounding.

Parameters **sample** – *Sample* to discretize

virtual void **setInitialConfiguration**() **override**
Configures CMA-ES.

virtual void **runGeneration**() **override**
Executes sampling & evaluation generation.

virtual void **printGenerationBefore**() **override**
Console Output before generation runs.

virtual void **printGenerationAfter**() **override**
Console output after generation.

virtual void **finalize**() **override**
Final console output at termination.

Public Members

size_t **_populationSize**
: Specifies the number of samples to evaluate per generation (preferably $4+3*log(N)$, where N is the number of variables).

size_t **_muValue**
: Number of best samples (offspring samples) used to update the covariance matrix and the mean (by default it is half the *Sample* Count).

std::string **_muType**
: Weights given to the Mu best values to update the covariance matrix and the mean.

double **_initialSigmaCumulationFactor**
: Controls the learning rate of the conjugate evolution path (by default this variable is internally calibrated).

double **_initialDampFactor**
: Controls the updates of the covariance matrix scaling factor (by default this variable is internally calibrated).

int **_useGradientInformation**
: Include gradient information for proposal distribution update.

float **_gradientStepSize**
: Scaling factor for gradient step, only relevant if gradient information used.

int **_isSigmaBounded**
: Sets an upper bound for the covariance matrix scaling factor. The upper bound is given by the average of the initial standard deviation of the variables.

double **_initialCumulativeCovariance**
: Controls the learning rate of the evolution path for the covariance update (must be in (0,1], by default this variable is internally calibrated).

int **_diagonalCovariance**
: Covariance matrix updates will be optimized for diagonal matrices.

int **_mirroredSampling**
: Generate the negative counterpart of each random number during sampling.

size_t **_viabilityPopulationSize**
: Specifies the number of samples per generation during the viability regime, i.e. during the search for a parameter vector not violating the constraints.

size_t **_viabilityMuValue**
: Number of best samples used to update the covariance matrix and the mean during the viability regime (by default this variable is half the Viability *Sample* Count).

size_t **_maxCovarianceMatrixCorrections**
: Max number of covairance matrix adaptions per generation during the constraint handling loop.

double **_targetSuccessRate**
: Controls the updates of the covariance matrix scaling factor during the viability regime.

double **_covarianceMatrixAdaptionStrength**
: Controls the covariane matrix adaption strength if samples violate constraints.

double **_normalVectorLearningRate**
: Learning rate of constraint normal vectors (must be in (0, 1], by default this variable is internally calibrated).

double **_globalSuccessLearningRate**
Learning rate of success probability of objective function improvements.

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**
[Internal Use] Normal random number generator.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Uniform random number generator.

int **_isViabilityRegime**
[Internal Use] True if mean is outside feasible domain. During viability regime CMA-ES is working with relaxed constraint boundaries that contract towards the true constraint boundaries.

std::vector<double> **_valueVector**
[Internal Use] Objective function values.

std::vector<*std*::vector<double>> **_gradients**
[Internal Use] Gradients of objective function evaluations.

size_t **_currentPopulationSize**
[Internal Use] Actual number of samples used per generation (Population Size or Viability Population Size).

size_t **_currentMuValue**
[Internal Use] Actual value of mu (Mu Value or Viability Mu Value).

std::vector<double> **_muWeights**
[Internal Use] Calibrated Weights for each of the Mu offspring samples.

double **_effectiveMu**
[Internal Use] Variance effective selection mass.

double **_sigmaCumulationFactor**
[Internal Use] Increment for sigma, calculated from muEffective and dimension.

double **_dampFactor**
[Internal Use] Dampening parameter controls step size adaption.

double **_cumulativeCovariance**
[Internal Use] Controls the step size adaption.

double **_chiSquareNumber**
[Internal Use] Expectation of $||N(0,I)||^2$.

size_t **_covarianceEigenvalueEvaluationFrequency**
[Internal Use] Establishes how frequently the eigenvalues are updated.

double **_sigma**
[Internal Use] Determines the step size.

double **_trace**
[Internal Use] The trace of the initial covariance matrix.

std::vector<*std*::vector<double>> **_samplePopulation**
[Internal Use] *Sample* coordinate information.

size_t **_finishedSampleCount**
[Internal Use] Counter of evaluated samples to terminate evaluation.

std::vector<double> **_currentBestVariables**
[Internal Use] Best variables of current generation.

double **_previousBestEverValue**
[Internal Use] Best ever model evaluation as of previous generation.

std::vector<size_t> **_sortingIndex**
[Internal Use] Sorted indeces of samples according to their model evaluation.

std::vector<double> **_covarianceMatrix**
[Internal Use] (Unscaled) covariance Matrix of proposal distribution.

std::vector<double> **_auxiliarCovarianceMatrix**
[Internal Use] Temporary Storage for Covariance Matrix.

std::vector<double> **_covarianceEigenvectorMatrix**
[Internal Use] Matrix with eigenvectors in columns.

std::vector<double> **_auxiliarCovarianceEigenvectorMatrix**
[Internal Use] Temporary Storage for Matrix with eigenvectors in columns.

std::vector<double> **_axisLengths**
[Internal Use] Axis lengths (sqrt(Evals))

std::vector<double> **_auxiliarAxisLengths**
[Internal Use] Temporary storage for Axis lengths.

std::vector<double> **_bDZMatrix**
[Internal Use] Temporary storage.

std::vector<double> **_auxiliarBDZMatrix**
[Internal Use] Temporary storage.

std::vector<double> **_currentMean**
[Internal Use] Current mean of proposal distribution.

std::vector<double> **_previousMean**
[Internal Use] Previous mean of proposal distribution.

std::vector<double> **_meanUpdate**
[Internal Use] Update differential from previous to current mean.

std::vector<double> **_evolutionPath**
[Internal Use] Evolution path for Covariance Matrix update.

std::vector<double> **_conjugateEvolutionPath**
[Internal Use] Conjugate evolution path for sigma update.

double **_conjugateEvolutionPathL2Norm**
[Internal Use] L2 Norm of the conjugate evolution path.

double **_maximumDiagonalCovarianceMatrixElement**
[Internal Use] Maximum diagonal element of the Covariance Matrix.

double **_minimumDiagonalCovarianceMatrixElement**
[Internal Use] Minimum diagonal element of the Covariance Matrix.

double **_maximumCovarianceEigenvalue**
[Internal Use] Maximum Covariance Matrix Eigenvalue.

double **_minimumCovarianceEigenvalue**
[Internal Use] Minimum Covariance Matrix Eigenvalue.

int **_isEigensystemUpdated**
[Internal Use] Flag determining if the covariance eigensystem is up to date.

std::vector<*std*::vector<int>> **_viabilityIndicator**
[Internal Use] Evaluation of each constraint for each sample.

int **_hasConstraints**
[Internal Use] True if the number of constraints is higher than zero.

double **_covarianceMatrixAdaptionFactor**
[Internal Use] This is the beta factor that indicates how fast the covariance matrix is adapted.

int **_bestValidSample**
[Internal Use] Index of best sample without constraint violations (otherwise -1).

double **_globalSuccessRate**
[Internal Use] Estimated Global Success Rate, required for calibration of covariance matrix scaling factor updates.

double **_viabilityFunctionValue**
[Internal Use] Viability Function Value.

size_t **_resampledParameterCount**
[Internal Use] Number of resampled parameters due constraint violation.

size_t **_covarianceMatrixAdaptationCount**
[Internal Use] Number of Covariance Matrix Adaptations.

std::vector<double> **_viabilityBoundaries**
[Internal Use] Viability Boundaries.

std::vector<int> **_viabilityImprovement**
[Internal Use] *Sample* evaluations larger than fviability.

size_t **_maxConstraintViolationCount**
[Internal Use] Temporary counter of maximal amount of constraint violations attained by a sample (must be 0).

std::vector<size_t> **_sampleConstraintViolationCounts**
[Internal Use] Maximal amount of constraint violations.

std::vector<*std*::vector<double>> **_constraintEvaluations**
[Internal Use] Functions to be evaluated as constraint evaluations, if the return from any of them is > 0, then the constraint is met.

std::vector<*std*::vector<double>> **_normalConstraintApproximation**
[Internal Use] Normal approximation of constraints.

std::vector<double> **_bestConstraintEvaluations**
[Internal Use] Constraint evaluations for best ever.

int **_hasDiscreteVariables**
[Internal Use] Flag indicating if at least one of the variables is discrete.

std::vector<double> **_discreteMutations**
[Internal Use] Vector storing discrete mutations, required for covariance matrix update.

size_t **_numberOfDiscreteMutations**
[Internal Use] Number of discrete mutations in current generation.

size_t **_numberMaskingMatrixEntries**
[Internal Use] Number of nonzero entries on diagonal in Masking Matrix.

std::vector<double> **_maskingMatrix**
[Internal Use] Diagonal Matrix signifying where an integer mutation may be conducted.

std::vector<double> **_maskingMatrixSigma**
[Internal Use] Sigma of the Masking Matrix.

double **_chiSquareNumberDiscreteMutations**
[Internal Use] Expectation of $||N(0,I^S)||^2$ for discrete mutations.

double **_currentMinStandardDeviation**
[Internal Use] Current minimum standard deviation of any variable.

double **_currentMaxStandardDeviation**
[Internal Use] Current maximum standard deviation of any variable.

size_t **_constraintEvaluationCount**
[Internal Use] Number of Constraint Evaluations.

double **_maxConditionCovarianceMatrix**
[Termination Criteria] Specifies the maximum condition of the covariance matrix.

double **_minStandardDeviation**
[Termination Criteria] Specifies the minimal standard deviation for any variable in any proposed sample.

double **_maxStandardDeviation**
[Termination Criteria] Specifies the maximal standard deviation for any variable in any proposed sample.

class *korali*::*conduit*::**Concurrent** : **public** *korali*::*Conduit*
#include <concurrent.hpp> Class declaration for module: *Concurrent*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **isRoot**() **const** **override**
Determines whether the caller rank/thread/process is root.

Returns True, if it is root; false, otherwise.

virtual void **initServer**() **override**
Initializes the worker/server bifurcation in the conduit.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **terminateServer**() **override**
Finalizes the workers.

virtual void **stackEngine**(*Engine* **engine*) **override**
Stacks a new *Engine* into the engine stack.

Parameters **engine** – A *Korali Engine*

virtual void **popEngine**() **override**
Pops the current *Engine* from the engine stack.

virtual void **listenWorkers**() **override**
(*Engine* <- Worker) Receives all pending incoming messages and stores them into the corresponding sample's message queue.

virtual void **broadcastMessageToWorkers**(knlohmann::json &*message*) **override**
(*Engine* -> Worker) Broadcasts a message to all workers

Parameters **message** – JSON object with information to broadcast

virtual void **sendMessageToEngine**(knlohmann::json &*message*) **override**
(*Sample* -> *Engine*) Sends an update to the engine to provide partial information while the sample is still active

Parameters **message** – Message to send to engine

virtual knlohmann::json **recvMessageFromEngine**() **override**
(*Sample* <- *Engine*) Blocking call that waits until any message incoming from the engine.

Returns message from the engine.

virtual void **sendMessageToSample**(*Sample* &*sample*, knlohmann::json &*message*) **override**
(*Engine* -> *Sample*) Sends an update to a still active sample

Parameters

- **sample** – The sample from which to receive an update
- **message** – Message to send to the sample.

virtual size_t **getProcessId**() **const** **override**
Returns the identifier corresponding to the executing process (to differentiate their random seeds)

Returns The executing process id

virtual size_t **getWorkerCount**() **const** **override**
Get total *Korali* worker count in the conduit.

Returns The number of workers

Public Members

size_t **_concurrentJobs**
Specifies the number of worker processes (jobs) running concurrently.

std::vector<pid_t> **_workerPids**
PID of worker processes.

int **_workerId**
Worker Id for current workers - 0 for the master process.

std::vector<*std*::vector<int>> **_resultContentPipe**
OS Pipe to handle result contents communication coming from worker processes.

std::vector<*std*::vector<int>> **_resultSizePipe**
OS Pipe to handle result size communication coming from worker processes.

std::vector<*std*::vector<int>> **_inputsPipe**
OS Pipe to handle sample parameter communication to worker processes.

struct *korali*::*problem*::*hierarchical*::*Psi*::**conditionalPriorInfo**
Stores the pre-computed positions (pointers) of the conditional priors to evaluate for performance.

Public Members

std::vector<size_t> **_samplePositions**
Stores the position of the conditional prior.

std::vector<double*> **_samplePointers**
Stores the pointer of the conditional prior.

class *korali*::**Conduit** : **public** *korali*::*Module*
#include <conduit.hpp> Class declaration for module: *Conduit*.

Subclassed by *korali::conduit::Concurrent*, *korali::conduit::Distributed*, *korali::conduit::Sequential*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **worker**()
Lifetime function for korali workers.

inline **virtual** bool **isRoot**() **const**
Determines whether the caller rank/thread/process is root.

Returns True, if it is root; false, otherwise.

inline **virtual** bool **isWorkerLeadRank**() **const**
Determines whether the caller rank is the leader of its worker root.

Returns True, if it is the worker leader rank; false, otherwise.

void **workerProcessSample**(**const** knlohmann::json &*js*)
(Worker Side) Starts the processing of a sample at the worker side

Parameters **js** – Contains sample's input data and metadata

void **workerStackEngine**(**const** knlohmann::json &*js*)
(Worker Side) Accepts and stacks an incoming *Korali* engine from the main process

Parameters **js** – Contains *Engine*'s input data and metadata

void **workerPopEngine**()
(Worker Side) Pops the top of the engine stack

void **start** (*Sample* &*sample*)
Starts the execution of the sample.

Parameters **sample** – A *Korali* sample

void **wait** (*Sample* &*sample*)
Waits for a given sample to finish. The experiment will not continue until the sample has been evaluated.

Parameters **sample** – A *Korali* sample

void **waitAll** (*std*::vector<*Sample*> &*samples*)
Waits for a set of sample to finish. The experiment will not continue until all samples have been evaluated.

Parameters **samples** – A list of *Korali* samples

size_t **waitAny** (*std*::vector<*Sample*> &*samples*)
Waits for a set of sample to finish. The experiment will not continue until at least one of the samples have been evaluated.

Parameters **samples** – A list of *Korali* samples

Returns Position in the vector of the sample that has finished.

virtual void **stackEngine** (*Engine* **engine*) = 0
Stacks a new *Engine* into the engine stack.

Parameters **engine** – A *Korali Engine*

virtual void **popEngine** () = 0
Pops the current *Engine* from the engine stack.

void **runSample** (*Sample* **sample*, *Engine* **engine*)
Starts the execution of a sample, given an *Engine*.

Parameters

- **sample** – the sample to execute
- **engine** – The *Korali* engine to use for its execution

virtual void **initServer** () = 0
Initializes the worker/server bifurcation in the conduit.

virtual void **terminateServer** () = 0
Finalizes the workers.

virtual void **broadcastMessageToWorkers** (knlohmann::json &*message*) = 0
(*Engine* -> Worker) Broadcasts a message to all workers

Parameters **message** – JSON object with information to broadcast

virtual void **listenWorkers** () = 0
(*Engine* <- Worker) Receives all pending incoming messages and stores them into the corresponding sample's message queue.

void **listen** (*std*::vector<*Sample*> &*samples*)
Start pending samples and retrieve any pending messages for them.

Parameters **samples** – The set of samples

virtual void **sendMessageToEngine** (knlohmann::json &*message*) = 0
(*Sample* -> *Engine*) Sends an update to the engine to provide partial information while the sample is still active

Parameters **message** – Message to send to engine

virtual knlohmann::json **recvMessageFromEngine**() = 0
(*Sample* <- *Engine*) Blocking call that waits until any message incoming from the engine.

Returns message from the engine.

virtual void **sendMessageToSample**(*Sample* &*sample*, knlohmann::json &*message*) = 0
(*Engine* -> *Sample*) Sends an update to a still active sample

Parameters

- **sample** – The sample from which to receive an update
- **message** – Message to send to the sample.

virtual size_t **getProcessId**() **const** = 0
Returns the identifier corresponding to the executing process (to differentiate their random seeds)

Returns The executing process id

virtual size_t **getWorkerCount**() **const** = 0
Get total *Korali* worker count in the conduit.

Returns The number of workers

Public Members

std::deque<size_t> **_workerQueue**
Double ended queue to store idle workers to assign samples to.

std::map<size_t, *Sample**> **_workerToSampleMap**
Map that links workers to their currently-executing sample.

Public Static Functions

static void **coroutineWrapper**()
Wrapper function for the sample coroutine.

class *korali*::*solver*::*agent*::**Continuous** : **public** *korali*::*solver*::*Agent*
#include <continuous.hpp> Class declaration for module: *Continuous*.

Subclassed by *korali::solver::agent::continuous::VRACER*

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

std::vector<float> **calculateImportanceWeightGradient**(**const** *std*::vector<float> &*action*, **const** *policy_t* &*curPolicy*, **const** *policy_t* &*oldPolicy*, **const** float *importanceWeight*)
Calculates the gradient of teh importance weight wrt to the parameter of the 2nd (current) distribution evaluated at old action.

Parameters

- **action** – The action taken by the agent in the given experience
- **oldPolicy** – The policy for the given state used at the time the action was performed
- **curPolicy** – The current policy for the given state
- **importanceWeight** – The importance weight

Returns gradient of policy wrt curParamsOne and curParamsTwo

std::vector<float> **calculateKLDivergenceGradient**(**const** *policy_t* &*oldPolicy*, **const** *policy_t* &*curPolicy*)
Calculates the gradient of KL(p_old, p_cur) wrt to the parameter of the 2nd (current) distribution.

Parameters

- **oldPolicy** – The policy for the given state used at the time the action was performed
- **curPolicy** – The current policy for the given state

Returns

std::vector<float> **generateTrainingAction**(*policy_t* &*curPolicy*)
Function to generate randomized actions from neural network output.

Parameters **curPolicy** – The current policy for the given state

Returns An action vector

std::vector<float> **generateTestingAction**(**const** *policy_t* &*curPolicy*)
Function to generate deterministic actions from neural network output required for policy evaluation, respectively testing.

Parameters **curPolicy** – The current policy for the given state

Returns An action vector

virtual float **calculateImportanceWeight**(**const** *std*::vector<float> &*action*, **const** *policy_t* &*curPolicy*, **const** *policy_t* &*oldPolicy*) **override**
Calculates importance weight of current action from old and new policies.

Parameters

- **action** – The action taken
- **curPolicy** – The current policy
- **oldPolicy** – The old policy, the one used for take the action in the first place

Returns The importance weight

virtual void **getAction**(*korali*::*Sample* &*sample*) **override**
Gathers the next action either from the policy or randomly.

Parameters **sample** – *Sample* on which the action and metadata will be stored

virtual void **initializeAgent**() **override**
Initializes the internal state of the policy.

Public Members

std::string **_policyDistribution**
Specifies which probability distribution to use for the policy.

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**
[Internal Use] Gaussian random number generator to generate the agent's action.

std::vector<float> **_actionShifts**
[Internal Use] Shifts required for bounded actions.

std::vector<float> **_actionScales**
[Internal Use] Scales required for bounded actions (half the action domain width).

std::vector<*std*::string> **_policyParameterTransformationMasks**
[Internal Use] Stores the transformations required for each parameter.

std::vector<float> **_policyParameterScaling**
[Internal Use] Stores the scaling required for the parameter after the transformation is applied.

std::vector<float> **_policyParameterShifting**
[Internal Use] Stores the shifting required for the parameter after the scaling is applied.

problem::*reinforcementLearning*::*Continuous* ***_problem**
Storage for the pointer to the (continuous) learning problem.

class *korali*::*problem*::*reinforcementLearning*::**Continuous** : **public** *korali*::*problem*::*ReinforcementLearnin*
#include <continuous.hpp> Class declaration for module: *Continuous*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

class *korali*::*neuralNetwork*::*layer*::**Convolution** : **public** *korali*::*neuralNetwork*::*Layer*
#include <convolution.hpp> Class declaration for module: *Convolution*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **copyHyperparameterPointers**(*Layer* *dstLayer) **override**
Replicates the pointers for the current layer onto a destination layer.

Parameters **dstLayer** – The destination layer onto which to copy the pointers

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual *std*::vector<float> **generateInitialHyperparameters**() **override**
Generates the initial weight/bias hyperparameters for the layer.

Returns The initial hyperparameters

virtual void **createHyperparameterMemory**() **override**
Initializes the layer's internal memory structures for hyperparameter storage.

virtual void **createForwardPipeline**() **override**
Initializes the layer's internal memory structures for the forward pipeline.

virtual void **createBackwardPipeline**() **override**
Initializes the internal memory structures for the backward pipeline.

virtual void **forwardData**(**const** size_t *t*) **override**
Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

virtual void **setHyperparameters**(**const** float **hyperparameters*) **override**
Updates layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (*Input*) Pointer to read the hyperparameters from.

virtual void **getHyperparameters**(float **hyperparameters*) **override**
Gets layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (*Output*) Pointer to write the hyperparameters to.

virtual void **getHyperparameterGradients**(float **gradient*) **override**
Gets the gradients of the layer's output wrt to is hyperparameters (e.g., weights and biases)

Parameters **gradient** – (*Output*) Pointer to write the hyperparameter gradients to.

virtual void **backwardData**(**const** size_t *t*) **override**
Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

virtual void **backwardHyperparameters**(**const** size_t *t*) **override**
Calculates the gradients of layer hyperparameters.

Parameters **t** – Indicates the current timestep

Public Members

ssize_t **_imageHeight**
Height of the incoming 2D image.

ssize_t **_imageWidth**
Width of the incoming 2D image.

ssize_t **_kernelHeight**
Height of the incoming 2D image.

ssize_t **_kernelWidth**
Width of the incoming 2D image.

ssize_t **_verticalStride**
Strides for the image on the vertical dimension.

ssize_t **_horizontalStride**
Strides for the image on the horizontal dimension.

ssize_t **_paddingLeft**
Paddings for the image left side.

ssize_t **_paddingRight**
Paddings for the image right side.

ssize_t **_paddingTop**
Paddings for the image top side.

ssize_t **_paddingBottom**
Paddings for the image Bottom side.

ssize_t **N**
Pre-calculated value for Mini-Batch Size.

ssize_t **IC**
Pre-calculated value for *Input* Channels.

ssize_t **IH**
Pre-calculated value for *Input* Image Height.

ssize_t **IW**
Pre-calculated value for *Input* Image Width.

ssize_t **OC**
Pre-calculated value for *Output* Channels.

ssize_t **OH**
Pre-calculated value for *Output* Image Height.

ssize_t **OW**
Pre-calculated value for *Output* Image Width.

ssize_t **KH**
Pre-calculated value for Kernel Image Height.

ssize_t **KW**
Pre-calculated value for Kernel Image Width.

ssize_t **PL**
Pre-calculated values for padding left.

ssize_t **PR**
Pre-calculated values for padding right.

ssize_t **PT**
Pre-calculated values for padding top.

ssize_t **PB**
Pre-calculated values for padding bottom.

ssize_t **SH**
Pre-calculated values for horizontal stride.

ssize_t **SV**
Pre-calculated values for vertical stride.

class *korali*::*problem*::*bayesian*::**Custom** : **public** *korali*::*problem*::*Bayesian*
#include <custom.hpp> Class declaration for module: *Custom*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **evaluateLoglikelihood**(*korali*::*Sample* &*sample*) **override**
Evaluates the log likelihood of the given sample, and stores it in sample["Log Likelihood"].

Parameters **sample** – A *Korali Sample*

virtual void **evaluateLoglikelihoodGradient**(*korali*::*Sample* &*sample*) **override**
Evaluates the gradient of the logLikelikood w.r.t. to the variables, and stores it in sample["logLikelihood Gradient"].

Parameters **sample** – A *Korali Sample*

virtual void **evaluateFisherInformation**(*korali*::*Sample* &*sample*) **override**
Evaluates the empirical Fisher information.

Parameters **sample** – A *Korali Sample*

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

Public Members

std::uint64_t **_likelihoodModel**
Stores the user-defined likelihood model. It should return the value of the Log Likelihood of the given sample.

class *korali*::*solver*::*optimizer*::**DEA** : **public** *korali*::*solver*::*Optimizer*
#include <DEA.hpp> Class declaration for module: *DEA*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **setInitialConfiguration**() **override**
Configures Differential Evolution/.

virtual void **runGeneration**() **override**
Executes sampling & evaluation generation.

virtual void **printGenerationBefore**() **override**
Console Output before generation runs.

virtual void **printGenerationAfter**() **override**
Console output after generation.

virtual void **finalize**() **override**
Final console output at termination.

Public Members

size_t **_populationSize**
Specifies the number of samples to evaluate per generation (preferably 5-10x the number of variables).

double **_crossoverRate**
Controls the rate at which dimensions of the samples are mixed (must be in [0,1]).

double **_mutationRate**
Controls the scaling of the vector differentials (must be in [0,2], preferably < 1).

std::string **_mutationRule**
Controls the Mutation Rate.

std::string **_parentSelectionRule**
Defines the selection rule of the parent vector.

std::string **_acceptRule**
Sets the accept rule after sample mutation and evaluation.

int **_fixInfeasible**
If set true, *Korali* samples a random sample between Parent and the voiolated boundary. If set false, infeasible samples are mutated again until feasible.

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**
[Internal Use] Normal random number generator.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Uniform random number generator.

std::vector<double> **_valueVector**
[Internal Use] Objective Function Values.

std::vector<double> **_previousValueVector**
[Internal Use] Objective Function Values from previous evaluations.

std::vector<*std*::vector<double>> **_samplePopulation**
[Internal Use] *Sample* variable information.

std::vector<*std*::vector<double>> **_candidatePopulation**
[Internal Use] *Sample* candidates variable information.

size_t **_bestSampleIndex**
[Internal Use] Index of the best sample in current generation.

double **_previousBestEverValue**
[Internal Use] Best ever model evaluation as of previous generation.

std::vector<double> **_currentMean**
[Internal Use] Current mean of population.

std::vector<double> **_previousMean**
[Internal Use] Previous mean of population.

std::vector<double> **_currentBestVariables**
[Internal Use] Best variables of current generation.

std::vector<double> **_maxDistances**
[Internal Use] Max distance between samples per dimension.

double **_currentMinimumStepSize**
[Internal Use] Minimum step size of any variable in the current generation.

double **_minValue**
[Termination Criteria] Specifies the target fitness to stop minimization.

double **_minStepSize**
[Termination Criteria] Specifies the minimal step size of the population mean from one gneration to another.

Private Functions

void **mutateSingle**(size_t *sampleIdx*)
Mutate a sample.

Parameters **sampleIdx** – Index of sample to be mutated.

void **fixInfeasible**(size_t *sampleIdx*)
Fix sample params that are outside of domain.

Parameters **sampleIdx** – Index of sample that is outside of domain.

void **updateSolver**(*std*::vector<*Sample*> &*samples*)
Update the state of Differential Evolution.

Parameters **samples** – *Sample* evaluations.

void **initSamples**()
Create new set of candidates.

void **prepareGeneration**()
Mutate samples and distribute them.

class *korali*::*neuralNetwork*::*layer*::**Deconvolution** : **public** *korali*::*neuralNetwork*::*Layer*
#include <deconvolution.hpp> Class declaration for module: *Deconvolution*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **copyHyperparameterPointers**(*Layer* *dstLayer) **override**
Replicates the pointers for the current layer onto a destination layer.

Parameters **dstLayer** – The destination layer onto which to copy the pointers

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual *std*::vector<float> **generateInitialHyperparameters**() **override**
Generates the initial weight/bias hyperparameters for the layer.

Returns The initial hyperparameters

virtual void **createHyperparameterMemory**() **override**
Initializes the layer's internal memory structures for hyperparameter storage.

virtual void **createForwardPipeline**() **override**
Initializes the layer's internal memory structures for the forward pipeline.

virtual void **createBackwardPipeline**() **override**
Initializes the internal memory structures for the backward pipeline.

virtual void **forwardData**(**const** size_t *t*) **override**
Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

virtual void **setHyperparameters**(**const** float **hyperparameters*) **override**
Updates layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (*Input*) Pointer to read the hyperparameters from.

virtual void **getHyperparameters**(float **hyperparameters*) **override**
Gets layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (*Output*) Pointer to write the hyperparameters to.

virtual void **getHyperparameterGradients**(float **gradient*) **override**
Gets the gradients of the layer's output wrt to is hyperparameters (e.g., weights and biases)

Parameters **gradient** – (*Output*) Pointer to write the hyperparameter gradients to.

virtual void **backwardData**(**const** size_t *t*) **override**
Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

virtual void **backwardHyperparameters**(**const** size_t *t*) **override**
Calculates the gradients of layer hyperparameters.

Parameters **t** – Indicates the current timestep

Public Members

ssize_t **_imageHeight**
Height of the incoming 2D image.

ssize_t **_imageWidth**
Width of the incoming 2D image.

ssize_t **_kernelHeight**
Height of the incoming 2D image.

ssize_t **_kernelWidth**
Width of the incoming 2D image.

ssize_t **_verticalStride**
: Strides for the image on the vertical dimension.

ssize_t **_horizontalStride**
: Strides for the image on the horizontal dimension.

ssize_t **_paddingLeft**
: Paddings for the image left side.

ssize_t **_paddingRight**
: Paddings for the image right side.

ssize_t **_paddingTop**
: Paddings for the image top side.

ssize_t **_paddingBottom**
: Paddings for the image Bottom side.

ssize_t **N**
: Pre-calculated value for Mini-Batch Size.

ssize_t **IC**
: Pre-calculated value for *Input* Channels.

ssize_t **IH**
: Pre-calculated value for *Input* Image Height.

ssize_t **IW**
: Pre-calculated value for *Input* Image Width.

ssize_t **OC**
: Pre-calculated value for *Output* Channels.

ssize_t **OH**
: Pre-calculated value for *Output* Image Height.

ssize_t **OW**
: Pre-calculated value for *Output* Image Width.

ssize_t **KH**
: Pre-calculated value for Kernel Image Height.

ssize_t **KW**
: Pre-calculated value for Kernel Image Width.

ssize_t **PL**
: Pre-calculated values for padding left.

ssize_t **PR**
: Pre-calculated values for padding right.

ssize_t **PT**
: Pre-calculated values for padding top.

ssize_t **PB**
: Pre-calculated values for padding bottom.

ssize_t **SH**
: Pre-calculated values for horizontal stride.

ssize_t **SV**
: Pre-calculated values for vertical stride.

class *korali*::*solver*::**DeepSupervisor** : **public** *korali*::*Solver*

#include <deepSupervisor.hpp> Class declaration for module: *DeepSupervisor*.

Public Functions

virtual bool **checkTermination**() **override**

Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**

Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**

Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**

Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**

Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**

Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

std::vector<*std*::vector<float>> &**getEvaluation**(**const** *std*::vector<*std*::vector<*std*::vector<float>>> &*input*)

Evaluates a neural network on a batch of sequential vectors.

Parameters **input** – Batch of seuential input data.

Returns Evaluation of batch of sequential data.

std::vector<float> **getHyperparameters**()

Returns the current hyperparameter of the neural network.

Returns The hyperparameter.

virtual void **initialize**() **override**

Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **runGeneration**() **override**

Runs the current generation.

virtual void **printGenerationAfter**() **override**

Prints solver information after the execution of the current generation.

void **runTrainingGeneration**()
Runs training generation.

void **runTestingGeneration**()
Runs testing generation.

std::vector<float> **backwardGradients**(**const** *std*::vector<*std*::vector<float>> &*gradients*)
Run the backward pipeline of the network given the gradient of the loss and return the gradient.

Parameters **gradients** – A vector containing the gradient of the loss with respect to the output of the network

Returns A vector containing the gradient of the loss with respect to the weights of the network

void **runTrainingOnWorker**(*korali*::*Sample* &*sample*)
Run the training pipeline of the network given an input and return the output.

Parameters **sample** – A sample containing the NN's input BxTxIC (B: Batch Size, T: Time steps, IC: Input channels) and solution BxOC data (B: Batch Size, OC: Output channels)

void **runEvaluationOnWorker**(*korali*::*Sample* &*sample*)
Run the forward evaluation pipeline of the network given an input and return the output.

Parameters **sample** – A sample containing the NN's input BxTxIC (B: Batch Size, T: Time steps, IC: Input channels)

void **updateHyperparametersOnWorker**(*korali*::*Sample* &*sample*)
Update the hyperparameters for the neural network after an update for every worker.

Parameters **sample** – A sample containing the new NN's hyperparameters

Public Members

std::string **_mode**
Specifies the operation mode for the learner.

knlohmann::json **_neuralNetworkHiddenLayers**
Sets the configuration of the hidden layers for the neural network.

knlohmann::json **_neuralNetworkOutputActivation**
Allows setting an aditional activation for the output layer.

knlohmann::json **_neuralNetworkOutputLayer**
Sets any additional configuration (e.g., masks) for the output NN layer.

std::string **_neuralNetworkEngine**
Specifies which Neural Network backend engine to use.

std::string **_neuralNetworkOptimizer**
Determines which optimizer algorithm to use to apply the gradients on the neural network's hyperparameters.

std::string **_lossFunction**
Function to calculate the difference (loss) between the NN inference and the exact solution and its gradients for optimization.

float **_learningRate**
Learning rate for the underlying ADAM optimizer.

int **_l2RegularizationEnabled**
Regulates if l2 regularization will be applied to the neural network.

int **_l2RegularizationImportance**
Importance weight of l2 regularization.

float **_outputWeightsScaling**
Specified by how much will the weights of the last linear transformation of the NN be scaled. A value of < 1.0 is useful for a more deterministic start.

size_t **_batchConcurrency**
Specifies in how many parts will the mini batch be split for concurrent processing. It must divide the training mini batch size perfectly.

std::vector<*std*::vector<float>> **_evaluation**
[Internal Use] The output of the neural network if running on testing mode.

float **_currentLoss**
[Internal Use] Current value of the loss function.

std::vector<float> **_lossHistory**
[Internal Use] Current value of the loss function.

std::vector<float> **_normalizationMeans**
[Internal Use] Stores the current neural network normalization mean parameters.

std::vector<float> **_normalizationVariances**
[Internal Use] Stores the current neural network normalization variance parameters.

korali::*fGradientBasedOptimizer* ***_optimizer**
[Internal Use] Stores a pointer to the optimizer.

float **_targetLoss**
[Termination Criteria] Specifies the maximum number of suboptimal generations.

problem::*SupervisedLearning* ***_problem**
Korali Problem for optimizing NN weights and biases.

korali::*Experiment* **_optExperiment**
Korali Experiment for optimizing the NN's weights and biases.

NeuralNetwork ***_neuralNetwork**
A neural network to be trained based on inputs and solutions.

class *korali*::*problem*::**Design** : **public** *korali*::*Problem*
#include <design.hpp> Class declaration for module: *Design*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

void **runModel**(*Sample* &*sample*)
Evaluates the model for a given sample from the prior distribution.

Parameters **sample** – A *Korali Sample*

Public Members

std::uint64_t **_model**
Stores the model function.

size_t **_parameterVectorSize**
[Internal Use] Stores the dimension of the parameter space.

size_t **_designVectorSize**
[Internal Use] Stores the dimension of the design space.

size_t **_measurementVectorSize**
[Internal Use] Stores the dimension of the design space.

std::vector<size_t> **_parameterVectorIndexes**
[Internal Use] Stores the indexes of the variables that constitute the parameter vector.

std::vector<size_t> **_designVectorIndexes**
[Internal Use] Stores the indexes of the variables that constitute the design vector.

std::vector<size_t> **_measurementVectorIndexes**
[Internal Use] Stores the indexes of the variables that constitute the design vector.

class *korali*::*solver*::**Designer** : **public** *korali*::*Solver*
#include <designer.hpp> Class declaration for module: *Designer*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

void **evaluateDesign**(*Sample* &*sample*)
Evaluates the utility function for a given design.

Parameters **sample** – A *Korali Sample*

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

virtual void **runGeneration**() **override**
Runs the current generation.

virtual void **printGenerationBefore**() **override**
Prints solver information before the execution of the current generation.

virtual void **printGenerationAfter**() **override**
Prints solver information after the execution of the current generation.

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

Public Members

size_t **_executionsPerGeneration**
Specifies the number of model executions per generation. By default this setting is 0, meaning that all executions will be performed in the first generation. For values greater 0, executions will be split into batches and split int generations for intermediate output.

double **_sigma**
Standard deviation for measurement.

std::vector<*std*::vector<double>> **_priorSamples**
[Internal Use] The samples of the prior distribution.

std::vector<*std*::vector<*std*::vector<double>>> **_modelEvaluations**
[Internal Use] Evaluations of the samples of the prior distribution.

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**
[Internal Use] Gaussian random number generator.

std::vector<double> **_parameterLowerBounds**
[Internal Use] The lower bound of the parameters.

std::vector<double> **_parameterUpperBounds**
[Internal Use] The upper bound of the parameters.

std::vector<double> **_parameterExtent**
[Internal Use] The extent of the domain of the parameters (for grid-based evaluation).

std::vector<size_t> **_numberOfParameterSamples**
[Internal Use] The number of samples per direction.

std::vector<int> **_parameterDistributionIndex**
[Internal Use] The distribution of parameters (for monte-carlo evaluation).

std::vector<double> **_parameterGridSpacing**
[Internal Use] The grid spacing of the parameters (for grid-based evaluation).

std::vector<size_t> **_parameterHelperIndices**
[Internal Use] Holds helper to calculate cartesian indices from linear index (for grid-based evaluation).

std::string **_parameterIntegrator**
[Internal Use] The integrator that is used for the parameter-integral.

std::vector<double> **_designLowerBounds**
[Internal Use] The lower bound of the designs.

std::vector<double> **_designUpperBounds**
[Internal Use] The upper bound of the designs.

std::vector<double> **_designExtent**
[Internal Use] The extent of the design space.

std::vector<size_t> **_numberOfDesignSamples**
[Internal Use] The number of samples per direction.

std::vector<double> **_designGridSpacing**
[Internal Use] The grid spacing of the designs (for grid-based evaluation).

std::vector<size_t> **_designHelperIndices**
[Internal Use] Holds helper to calculate cartesian indices from linear index (for grid-based evaluation).

std::vector<*std*::vector<double>> **_designCandidates**
[Internal Use] Holds candidate designs.

std::vector<size_t> **_numberOfMeasurementSamples**
[Internal Use] The number of samples per direction.

size_t **_numberOfPriorSamples**
[Internal Use] Specifies the number of samples drawn from the prior distribution.

size_t **_numberOfLikelihoodSamples**
[Internal Use] Specifies the number of samples drawn from the likelihood.

size_t **_numberOfDesigns**
[Internal Use] Specifies the number of design parameters (for grid-based evaluation).

size_t **_optimalDesignIndex**
[Internal Use] Index of the optimal design.

std::vector<double> **_utility**
[Internal Use] Evaluation of utility.

std::vector<*Sample*> **_samples**
Container for samples to be evaluated per generation.

problem::*Design* ***_problem**
Problem pointer.

class *korali*::*solver*::*agent*::**Discrete** : **public** *korali*::*solver*::*Agent*
#include <discrete.hpp> Class declaration for module: *Discrete*.

Subclassed by *korali::solver::agent::discrete::dVRACER*

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual float **calculateImportanceWeight**(**const** *std*::vector<float> &*action*, **const** *policy_t* &*curPolicy*, **const** *policy_t* &*oldPolicy*) **override**
Calculates importance weight of current action from old and new policies.

Parameters

- **action** – The action taken
- **curPolicy** – The current policy
- **oldPolicy** – The old policy, the one used for take the action in the first place

Returns The importance weight

std::vector<float> **calculateImportanceWeightGradient**(**const** *policy_t* &*curPolicy*, **const** *policy_t* &*oldPolicy*)
Calculates the gradient of importance weight wrt to NN output.

Parameters

- **curPolicy** – current policy object
- **oldPolicy** – old policy object from RM

Returns gradient of importance weight wrt NN output (q_i's and inverse temperature)

std::vector<float> **calculateKLDivergenceGradient**(**const** *policy_t* &*oldPolicy*, **const** *policy_t* &*curPolicy*)
Calculates the gradient of KL(p_old, p_cur) wrt to the NN output.

Parameters

- **oldPolicy** – current policy object
- **curPolicy** – old policy object from RM

Returns gradient of KL wrt curent distribution parameter (q_i's and inverse temperature)

virtual void **getAction**(*korali*::*Sample* &*sample*) **override**
Gathers the next action either from the policy or randomly.

Parameters **sample** – *Sample* on which the action and metadata will be stored

virtual void **initializeAgent**() **override**
Initializes the internal state of the policy.

Public Members

problem::*reinforcementLearning*::*Discrete* ***_problem**
Storage for the pointer to the (discrete) learning problem.

class *korali*::*problem*::*reinforcementLearning*::**Discrete** : **public** *korali*::*problem*::*ReinforcementLearning*
#include <discrete.hpp> Class declaration for module: *Discrete*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

Public Members

std::vector<*std*::vector<float>> **_possibleActions**
The set of all possible actions.

class *korali*::*conduit*::**Distributed** : **public** *korali*::*Conduit*
#include <distributed.hpp> Class declaration for module: *Distributed*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initServer**() **override**
Initializes the worker/server bifurcation in the conduit.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **terminateServer**() **override**
Finalizes the workers.

virtual void **stackEngine**(*Engine* **engine*) **override**
Stacks a new *Engine* into the engine stack.

Parameters **engine** – A *Korali Engine*

virtual void **popEngine**() **override**
Pops the current *Engine* from the engine stack.

virtual void **listenWorkers**() **override**
(*Engine* <- Worker) Receives all pending incoming messages and stores them into the corresponding sample's message queue.

virtual void **broadcastMessageToWorkers**(knlohmann::json &*message*) **override**
(*Engine* -> Worker) Broadcasts a message to all workers

Parameters **message** – JSON object with information to broadcast

virtual void **sendMessageToEngine**(knlohmann::json &*message*) **override**
(*Sample* -> *Engine*) Sends an update to the engine to provide partial information while the sample is still active

Parameters **message** – Message to send to engine

virtual knlohmann::json **recvMessageFromEngine**() **override**
(*Sample* <- *Engine*) Blocking call that waits until any message incoming from the engine.

Returns message from the engine.

virtual void **sendMessageToSample**(*Sample* &*sample*, knlohmann::json &*message*) **override**
(*Engine* -> *Sample*) Sends an update to a still active sample

Parameters

- **sample** – The sample from which to receive an update
- **message** – Message to send to the sample.

virtual size_t **getProcessId**() **const override**
Returns the identifier corresponding to the executing process (to differentiate their random seeds)

Returns The executing process id

int **getRootRank**() **const**
Determines which rank is the root.

Returns The rank id of the root rank.

virtual bool **isRoot**() **const override**
Determines whether the caller rank/thread/process is root.

Returns True, if it is root; false, otherwise.

virtual bool **isWorkerLeadRank**() **const override**
Determines whether the caller rank is the leader of its worker root.

Returns True, if it is the worker leader rank; false, otherwise.

virtual size_t **getWorkerCount**() **const override**
Get total *Korali* worker count in the conduit.

Returns The number of workers

Public Members

int **_ranksPerWorker**
Specifies the number of MPI ranks per *Korali* worker.

int **_engineRanks**
Specifies the number of MPI ranks for the *Korali* engine.

int **_rankId**
ID of the current rank.

int **_rankCount**
Total number of ranks in execution.

int **_workerCount**
Number of *Korali* Teams in execution.

int **_workerIdSet**
Signals whether the worker has been assigned a team.

int **_localRankId**
Local ID the rank within its *Korali* Worker.

std::vector<*std*::vector<int>> **_workerTeams**
Storage that contains the rank teams for each worker.

std::vector<int> **_rankToWorkerMap**
Map that indicates to which worker does the current rank correspond to.

class *korali*::**Distribution** : **public** *korali*::*Module*

#include <distribution.hpp> Class declaration for module: *Distribution*.

Subclassed by *korali::distribution::Multivariate*, *korali::distribution::Specific*, *korali::distribution::Univariate*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**

Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**

Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**

Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**

Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

gsl_rng ***setRange**(**const** *std*::string *rangeString*)

Creates and sets the RNG range (state and seed) of the random distribution.

Parameters **rangeString** – The range to load, in string of hexadecimal values form

Returns Pointer to the new range.

std::string **getRange**(gsl_rng **range*) **const**

Gets a hexadecimal string from a given range's state and seed.

Parameters **range** – Range to read from

Returns Hexadecimal string produced.

inline **virtual** void **updateDistribution**()

Updates the parameters of the distribution based on conditional variables.

inline **virtual** double ***getPropertyPointer**(**const** *std*::string &*property*)

Gets the pointer to a distribution property.

Parameters **property** – The name of the property to update

Returns Pointer to the property

Public Members

std::string **_name**

Defines the name of the distribution.

size_t **_randomSeed**

Defines the random seed of the distribution.

gsl_rng ***_range**

Stores the current state of the distribution in hexadecimal notation.

std::map<*std*::string, double*> **_conditionalsMap**

Map to store the link between parameter names and their pointers.

double **_aux**
Auxiliar variable to hold pre-calculated data to avoid re-processing information.

bool **_hasConditionalVariables**
Indicates whether or not this distribution contains conditional variables.

class *korali*::*solver*::*agent*::*discrete*::**dVRACER** : **public** *korali*::*solver*::*agent*::*Discrete*
#include <dVRACER.hpp> Class declaration for module: *dVRACER*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **updateVtbc**(size_t *expId*)
Update the V-target or current and previous experiences in the episode.

Parameters **expId** – Current Experience Id

void **calculatePolicyGradients**(**const** *std*::vector<*std*::pair<size_t, size_t>> &*miniBatch*, **const** size_t *policyIdx*)
Calculates the gradients for the policy/critic neural network.

Parameters

- **miniBatch** – The indexes of the experience mini batch
- **policyIdx** – The indexes of the policy to compute the gradient for

std::vector<*policy_t*> **getPolicyInfo**(**const** *std*::vector<*std*::pair<size_t, size_t>> &*miniBatch*) **const**
Retreives the policy infos for the samples in the minibatch.

Parameters **miniBatch** – The indexes of the experience mini batch

Returns A vector containing the policy infos in the order they are given in the miniBatch

virtual float **calculateStateValue**(**const** *std*::vector<*std*::vector<float>> &*stateSequence*, size_t *policyIdx* = 0) **override**
Function to pass a state time series through the NN and calculates the action probabilities, along with any additional information.

Parameters

- **stateSequence** – The batch of state time series (Format: BxTxS, B is batch size, T is the time series lenght, and S is the state size)
- **policyIdx** – The index for the policy for which the state-value is computed

Returns A JSON object containing the information produced by the policies given the current state series

virtual void **runPolicy**(**const** *std*::vector<*std*::vector<*std*::vector<float>>> &*stateSequenceBatch*, *std*::vector<*policy_t*> &*policy*, size_t *policyIdx* = 0) **override**

Function to pass a state time series through the NN and calculates the action probabilities, along with any additional information.

Parameters

- **stateSequenceBatch** – The batch of state time series (Format: BxTxS, B is batch size, T is the time series lenght, and S is the state size)
- **policy** – Vector with policy objects that is filled after forwarding the policy
- **policyIdx** – The index for the policy for which the state-value is computed

virtual knlohmann::json **getPolicy**() **override**

Obtains the policy hyperaparamters from the learner for the agent to generate new actions.

Returns The current policy hyperparameters

virtual void **setPolicy**(**const** knlohmann::json &*hyperparameters*) **override**

Updates the agent's hyperparameters.

Parameters **hyperparameters** – The hyperparameters to update the agent.

virtual void **trainPolicy**() **override**

Trains the policy, based on the new experiences.

virtual void **printInformation**() **override**

Prints information about the training policy.

virtual void **initializeAgent**() **override**

Initializes the internal state of the policy.

Public Members

float **_initialInverseTemperature**

Initial inverse temperature of the softmax distribution. Large values lead to a distribution that is more concentrated around the action with highes Q-value estimate.

float **_statisticsAverageInverseTemperature**

[Internal Use] Measure of unlikeability for categorial data, approaches 1.0 for uniform behavior and 0. for deterministic case.

float **_statisticsAverageActionUnlikeability**

[Internal Use] Measure of unlikeability for categorial data, approaches 1.0 for uniform behavior and 0. for deterministic case.

struct *korali*::*solver*::*sampler*::**ellipse_t**

#include <Nested.hpp> Ellipse object to generate bounds.

Public Functions

ellipse_t() = delete
Default c-tor (avoid empty initialization).

inline ellipse_t(size_t *dim*)
Init d-dimensional ellipse without covariance.

Parameters **dim** – Dimension of ellipsoid.

void **initSphere**()
Init d-dimensional unit sphere.

void **scaleVolume**(double *factor*)
Scale volume.

Parameters **factor** – Volume multiplicator.

Public Members

size_t **dim**
Dimension of ellipsoid.

size_t **num**
Number samples in ellipse.

double **det**
Determinant of covariance.

std::vector<size_t> **sampleIdx**
Indices of samples from live data set.

std::vector<double> **mean**
Mean vector of samples in ellipse.

std::vector<double> **cov**
Covariance Matrix of samples in ellipse.

std::vector<double> **invCov**
Inverse of Covariance Matrix.

std::vector<double> **axes**
Axes of the ellipse.

std::vector<double> **evals**
Eigenvalues of the ellipse.

std::vector<double> **paxes**
Principal axes of the ellipse.

double **volume**
Volume estimated from covariance.

double **pointVolume**
'True' volume from which the subset of samples were sampled from.

class *korali*::**Engine**
#include <engine.hpp> A *Korali Engine* initializes the conduit and experiments, and guides their execution.

Public Functions

Engine()

void **saveProfilingInfo**(**const** bool *forceSave* = false)
Saves the profiling information to the specified path.

Parameters **forceSave** – Saves even if the current generation does not divide _profilingFrequency. Reserved for last generation.

void **initializeExperiments**()
Initializer for the engine's experiments.

void **run**(*std*::vector<*Experiment*> &*experiments*)
Stores a set experiments into the experiment list and runs them to completion.

Parameters **experiments** – Set of experiments.

void **run**(*Experiment* &*experiment*)
Stores a single experiment into the experiment list and runs it to completion.

Parameters **experiment** – The experiment to run.

void **start**()
Runs the stored list of experiments.

knlohmann::json &**operator[]**(**const** *std*::string &*key*)
C++ wrapper for the getItem operator.

Parameters **key** – A C++ string acting as JSON key.

Returns The referenced JSON object content.

knlohmann::json &**operator[]**(**const** unsigned long int &*key*)
C++ wrapper for the getItem operator.

Parameters **key** – A C++ integer acting as JSON key.

Returns The referenced JSON object content.

pybind11::object **getItem**(**const** pybind11::object *key*)
Gets an item from the JSON object at the current pointer position.

Parameters **key** – A pybind11 object acting as JSON key (number or string).

Returns A pybind11 object

void **setItem**(**const** pybind11::object *key*, **const** pybind11::object *val*)
Sets an item on the JSON object at the current pointer position.

Parameters

- **key** – A pybind11 object acting as JSON key (number or string).
- **val** – The value of the item to set.

void **serialize**(knlohmann::json &*js*)
Serializes *Engine*'s data into a JSON object.

Parameters **js** – Json object onto which to store the *Engine* data.

Public Members

Conduit *__conduit**
A pointer to the execution conduit. Shared among all experiments in the engine.

std::string __verbosityLevel**
Verbosity level of the *Engine* ('Silent', 'Minimal' (default), 'Normal' or 'Detailed').

std::vector<*Experiment**> __experimentVector**
Stores the list of experiments to run.

cothread_t __thread**
Stores the main execution thread (coroutine).

std::string __profilingPath**
Saves the output path for the profiling information file.

std::string __profilingDetail**
Specifies how much detail will be saved in the profiling file (None, Full)

double __profilingFrequency**
Specifies every how many generation will the profiling file be updated.

std::chrono::time_point<*std*::chrono::high_resolution_clock> __profilingLastSave**
Stores the timepoint of the last time the profiling information was saved.

KoraliJson __js**
Stores the JSON based configuration for the engine.

bool __isDryRun**
Determines whether this is a dry run (no conduit initialization nor execution)

Experiment *__currentExperiment**
(Worker) Stores a pointer to the current *Experiment* being processed

Public Static Functions

static *Engine* ***deserialize**(**const** knlohmann::json &*js*)
Deserializes JSON object and returns a *Korali Engine*.

Parameters **js** – Json object onto which to store the *Engine* data.

Returns The *Korali Engine*

class *korali*::*solver*::**Executor** : **public** *korali*::*Solver*
#include <executor.hpp> Class declaration for module: *Executor*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **runGeneration**() **override**
Generate a sample and evaluate it.

virtual void **printGenerationBefore**() **override**
Console Output before generation runs.

virtual void **printGenerationAfter**() **override**
Console output after generation.

Public Members

size_t **_executionsPerGeneration**
Specifies the number of model executions per generation. By default this setting is 0, meaning that all executions will be performed in the first generation. For values greater 0, executions will be split into batches and split int generations for intermediate output.

size_t **_sampleCount**
[Internal Use] Number of samples to execute.

class *korali*::**Experiment** : **public** *korali*::*Module*
#include <experiment.hpp> Class declaration for module: *Experiment*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

Experiment()

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

pybind11::object **getItem**(**const** pybind11::object *key*)
Gets an item from the JSON object at the current pointer position.

Parameters **key** – A pybind11 object acting as JSON key (number or string).

Returns A pybind11 object

void **setItem**(**const** pybind11::object *key*, **const** pybind11::object *val*)
Sets an item on the JSON object at the current pointer position.

Parameters

- **key** – A pybind11 object acting as JSON key (number or string).
- **val** – The value of the item to set.

bool **loadState**(**const** *std*::string &*path*)
Load the state of an experiment from a *Korali* result file.

Parameters **path** – Path from which to load the experiment state.

Returns true, if file was found; false, otherwise

void **saveState**()
Saves the state into the experiment's result path.

void **run**()
Start the execution of the current experiment.

knlohmann::json &**operator[]**(**const** *std*::string &*key*)
C++ wrapper for the getItem operator.

Parameters **key** – A C++ string acting as JSON key.

Returns The referenced JSON object content.

void **setSeed**(knlohmann::json &*js*)
Initializes seed to a random value based on current time if not set by the user (i.e. Random Seed is 0).

Parameters **js** – Json object onto which to store the *Experiment* data.

Public Members

size_t **_randomSeed**
Specifies the initializing seed for the generation of random numbers. If 0 is specified, *Korali* will automatically derivate a new seed base on the current time.

int **_preserveRandomNumberGeneratorStates**
Indicates that the engine must preserve the state of their RNGs for reproducibility purposes.

std::vector<*korali*::*distribution*::*Univariate**> **_distributions**
Represents the distributions to use during execution.

std::vector<*korali*::Variable*> **_variables**
Sample coordinate information.

korali::*Problem* ***_problem**
Represents the configuration of the problem to solve.

korali::*Solver* ***_solver**
Represents the state and configuration of the solver algorithm.

std::string **_fileOutputPath**
Specifies the path of the results directory.

int **_fileOutputUseMultipleFiles**
If true, *Korali* stores a different generation file per generation with incremental numbering. If disabled, *Korali* stores the latest generation files into a single file, overwriting previous results.

int **_fileOutputEnabled**
Specifies whether the partial results should be saved to the results directory.

size_t **_fileOutputFrequency**
Specifies how often (in generations) will partial result files be saved on the results directory. The default, 1, indicates that every generation's results will be saved. 0 indicates that only the latest is saved.

int **_storeSampleInformation**
Specifies whether the sample information should be saved to samples.json in the results path.

std::string **_consoleOutputVerbosity**
Specifies how much information will be displayed on console when running *Korali*.

size_t **_consoleOutputFrequency**
Specifies how often (in generations) will partial results be printed on console. The default, 1, indicates that every generation's results will be printed.

size_t **_currentGeneration**
[Internal Use] Indicates the current generation in execution.

int **_isFinished**
[Internal Use] Indicates whether execution has reached a termination criterion.

size_t **_runID**
[Internal Use] Specifies the *Korali* run's unique identifier. Used to distinguish run results when two or more use the same output directory.

std::string **_timestamp**
[Internal Use] Indicates the current time when saving a result file.

KoraliJson **_js**
JSON object to store the experiment's configuration.

Logger ***_logger**
A pointer to the *Experiment*'s logger object.

Engine ***_engine**
A pointer to the parent engine.

KoraliJson **_sampleInfo**
JSON object to details of all the samples that have been executed, if requested by the user.

size_t **_experimentId**
Experiment Identifier.

cothread_t **_thread**
Experiment's coroutine (thread). It is swapped among other experiments, and sample threads.

bool **_isInitialized**
Flag to indicate that the experiment has been initialized to prevent it from re-initializing upon resuming.

double **_resultSavingTime**
[Profiling] Measures the amount of time taken by saving results

bool **_overrideEngine** = false
For testing purposes, this field establishes whether the engine is the one to run samples (default = false) or a custom function (true)

std::function<void (*Sample*&) > **_overrideFunction**
For testing purposes, this field establishes which custom function to use to override the engine on sample execution for testing.

class *korali*::*distribution*::*univariate*::**Exponential** : **public** *korali*::*distribution*::*Univariate*
#include <exponential.hpp> Class declaration for module: *Exponential*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_location**
[Conditional Variable Value] Shift for the exponential distribution.

std::string **_locationConditional**
[Conditional Variable Reference] Shift for the exponential distribution.

double **_mean**
[Conditional Variable Value] Mean and standard deviation of the (unshifted) exponential distribution.

std::string **_meanConditional**
[Conditional Variable Reference] Mean and standard deviation of the (unshifted) exponential distribution.

class *korali*::**fAdaBelief** : **public** *korali*::*fGradientBasedOptimizer*
#include <fAdaBelief.hpp> Class declaration for module: *fAdaBelief*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **processResult**(*std*::vector<float> &*gradient*) **override**
Takes a sample evaluation and its gradient and calculates the next set of parameters.

Parameters **gradient** – The gradient of the objective function at the current set of parameters

virtual void **reset**() **override**
Restores the optimizer to the initial state.

virtual void **printInternals**() **override**
Prints internals of solver.

Public Members

float **_beta1**
Term to guard agains numerical instability.

float **_beta2**
Term to guard agains numerical instability.

float **_beta1Pow**
[Internal Use] First running powers of beta_1^t.

float **_beta2Pow**
[Internal Use] Second running powers of beta_2^t.

std::vector<double> **_firstMoment**
[Internal Use] First moment of Gradient.

std::vector<double> **_secondCentralMoment**
[Internal Use] Second central moment.

class *korali*::**fAdaGrad** : **public** *korali*::*fGradientBasedOptimizer*
#include <fAdaGrad.hpp> Class declaration for module: *fAdaGrad*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **processResult**(*std*::vector<float> &*gradient*) **override**
Takes a sample evaluation and its gradient and calculates the next set of parameters.

Parameters **gradient** – The gradient of the objective function at the current set of parameters

virtual void **reset**() **override**
Restores the optimizer to the initial state.

virtual void **printInternals**() **override**
Prints internals of solver.

Public Members

std::vector<float> **_gdiag**
[Internal Use] Digaonal sum of the outer products of the gradients diag(gg^T)

class *korali*::**fAdam** : **public** *korali*::*fGradientBasedOptimizer*
#include <fAdam.hpp> Class declaration for module: *fAdam*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **processResult**(*std*::vector<float> &*gradient*) **override**
Takes a sample evaluation and its gradient and calculates the next set of parameters.

Parameters **gradient** – The gradient of the objective function at the current set of parameters

virtual void **reset**() **override**
Restores the optimizer to the initial state.

virtual void **printInternals**() **override**
Prints internals of solver.

Public Members

float **_beta1**
Term to guard agains numerical instability.

float **_beta2**
Term to guard agains numerical instability.

float **_beta1Pow**
[Internal Use] First running powers of beta_1^t.

float **_beta2Pow**
[Internal Use] Second running powers of beta_2^t.

std::vector<double> **_firstMoment**
[Internal Use] First moment of Gradient.

std::vector<double> **_secondMoment**
[Internal Use] Second moment of Gradient.

class *korali*::**fGradientBasedOptimizer** : **public** *korali*::*Module*
#include <fGradientBasedOptimizer.hpp> Class declaration for module: *fGradientBasedOptimizer*.

Subclassed by *korali::fAdaBelief*, *korali::fAdaGrad*, *korali::fAdam*, *korali::fMadGrad*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **processResult**(*std*::vector<float> &*gradient*) = 0
Takes a sample evaluation and its gradient and calculates the next set of parameters.

Parameters **gradient** – The gradient of the objective function at the current set of parameters

virtual void **preProcessResult**(*std*::vector<float> &*gradient*)
Checks size and values of gradient.

Parameters **gradient** – Gradient values to check

virtual void **postProcessResult**(*std*::vector<float> &*parameters*)
Checks the result of the gradient update.

Parameters **parameters** – Parameter values to check

virtual void **printInternals**() = 0
Prints internals of solver.

virtual void **reset**() = 0
Restores the optimizer to the initial state.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

Public Members

float **_epsilon**
Term to guard agains numerical instability.

size_t **_nVars**
Size of variable space size(x) of f(x)

float **_eta**
Step size/learning rate for current iterration.

std::vector<float> **_currentValue**
[Internal Use] Holds current values of the parameters.

class *korali*::**fMadGrad** : **public** *korali*::*fGradientBasedOptimizer*
#include <fMadGrad.hpp> Class declaration for module: *fMadGrad*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **processResult**(*std*::vector<float> &*gradient*) **override**
Takes a sample evaluation and its gradient and calculates the next set of parameters.

Parameters **gradient** – The gradient of the objective function at the current set of parameters

virtual void **reset**() **override**
Restores the optimizer to the initial state.

virtual void **printInternals**() **override**
Prints internals of solver.

Public Members

std::vector<float> **_initialValue**
[Internal Use] Intitial value x0, currently set to 0.

std::vector<float> **_s**
[Internal Use] Scaled gradient sum.

std::vector<float> **_v**
[Internal Use] Scaled digaonal sum of the outer products of the gradients diag(gg^T).

std::vector<float> **_z**
[Internal Use] Update rule.

float **_momentum**
[Internal Use] Momentum to be used.

struct *korali*::*solver*::*sampler*::**fparam_s**
#include <TMCMC.hpp> Struct for *TMCMC* optimization operations.

Public Members

const double ***loglike**
Likelihood values in current generation.

size_t **Ns**
Population size of current generation.

double **exponent**
Annealing exponent of current generation.

double **cov**
Target coefficient of variation.

class *korali*::*distribution*::*univariate*::**Gamma** : **public** *korali*::*distribution*::*Univariate*
#include <gamma.hpp> Class declaration for module: *Gamma*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_shape**
[Conditional Variable Value] The shape parameter of the *Gamma* distribution, it controls the mean and skewness.

std::string **_shapeConditional**
[Conditional Variable Reference] The shape parameter of the *Gamma* distribution, it controls the mean and skewness.

double **_scale**
[Conditional Variable Value] The scale parameter of the *Gamma* distribution, it controls the mean.

std::string **_scaleConditional**
[Conditional Variable Reference] The scale parameter of the *Gamma* distribution, it controls the mean.

class *korali*::*distribution*::*univariate*::**Geometric** : **public** *korali*::*distribution*::*Univariate*
#include <geometric.hpp> Class declaration for module: *Geometric*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_successProbability**
[Conditional Variable Value] Probability of success of an individual trial.

std::string **_successProbabilityConditional**
[Conditional Variable Reference] Probability of success of an individual trial.

class *korali*::*solver*::*optimizer*::**GridSearch** : **public** *korali*::*solver*::*Optimizer*
#include <gridSearch.hpp> Class declaration for module: *GridSearch*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

virtual void **runGeneration**() **override**
Runs the current generation.

virtual void **printGenerationBefore**() **override**
Prints solver information before the execution of the current generation.

virtual void **printGenerationAfter**() **override**
Prints solver information after the execution of the current generation.

Public Members

size_t **_numberOfValues**
[Internal Use] Total number of parameter to evaluate (samples per generation).

std::vector<double> **_objective**
[Internal Use] Vector containing values of the objective function.

std::vector<size_t> **_indexHelper**
[Internal Use] Holds helper to calculate cartesian indices from linear index.

class *korali*::*neuralNetwork*::*layer*::*recurrent*::**GRU** : **public** *korali*::*neuralNetwork*::*layer*::*Recurrent*
#include <gru.hpp> Class declaration for module: *GRU*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **createForwardPipeline**() **override**
Initializes the layer's internal memory structures for the forward pipeline.

virtual void **createBackwardPipeline**() **override**
Initializes the internal memory structures for the backward pipeline.

virtual void **forwardData**(**const** size_t *t*) **override**
Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

virtual void **backwardData**(**const** size_t *t*) **override**
Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

class *korali*::*solver*::*sampler*::**Hamiltonian**
#include <hamiltonian_base.hpp> Abstract base class for *Hamiltonian* objects.

Subclassed by *korali::solver::sampler::HamiltonianEuclidean*, *korali::solver::sampler::HamiltonianRiemannian*

Public Functions

Hamiltonian() = default
Default constructor.

inline Hamiltonian(**const** size_t *stateSpaceDim*, *korali*::*Experiment* *k*)
Constructor with State Space Dim.

Parameters

- **stateSpaceDim** – Dimension of State Space.
- **k** – Pointer to *Korali* object.

virtual ~Hamiltonian() = default
Destructor of abstract base class.

virtual double **H**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) = 0
Purely abstract total energy function used for *Hamiltonian* Dynamics.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverse of metric.

Returns Total energy.

virtual double **K**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) = 0
Purely virtual kinetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverse of metric.

Returns Kinetic energy.

virtual *std*::vector<double> **dK**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) = 0
Purely virtual gradient of kintetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverse metric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual double **U**()
Potential Energy function.

Returns Potential energy.

inline virtual *std*::vector<double> **dU**()
Gradient of Potential Energy function.

Returns Gradient of Potential energy.

virtual double **tau**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) = 0
Purely virtual function tau(q, p) = 0.5 * momentum^T * inverseMetric(q) * momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

virtual *std*::vector<double> **dtau_dq**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) = 0

Purely virtual gradient of dtau_dq(q, p) wrt. position.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

virtual *std*::vector<double> **dtau_dp**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) = 0

Purely virtual gradient of dtau_dp(q, p) wrt. momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

virtual double **phi**() = 0

Purely virtual gradient of kinetic energy.

Returns Gradient of kinetic energy.

virtual *std*::vector<double> **dphi_dq**() = 0

Purely virtual gradient of kinetic energy.

Returns Gradient of kinetic energy.

virtual double **innerProduct**(**const** *std*::vector<double> &*leftMomentum*, **const** *std*::vector<double> &*rightMomentum*, **const** *std*::vector<double> &*inverseMetric*) **const** = 0

Purely virtual, calculates inner product induces by inverse metric.

Parameters

- **leftMomentum** – Left vector of inner product.
- **rightMomentum** – Right vector of inner product.
- **inverseMetric** – Inverse of current metric.

Returns inner product

inline virtual void **updateHamiltonian**(**const** *std*::vector<double> &*position*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*)

Updates current position of hamiltonian.

Parameters

- **position** – Current position.
- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

virtual *std*::vector<double> **sampleMomentum**(**const** *std*::vector<double> &*metric*) **const** = 0
Purely virtual function to generates momentum vector.

Parameters **metric** – Current metric.

Returns Momentum sampled from normal distribution with metric as covariance matrix.

inline bool **computeStandardCriterion**(**const** *std*::vector<double> &*positionLeft*, **const** *std*::vector<double> &*momentumLeft*, **const** *std*::vector<double> &*positionRight*, **const** *std*::vector<double> &*momentumRight*) **const**
Computes NUTS criterion on euclidean domain.

Parameters

- **positionLeft** – Leftmost position.
- **momentumLeft** – Leftmost momentum.
- **positionRight** – Rightmost position.
- **momentumRight** – Rightmost momentum.

Returns Returns criterion if tree should be further increased.

inline virtual int **updateMetricMatricesEuclidean**(**const** *std*::vector<*std*::vector<double>> &*samples*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*)
Updates Inverse Metric by approximating the covariance matrix with the Fisher information.

Parameters

- **samples** – Vector of samples.
- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

Returns Error code of Cholesky decomposition.

inline virtual int **updateMetricMatricesRiemannian**(*std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*)
Updates Metric and Inverse Metric by using hessian.

Parameters

- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

Returns Error code to indicate if update was successful.

Public Members

size_t **_modelEvaluationCount**
Number of model evaluations.

korali::*Experiment* ***_k**
Pointer to the korali experiment.

korali::*problem*::*Sampling* ***samplingProblemPtr**
Pointer to the sampling problem (might be NULL)

korali::*problem*::*Bayesian* ***bayesianProblemPtr**
Pointer to the Bayesian problem (might be NULL)

double **_currentEvaluation**
Current evaluation of objective (return value of sample evaluation).

std::vector<double> **_currentGradient**
Current gradient of objective (return value of sample evaluation).

size_t **_stateSpaceDim**
State Space Dimension needed for *Leapfrog* integrator.

class *korali*::*solver*::*sampler*::**HamiltonianEuclidean** : **public** *korali*::*solver*::*sampler*::*Hamiltonian*
#include <hamiltonian_euclidean_base.hpp> Abstract base class for Euclidean *Hamiltonian* objects.

Subclassed by *korali::solver::sampler::HamiltonianEuclideanDense*, *korali::solver::sampler::HamiltonianEuclideanDiag*

Public Functions

HamiltonianEuclidean() = default
Default constructor.

inline HamiltonianEuclidean(**const** size_t *stateSpaceDim*, *korali*::*Experiment* **k*)
Constructor with State Space Dim.

Parameters

- **stateSpaceDim** – Dimension of State Space.
- **k** – Pointer to *Korali* object.

virtual ~HamiltonianEuclidean() = default
Destructor of abstract base class.

inline virtual double **tau**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**
Calculates tau(q, p) = 0.5 * momentum^T * inverseMetric(q) * momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual *std*::vector<double> **dtau_dq**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**
Calculates gradient of tau(q, p) wrt. position.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual *std*::vector<double> **dtau_dp**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates gradient of tau(q, p) wrt. momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual double **phi**() **override**

Calculates gradient of kinetic energy.

Returns Gradient of kinetic energy.

inline virtual *std*::vector<double> **dphi_dq**() **override**

Calculates gradient of kinetic energy.

Returns Gradient of kinetic energy.

class *korali*::*solver*::*sampler*::**HamiltonianEuclideanDense** : **public** *korali*::*solver*::*sampler*::*HamiltonianEu*

#include <hamiltonian_euclidean_dense.hpp> Used for calculating energies with euclidean metric.

Public Functions

inline HamiltonianEuclideanDense(**const** size_t *stateSpaceDim*, *korali*::*distribution*::*multivariate*::*Normal* **multivariateGenerator*, **const** *std*::vector<double> &*metric*, *korali*::*Experiment* **k*)

Constructor with State Space Dim.

Parameters

- **stateSpaceDim** – Dimension of State Space.
- **metric** – Metric of space.
- **multivariateGenerator** – Generator needed for momentum sampling.
- **k** – Pointer to *Korali* object.

~HamiltonianEuclideanDense() = default

Destructor of derived class.

inline virtual double **H**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Total energy function used for *Hamiltonian* Dynamics.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Inverse of current metric.

Returns Total energy.

inline virtual double **K**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Kinetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Inverse of current metric.

Returns Kinetic energy.

inline virtual *std*::vector<double> **dK**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Gradient of kintetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of kinetic energy wrt. current momentum.

inline virtual *std*::vector<double> **sampleMomentum**(**const** *std*::vector<double> &*metric*) **const override**

Generates sample of momentum.

Parameters **metric** – Current metric.

Returns Momentum sampled from normal distribution with metric as covariance matrix.

inline virtual double **innerProduct**(**const** *std*::vector<double> &*leftMomentum*, **const** *std*::vector<double> &*rightMomentum*, **const** *std*::vector<double> &*inverseMetric*) **const override**

Calculates inner product induced by inverse metric.

Parameters

- **leftMomentum** – Left vector of inner product.
- **rightMomentum** – Right vector of inner product.
- **inverseMetric** – Inverse of current metric.

Returns inner product

inline virtual int **updateMetricMatricesEuclidean**(**const** *std*::vector<*std*::vector<double>> &*samples*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*) **override**

Updates inverse Metric by approximating the covariance matrix with the Fisher information.

Parameters

- **samples** – Vector of samples.
- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

Returns Error code of Cholesky decomposition.

Protected Functions

inline int **invertMatrix**(**const** *std*::vector<double> &*matrix*, *std*::vector<double> &*inverseMat*)
Inverts s.p.d. matrix via Cholesky decomposition.

Parameters

- **matrix** – Input matrix interpreted as square symmetric matrix.
- **inverseMat** – Result of inversion.

Returns Error code of Cholesky decomposition used to invert matrix.

Private Members

korali::*distribution*::*multivariate*::*Normal* ***_multivariateGenerator**
Multivariate normal generator needed for sampling of momentum from dense metric.

class *korali*::*solver*::*sampler*::**HamiltonianEuclideanDiag** : **public** *korali*::*solver*::*sampler*::*HamiltonianEuc*
#include <hamiltonian_euclidean_diag.hpp> Used for calculating energies with unit euclidean metric.

Public Functions

inline **HamiltonianEuclideanDiag**(**const** size_t *stateSpaceDim*, *korali*::*Experiment* **k*)
Constructor with State Space Dim.

Parameters

- **stateSpaceDim** – Dimension of State Space.
- **k** – Pointer to *Korali* object.

inline **HamiltonianEuclideanDiag**(**const** size_t *stateSpaceDim*, *korali*::*distribution*::*univariate*::*Normal* **normalGenerator*, *korali*::*Experiment* **k*)
Constructor with State Space Dim.

Parameters

- **stateSpaceDim** – Dimension of State Space.
- **normalGenerator** – Generator needed for momentum sampling.
- **k** – Pointer to *Korali* object.

~HamiltonianEuclideanDiag() = default
Destructor of derived class.

inline **virtual** double **H**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**
Total energy function used for *Hamiltonian* Dynamics.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Inverse of current metric.

Returns Total energy.

inline virtual double **K**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Kinetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Inverse of current metric.

Returns Kinetic energy.

inline virtual *std*::vector<double> **dK**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Gradient of kintetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of kinetic energy wrt. current momentum.

inline virtual *std*::vector<double> **sampleMomentum**(**const** *std*::vector<double> &*metric*) **const override**

Generates sample of momentum.

Parameters **metric** – Current metric.

Returns Momentum sampled from normal distribution with metric as covariance matrix.

inline virtual double **innerProduct**(**const** *std*::vector<double> &*leftMomentum*, **const** *std*::vector<double> &*rightMomentum*, **const** *std*::vector<double> &*inverseMetric*) **const override**

Calculates inner product induces by inverse metric.

Parameters

- **leftMomentum** – Left vector of inner product.
- **rightMomentum** – Right vector of inner product.
- **inverseMetric** – Inverse of current metric.

Returns inner product

inline virtual int **updateMetricMatricesEuclidean**(**const** *std*::vector<*std*::vector<double>> &*samples*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*) **override**

Updates inverse Metric by approximating the covariance matrix with the Fisher information.

Parameters

- **samples** – Vector of samples.
- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

Returns Error code of Cholesky decomposition.

Private Members

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**
One dimensional normal generator needed for sampling of momentum from diagonal _metric.

class *korali*::*solver*::*sampler*::**HamiltonianRiemannian** : **public** *korali*::*solver*::*sampler*::*Hamiltonian*
#include <hamiltonian_riemannian_base.hpp> Abstract base class for *Hamiltonian* objects.

Subclassed by *korali::solver::sampler::HamiltonianRiemannianConstDense*, *korali::solver::sampler::HamiltonianRiemannianConstDiag*, *korali::solver::sampler::HamiltonianRiemannianDiag*

Public Functions

inline **HamiltonianRiemannian**(**const** size_t *stateSpaceDim*, *korali*::*Experiment* **k*)
Constructor with State Space Dim.

Parameters

- **stateSpaceDim** – Dimension of State Space.
- **k** – Pointer to *Korali* object.

virtual **~HamiltonianRiemannian**() = default
Destructor of abstract base class.

inline *std*::vector<double> **hessianU**() **const**
Hessian of potential energy function used for Riemannian metric.

Returns Hessian of potential energy.

inline double **softAbsFunc**(**const** double *x*, **const** double *alpha*) **const**
Helper function f(x) = x * coth(alpha * x) for SoftAbs metric.

Parameters

- **x** – Point of evaluation.
- **alpha** – Hardness parameter

Returns function value at x.

Public Members

std::vector<double> **_currentHessian**
Current Hessian of objective (return value of sample evaluation).

double **_logDetMetric**
normalization constant for kinetic energy (isn't constant compared to Euclidean *Hamiltonian* => have to include term in calculation)

class *korali*::*solver*::*sampler*::**HamiltonianRiemannianConstDense** : **public** *korali*::*solver*::*sampler*::*Hamil*
#include <hamiltonian_riemannian_const_dense.hpp> Used for dense Riemannian metric.

Public Functions

inline HamiltonianRiemannianConstDense(**const** size_t *stateSpaceDim*, *korali*::*distribution*::*multivariate*::*Normal* *multivariateGenerator*, **const** *std*::vector<double> &*metric*, **const** double *inverseRegularizationParam*, *korali*::*Experiment* *k*)

Constructor with State Space Dim.

Parameters

- **stateSpaceDim** – Dimension of State Space.
- **multivariateGenerator** – Generator needed for momentum sampling.
- **metric** – Metric of space.
- **inverseRegularizationParam** – Inverse regularization parameter of SoftAbs metric that controls hardness of approximation: For large values inverseMetric is closer to analytical formula (and therefore closer to degeneracy in certain cases).
- **k** – Pointer to *Korali* object.

~HamiltonianRiemannianConstDense() = default

Destructor of derived class.

inline virtual double **H**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Total energy function used for *Hamiltonian* Dynamics.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Inverse of current metric.

Returns Total energy.

inline virtual double **K**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Kinetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Inverse of current metric.

Returns Kinetic energy.

inline virtual *std*::vector<double> **dK**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Gradient of kintetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of kinetic energy wrt. current momentum.

inline virtual double **tau**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates tau(q, p) = 0.5 * momentum^T * inverseMetric(q) * momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual *std*::vector<double> **dtau_dq**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates gradient of tau(q, p) wrt. position.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual *std*::vector<double> **dtau_dp**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates gradient of tau(q, p) wrt. momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual double **phi**() **override**

Calculates gradient of kinetic energy.

Returns Gradient of kinetic energy.

inline virtual *std*::vector<double> **dphi_dq**() **override**

Calculates gradient of kinetic energy.

Returns Gradient of kinetic energy.

inline virtual void **updateHamiltonian**(**const** *std*::vector<double> &*position*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*) **override**

Updates current position of hamiltonian.

Parameters

- **position** – Current position.
- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

inline virtual *std*::vector<double> **sampleMomentum**(**const** *std*::vector<double> &*metric*) **const override**

Generates sample of momentum.

Parameters **metric** – Current metric.

Returns Momentum sampled from normal distribution with metric as covariance matrix.

inline virtual double **innerProduct**(**const** *std*::vector<double> &*momentumLeft*, **const** *std*::vector<double> &*momentumRight*, **const** *std*::vector<double> &*inverseMetric*) **const override**

Calculates inner product induces by inverse metric.

Parameters

- **momentumLeft** – Left vector of inner product.
- **momentumRight** – Right vector of inner product.
- **inverseMetric** – Inverse of current metric.

Returns inner product

inline virtual int **updateMetricMatricesRiemannian**(*std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*) **override**

Updates Metric and Inverse Metric by using hessian.

Parameters

- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

Returns Error code to indicate if update was successful.

Public Members

double **_inverseRegularizationParam**

Inverse regularization parameter of SoftAbs metric that controls hardness of approximation.

Private Members

korali::*distribution*::*multivariate*::*Normal* ***_multivariateGenerator**

Multi dimensional normal generator needed for sampling of momentum from dense metric.

gsl_matrix ***Q**

gsl_vector ***lambda**

gsl_eigen_symmv_workspace ***w**

gsl_matrix ***lambdaSoftAbs**

gsl_matrix ***inverseLambdaSoftAbs**

gsl_matrix ***tmpMatOne**

gsl_matrix ***tmpMatTwo**

gsl_matrix ***tmpMatThree**

gsl_matrix ***tmpMatFour**

class *korali*::*solver*::*sampler*::**HamiltonianRiemannianConstDiag** : **public** *korali*::*solver*::*sampler*::*Hamiltc*

#include <hamiltonian_riemannian_const_diag.hpp> Used for diagonal Riemannian metric.

Public Functions

inline HamiltonianRiemannianConstDiag(**const** size_t *stateSpaceDim*, *korali*::*distribution*::*univariate*::*Normal* **normalGenerator*, **const** double *inverseRegularizationParam*, *korali*::*Experiment* **k*)

Constructor with State Space Dim.

Parameters

- **stateSpaceDim** – Dimension of State Space.
- **normalGenerator** – Generator needed for momentum sampling.
- **inverseRegularizationParam** – Inverse regularization parameter of SoftAbs metric that controls hardness of approximation: For large values inverseMetric is closer to analytical formula (and therefore closer to degeneracy in certain cases).
- **k** – Pointer to *Korali* object.

~HamiltonianRiemannianConstDiag() = default

Destructor of derived class.

inline virtual double **H**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Total energy function used for *Hamiltonian* Dynamics.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Inverse of current metric.

Returns Total energy.

inline virtual double **K**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Kinetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Inverse of current metric.

Returns Kinetic energy.

inline virtual *std*::vector<double> **dK**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Gradient of kintetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of kinetic energy wrt. current momentum.

inline virtual double **tau**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates tau(q, p) = 0.5 * momentum^T * inverseMetric(q) * momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual *std*::vector<double> **dtau_dq**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates gradient of tau(q, p) wrt. position.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual *std*::vector<double> **dtau_dp**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates gradient of tau(q, p) wrt. momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual double **phi**() **override**

Calculates gradient of kinetic energy.

Returns Gradient of kinetic energy.

inline virtual *std*::vector<double> **dphi_dq**() **override**

Calculates gradient of kinetic energy.

Returns Gradient of kinetic energy.

inline virtual void **updateHamiltonian**(**const** *std*::vector<double> &*position*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*) **override**

Updates current position of hamiltonian.

Parameters

- **position** – Current position.
- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

inline virtual *std*::vector<double> **sampleMomentum**(**const** *std*::vector<double> &*metric*) **const override**

Generates sample of momentum.

Parameters **metric** – Current metric.

Returns Momentum sampled from normal distribution with metric as covariance matrix.

inline virtual double **innerProduct**(**const** *std*::vector<double> &*momentumLeft*, **const** *std*::vector<double> &*momentumRight*, **const** *std*::vector<double> &*inverseMetric*) **const override**

Calculates inner product induces by inverse metric.

Parameters

- **momentumLeft** – Left vector of inner product.
- **momentumRight** – Right vector of inner product.

- **inverseMetric** – Inverse of curret metric.

Returns inner product

inline virtual int **updateMetricMatricesRiemannian**(*std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*) **override**

Updates Metric and Inverse Metric by using hessian.

Parameters

- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

Returns Error code to indicate if update was successful.

Public Members

double **_inverseRegularizationParam**

Inverse regularization parameter of SoftAbs metric that controls hardness of approximation.

Private Members

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**

One dimensional normal generator needed for sampling of momentum from diagonal metric.

class *korali*::*solver*::*sampler*::**HamiltonianRiemannianDiag** : **public** *korali*::*solver*::*sampler*::*HamiltonianRi*

#include <hamiltonian_riemannian_diag.hpp> Used for diagonal Riemannian metric.

Public Functions

inline HamiltonianRiemannianDiag(**const** size_t *stateSpaceDim*, *korali*::*distribution*::*univariate*::*Normal* ***normalGenerator*, **const** double *inverseRegularizationParam*, *korali*::*Experiment* **k*)

Constructor with State Space Dim.

Parameters

- **stateSpaceDim** – Dimension of State Space.
- **normalGenerator** – Generator needed for momentum sampling.
- **inverseRegularizationParam** – Inverse regularization parameter of SoftAbs metric that controls hardness of approximation: For large values inverseMetric is closer to analytical formula (and therefore closer to degeneracy in certain cases).
- **k** – Pointer to *Korali* object.

~HamiltonianRiemannianDiag() = default

Destructor of derived class.

inline virtual double **H**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Total energy function used for *Hamiltonian* Dynamics.

Parameters

- **momentum** – Current momentum.

- **inverseMetric** – Inverse of current metric.

Returns Total energy.

inline virtual double **K**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Kinetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Inverse of current metric.

Returns Kinetic energy.

inline virtual *std*::vector<double> **dK**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Gradient of kintetic energy function.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of kinetic energy wrt. current momentum.

inline virtual double **tau**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates tau(q, p) = 0.5 * momentum^T * inverseMetric(q) * momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual *std*::vector<double> **dtau_dq**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates gradient of tau(q, p) wrt. position.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual *std*::vector<double> **dtau_dp**(**const** *std*::vector<double> &*momentum*, **const** *std*::vector<double> &*inverseMetric*) **override**

Calculates gradient of tau(q, p) wrt. momentum.

Parameters

- **momentum** – Current momentum.
- **inverseMetric** – Current inverseMetric.

Returns Gradient of Kinetic energy with current momentum.

inline virtual double **phi**() **override**

Purely virtual gradient of phi(q) = 0.5 * logDetMetric(q) + U(q) used for *Hamiltonian* Dynamics.

Returns Gradient of Kinetic energy with current momentum.

inline virtual *std*::vector<double> **dphi_dq**() **override**
Calculates gradient of kinetic energy.

Returns Gradient of kinetic energy.

inline virtual void **updateHamiltonian**(**const** *std*::vector<double> &*position*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*) **override**
Updates current position of hamiltonian.

Parameters

- **position** – Current position.
- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.

inline virtual *std*::vector<double> **sampleMomentum**(**const** *std*::vector<double> &*metric*) **const override**
Generates sample of momentum.

Parameters **metric** – Current metric.

Returns *Sample* of momentum from normal distribution with covariance matrix metric. Only variance taken into account with diagonal metric.

inline virtual double **innerProduct**(**const** *std*::vector<double> &*momentumLeft*, **const** *std*::vector<double> &*momentumRight*, **const** *std*::vector<double> &*inverseMetric*) **const override**
Calculates inner product induces by inverse metric.

Parameters

- **momentumLeft** – Left vector of inner product.
- **momentumRight** – Right vector of inner product.
- **inverseMetric** – Inverse of current metric.

Returns inner product

inline double **taylorSeriesPhiFunc**(**const** double *x*, **const** double *alpha*)
Helper function f(x) = 1/x - alpha * x / (sinh(alpha * x^2) * cosh(alpha * x^2)) for SoftAbs metric.

Parameters

- **x** – Point of evaluation.
- **alpha** – Hyperparameter.

Returns function value at x.

inline double **taylorSeriesTauFunc**(**const** double *x*, **const** double *alpha*)
Helper function f(x) = 1/x * (alpha / cosh(alha * x^2)^2 - tanh(alpha * x^2) / x^2) for SoftAbs metric.

Parameters

- **x** – Point of evaluation.
- **alpha** – Hyperparameter.

Returns function value at x.

Public Members

double **_inverseRegularizationParam**
Inverse regularization parameter of SoftAbs metric that controls hardness of approximation.

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**
One dimensional normal generator needed for sampling of momentum from diagonal metric.

class *korali*::*problem*::**Hierarchical** : **public** *korali*::*Problem*
#include <hierarchical.hpp> Class declaration for module: *Hierarchical*.

Subclassed by *korali::problem::hierarchical::Psi*, *korali::problem::hierarchical::Theta*, *korali::problem::hierarchical::ThetaNew*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

bool **isSampleFeasible**(*korali*::*Sample* &*sample*)
Checks whether the proposed sample fits within the range of the prior distribution.

Parameters **sample** – A *Korali Sample*

Returns True, if feasible; false, otherwise.

virtual void **evaluate**(*korali*::*Sample* &*sample*)
Produces a generic evaluation from the Posterior distribution of the sample, for optimization with CMAES, DEA, storing it in and stores it in sample["F(x)"].

Parameters **sample** – A *Korali Sample*

void **evaluateLogPrior**(*korali*::*Sample* &*sample*)
Evaluates the log prior of the given sample, and stores it in sample["Log Prior"].

Parameters **sample** – A *Korali Sample*

virtual void **evaluateLogLikelihood**(*korali*::*Sample* &*sample*) = 0
Evaluates the log likelihood of the given sample, and stores it in sample["Log Likelihood"].

Parameters **sample** – A *Korali Sample*

void **evaluateLogPosterior**(*korali*::*Sample* &*sample*)
Evaluates the log posterior of the given sample, and stores it in sample["Log Posterior"].

Parameters **sample** – A *Korali Sample*

class *korali*::*solver*::*sampler*::**HMC** : **public** *korali*::*solver*::*Sampler*
#include <HMC.hpp> Class declaration for module: *HMC*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **setInitialConfiguration**() **override**
Configures *HMC*.

virtual void **finalize**() **override**
Final console output at termination.

virtual void **runGeneration**() **override**
Generate a sample and evaluate it.

virtual void **printGenerationBefore**() **override**
Console Output before generation runs.

virtual void **printGenerationAfter**() **override**
Console output after generation.

Public Members

size_t **_burnIn**
Specifies the number of preliminary *HMC* steps before samples are being drawn. This may reduce effects from improper initialization.

int **_useDiagonalMetric**
Specifies if Metric is restricted to be diagonal.

size_t **_numIntegrationSteps**
Number of Integration steps used in *Leapfrog* scheme. Only relevant if Adaptive Step Size not used.

size_t **_maxIntegrationSteps**
Number of Integration steps used in *Leapfrog* scheme. Only relevant if Adaptive Step Size is used.

int **_useNUTS**
Specifies if No-U-Turn *Sampler* (NUTS) is used.

double **_stepSize**
Step size used in *Leapfrog* scheme.

int **_useAdaptiveStepSize**
Controls whether dual averaging technique for adaptive step size calibration is used.

double **_targetAcceptanceRate**
Desired Acceptance Rate for Adaptive Step Size calibration.

double **_acceptanceRateLearningRate**
Learning rate of running acceptance rate estimate.

double **_targetIntegrationTime**
Targeted Integration Time for *Leapfrog* scheme. Only relevant if Adaptive Step Size used.

double **_adaptiveStepSizeSpeedConstant**
Controls how fast the step size is adapted. Only relevant if Adaptive Step Size used.

double **_adaptiveStepSizeStabilizationConstant**
Controls stability of adaptive step size calibration during the inital iterations. Only relevant if Adaptive Step Size used.

double **_adaptiveStepSizeScheduleConstant**
Controls the weight of the previous step sizes. Only relevant if Adaptive Step Size used. The smaller the higher the weight.

size_t **_maxDepth**
Sets the maximum depth of NUTS binary tree.

std::string **_version**
Metric can be set to 'Static', 'Euclidean' or 'Riemannian'.

double **_inverseRegularizationParameter**
Controls hardness of inverse metric approximation: For large values the Inverse Metric is closer the to Hessian (and therefore closer to degeneracy in certain cases).

size_t **_maxFixedPointIterations**
Max number of fixed point iterations during implicit leapfrog scheme.

double **_stepSizeJitter**
Step Size Jitter to vary trajectory length. Number must be in the interval [0.0. 1.0]. A uniform realization between [-(Step Size Jitter) * (Step Size), (Step Size Jitter) * (Step Size)) is sampled and added to the current Step Size.

size_t **_initialFastAdaptionInterval**
Initial warm-up interval during which step size is adaptively adjusted.

size_t **_finalFastAdaptionInterval**
Final warm-up interval during which step size is adaptively adjusted.

size_t **_initialSlowAdaptionInterval**
Lenght of first (out of 5) warm-up intervals during which euclidean metric is adapted. The length of each following slow adaption intervals is doubled.

Metric **_metricType**
[Internal Use] Metric Type can be set to 'Static', 'Euclidean' or 'Riemannian'.

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**
[Internal Use] Normal random number generator.

korali::*distribution*::*multivariate*::*Normal* ***_multivariateGenerator**
[Internal Use] Random number generator with a multivariate normal distribution.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Uniform random number generator.

double **_acceptanceRate**
[Internal Use] Ratio proposed to accepted samples (including Burn In period).

double **_runningAcceptanceRate**
[Internal Use] Running estimate of current acceptance rate.

size_t **_acceptanceCount**
[Internal Use] Number of accepted samples (including Burn In period).

size_t **_proposedSampleCount**
[Internal Use] Number of proposed samples.

std::vector<*std*::vector<double>> **_sampleDatabase**
[Internal Use] Parameters generated by *HMC* and stored in the database.

std::vector<*std*::vector<double>> **_euclideanWarmupSampleDatabase**
[Internal Use] Parameters generated during warmup. Used for Euclidean Metric approximation.

std::vector<double> **_sampleEvaluationDatabase**
[Internal Use] *Sample* evaluations coresponding to the samples stored in *Sample* Databse.

size_t **_chainLength**
[Internal Use] Current Chain Length (including Burn In and Leaped Samples).

double **_leaderEvaluation**
[Internal Use] Evaluation of leader.

double **_candidateEvaluation**
[Internal Use] Evaluation of candidate.

std::vector<double> **_positionLeader**
[Internal Use] Variables of the newest position/sample in the Markov chain.

std::vector<double> **_positionCandidate**
[Internal Use] Candidate position to be accepted or rejected.

std::vector<double> **_momentumLeader**
[Internal Use] Latest momentum sample.

std::vector<double> **_momentumCandidate**
[Internal Use] Proposed momentum after propagating Chain Leader and Momentum Leader according to *Hamiltonian* dynamics.

double **_logDualStepSize**
: [Internal Use] Logarithm of smoothed average step size. Step size that is used after burn in period. Only relevant if adaptive step size used.

double **_mu**
: [Internal Use] Constant used for Adaptive Step Size option.

double **_hBar**
: [Internal Use] Constant used for Adaptive Step Size option.

double **_acceptanceCountNUTS**
: [Internal Use] TODO: is this the number of accepted proposals?

size_t **_currentDepth**
: [Internal Use] Depth of NUTS binary tree in current generation.

double **_acceptanceProbability**
: [Internal Use] Metropolis update acceptance probability - usually denoted with alpha - needed due to numerical error during integration.

double **_acceptanceRateError**
: [Internal Use] Accumulated differences of Acceptance Probability and Target Acceptance Rate.

std::vector<double> **_metric**
: [Internal Use] Metric for proposal distribution.

std::vector<double> **_inverseMetric**
: [Internal Use] Inverse Metric for proposal distribution.

size_t **_maxSamples**
: [Termination Criteria] Number of Samples to Generate.

Private Functions

void **updateState**()
: Updates internal state such as mean, Metric and InverseMetric.

void **finishSample**(size_t *sampleId*)
: Process sample after evaluation.

void **runGenerationHMC**(**const** double *logUniSample*)
: Runs generation of *HMC* sampler.

  Parameters **logUniSample** – Log of uniform sample needed for Metropolis accepance / rejection step.

void **runGenerationNUTS**(**const** double *logUniSample*)
: Runs NUTS algorithm with buildTree.

  Parameters **logUniSample** – Log of uniform sample needed for Metropolis accepance / rejection step.

void **runGenerationNUTSRiemannian**(**const** double *logUniSample*)
: Runs NUTS algorithm with buildTree.

  Parameters **logUniSample** – Log of uniform sample needed for Metropolis accepance / rejection step.

void **saveSample**()
: Saves sample.

void **updateStepSize**()
Updates Step Size for Adaptive Step Size.

void **buildTree**(*std*::shared_ptr<*TreeHelperEuclidean*> *helper*, **const** size_t *depth*)
Recursive binary tree building algorithm. Applied if configuration 'Use NUTS' is set to True.

Parameters

- **helper** – Helper struct for large argument list.
- **depth** – Current depth of binary tree.

void **buildTreeIntegration**(*std*::shared_ptr<*TreeHelperRiemannian*> *helper*, *std*::vector<double> &*rho*, **const** size_t *depth*)
Recursive binary tree building algorithm. Applied if configuration 'Use NUTS' is set to True.

Parameters

- **helper** – Helper struct for large argument list.
- **rho** – Sum of momenta encountered along path.
- **depth** – Current depth of binary tree.

Private Members

std::shared_ptr<*Hamiltonian*> **_hamiltonian**

std::unique_ptr<*Leapfrog*> **_integrator**

class *korali*::*distribution*::*univariate*::**Igamma** : **public** *korali*::*distribution*::*Univariate*
#include <igamma.hpp> Class declaration for module: *Igamma*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_shape**
[Conditional Variable Value] The shape parameter of the inverse gamma distribution.

std::string **_shapeConditional**
[Conditional Variable Reference] The shape parameter of the inverse gamma distribution.

double **_scale**
[Conditional Variable Value] The scale parameter of the inverse gamma distribution.

std::string **_scaleConditional**
[Conditional Variable Reference] The scale parameter of the inverse gamma distribution.

Private Members

double **_auxLog**

class *korali*::*neuralNetwork*::*layer*::**Input** : **public** *korali*::*neuralNetwork*::*Layer*
#include <input.hpp> Class declaration for module: *Input*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **forwardData**(**const** size_t *t*) **override**
Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

virtual void **backwardData**(**const** size_t *t*) **override**
Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

class *korali*::*problem*::**Integration** : **public** *korali*::*Problem*
#include <integration.hpp> Class declaration for module: *Integration*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

void **execute**(*korali*::*Sample* &*sample*)
Produces an evaluation of the model, storing it in and stores it in sample["Evaluation"].

Parameters **sample** – A *Korali Sample*

Public Members

std::uint64_t **_integrand**
Stores the function to integrate.

class *korali*::*solver*::**Integrator** : **public** *korali*::*Solver*
#include <integrator.hpp> Class declaration for module: *Integrator*.

Subclassed by *korali::solver::integrator::MonteCarlo*, *korali::solver::integrator::Quadrature*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **launchSample**(size_t *sampleIndex*) = 0
Prepares and launches a sample to be evaluated.

Parameters **sampleIndex** – index of sample to be launched

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

virtual void **runGeneration**() **override**
Runs the current generation.

virtual void **printGenerationBefore**() **override**
Prints solver information before the execution of the current generation.

virtual void **printGenerationAfter**() **override**
Prints solver information after the execution of the current generation.

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

Public Members

size_t **_executionsPerGeneration**
Specifies the number of model executions per generation. By default this setting is 0, meaning that all executions will be performed in the first generation. For values greater 0, executions will be split into batches and split int generations for intermediate output.

double **_accumulatedIntegral**
[Internal Use] Current value of the integral.

std::vector<*std*::vector<float>> **_gridPoints**
[Internal Use] Gridpoints for quadrature.

float **_weight**
[Internal Use] Precomputed weight for MC sample.

std::vector<*Sample*> **_samples**
Container for samples to be evaluated per generation.

template<typename **keyType**, typename **valType**, typename **timerType**>
class *korali*::**kCache**
#include <kcache.hpp> This class defines a circular buffer with overwrite policy on add.

Public Functions

inline **kCache**()
Lock for thread-safe operation.

Default constructor

inline void **setMaxAge**(**const** *timerType* &*maxAge*)
Re-sets the maximum age threshold.

Parameters **maxAge** – the maximum age threshold

inline void **setTimer**(*timerType* **timer*)
Sets the pointer to an external timer.

Parameters **timer** – The external timer

inline void **set**(**const** *keyType* &*key*, **const** *valType* &*val*)
Updates the value of a data element in the cache.

Parameters

- **key** – Key of the data element to update
- **val** – Value of the data element to update

inline void **set**(**const** *keyType* &*key*, **const** *valType* &*val*, **const** *timerType* &*time*)
Updates the value of a data element in the cache, forcing a specific time for it.

Parameters

- **key** – Key of the data element to update
- **val** – Value of the data element to update
- **time** – Time assigned to the data element

inline bool **contains**(**const** *keyType* &*key*)
Checks whether a given data element is present in cache.

Parameters **key** – Key of the data element to check

Returns Whether or not the data element is present

inline *valType* **get**(**const** *keyType* &*key*)
Reads the value of a data element from the cache. The data element should be present, or an error will occur.

Parameters **key** – Key of the data element to access

Returns The value of the element stored in cache

inline *valType* **access**(**const** *keyType* &*key*, *std*::function<*valType*) void
> *func*Reads the value of a data element from the cache. If the element is not present, it calls the provided function to generate it.

Parameters

- **key** – Key of the data element to access
- **func** – Function (lambda or regular) that, upon calling, will return the new value for the element.

Returns The value of the element stored in cache or generated by the function

inline *std*::vector<*keyType*> **getKeys**()
Returns the stored entry keys as an ordered vector.

Returns A vector containing all ordered keys

inline *std*::vector<*valType*> **getValues**()
Returns the stored entry values in the cache.

Returns A vector containing all stored entry values, ordered by key

inline *std*::vector<*timerType*> **getTimes**()
Returns the stored entry times in the cache.

Returns A vector containing all stored entry times, ordered by key

Private Members

std::map<*keyType*, *cacheElement_t*<*valType*, *timerType*>> **_data**
Container for cache elements.

timerType ***_timer**
Pointer to the external timer.

timerType **_maxAge**
Maximum age threshold of data before expiration.

class *korali*::**KoraliJson**
#include <koraliJson.hpp> This class encapsulates a JSON object, making it compatible with *Korali* C++ objects and Pybind11.

Public Functions

KoraliJson()

knlohmann::json &**getJson**()
Function to obtain the JSON object.

Returns A reference to the JSON object.

void **getCopy**(knlohmann::json &*dst*) **const**
Function to make a copy of the JSON object.

Parameters **dst** – destination js

void **setJson**(knlohmann::json &*js*)
Function to set the JSON object.

Parameters **js** – The input JSON object.

pybind11::object **getItem**(**const** pybind11::object *key*)
Gets an item from the JSON object at the current pointer position.

Parameters **key** – A pybind11 object acting as JSON key (number or string).

Returns A pybind11 object

void **setItem**(**const** pybind11::object *key*, **const** pybind11::object *val*)
Sets an item on the JSON object at the current pointer position.

Parameters

- **key** – A pybind11 object acting as JSON key (number or string).
- **val** – The value of the item to set.

knlohmann::json &**operator[]**(**const** *std*::string &*key*)
C++ wrapper for the getItem operator.

Parameters **key** – A C++ string acting as JSON key.

Returns The referenced JSON object content.

knlohmann::json &**operator[]**(**const** unsigned long int &*key*)
C++ wrapper for the getItem operator.

Parameters **key** – A C++ integer acting as JSON key.

Returns The referenced JSON object content.

bool **contains**(**const** *std*::string &*key*)
Indicates whether the JSON object contains the given path.

Parameters **key** – key A C++ string acting as JSON key.

Returns true, if path is found; false, otherwise.

void **traverseKey**(pybind11::object *key*)
Advances the JSON object pointer, given the key.

Parameters **key** – A C++ string acting as JSON key.

Public Members

knlohmann::json **_js**
Container for the JSON object.

knlohmann::json ***_opt**
Pointer that stores the current access position of the JSON object. It advances with getItem, and resets upon setJson or finding a native data type (not a path).

class *korali*::*distribution*::*univariate*::**Laplace** : **public** *korali*::*distribution*::*Univariate*
#include <laplace.hpp> Class declaration for module: *Laplace*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian** (double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber** () **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_mean**
[Conditional Variable Value] The mean of the *Laplace* distribution.

std::string **_meanConditional**
[Conditional Variable Reference] The mean of the *Laplace* distribution.

double **_width**
[Conditional Variable Value] The scale of the *Laplace* distribution, it controls the variance.

std::string **_widthConditional**
[Conditional Variable Reference] The scale of the *Laplace* distribution, it controls the variance.

class *korali*::*neuralNetwork*::**Layer** : **public** *korali*::*Module*
#include <layer.hpp> Class declaration for module: *Layer*.

Subclassed by *korali::neuralNetwork::layer::Activation*, *korali::neuralNetwork::layer::Convolution*, *korali::neuralNetwork::layer::Deconvolution*, *korali::neuralNetwork::layer::Input*, *korali::neuralNetwork::layer::Linear*, *korali::neuralNetwork::layer::Output*, *korali::neuralNetwork::layer::Pooling*, *korali::neuralNetwork::layer::Recurrent*

Public Functions

virtual void **getConfiguration** (knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration** (knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults** (knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults** () **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

Layer () = default
Default constructor.

virtual ~Layer () = default
Default destructor.

std::vector<*std*::vector<float>> **getOutput** ()
Returns the output values for the current layer.

Returns The output values

virtual *std*::vector<float> **generateInitialHyperparameters** ()
Generates the initial weight/bias hyperparameters for the layer.

Returns The initial hyperparameters

virtual void **createHyperparameterMemory** ()
Initializes the layer's internal memory structures for hyperparameter storage.

inline **virtual** void **copyHyperparameterPointers** (*Layer* **dstLayer*)
Replicates the pointers for the current layer onto a destination layer.

Parameters **dstLayer** – The destination layer onto which to copy the pointers

virtual void **createForwardPipeline** ()
Initializes the layer's internal memory structures for the forward pipeline.

virtual void **createBackwardPipeline** ()
Initializes the internal memory structures for the backward pipeline.

virtual void **forwardData** (**const** size_t *t*) = 0
Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

inline **virtual** void **setHyperparameters** (**const** float **hyperparameters*)
Updates layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (Input) Pointer to read the hyperparameters from.

inline **virtual** void **getHyperparameters** (float **hyperparameters*)
Gets layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (Output) Pointer to write the hyperparameters to.

inline **virtual** void **getHyperparameterGradients** (float **gradient*)
Gets the gradients of the layer's output wrt to is hyperparameters (e.g., weights and biases)

Parameters **gradient** – (Output) Pointer to write the hyperparameter gradients to.

virtual void **backwardData** (**const** size_t *t*) = 0
Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

virtual void **backwardHyperparameters** (**const** size_t *t*)
Calculates the gradients of layer hyperparameters.

Parameters **t** – Indicates the current timestep

Public Members

size_t **_outputChannels**
Indicates the size of the output vector produced by the layer.

float **_weightScaling**
Factor that is mutliplied by the layers' weights.

size_t **_index**
Index of the current layer within the NN.

NeuralNetwork ***_nn**
Pointer to the parent neural network.

layerPipeline_t ***_pipeline**
A pointer to the layer's containing pipeline.

Layer ***_prevLayer**
Pointer to previous layer, NULL if this is the first layer.

Layer ***_nextLayer**
Pointer to next layer, NULL if this is the last layer.

size_t **_hyperparameterCount**
Number of layer hyperparameters.

size_t **_hyperparameterIndex**
Starting index of hyperparameters.

size_t **_batchSize**
Contains the batch size corresponding to the pipeline.

float ***_outputValues**
Contains the output values of the layer.

float ***_outputGradient**
Contains the gradients of the outputs of the layer.

struct *korali*::**layerPipeline_t**
#include <neuralNetwork.hpp> Structure containing the information of a layer pipeline. There is one pipeline per threadCount x batchSize combination.

Public Members

std::vector<*korali*::*neuralNetwork*::*Layer**> **_layerVector**
Internal container for the NN layer forward/backward pipelines.

std::vector<float> **_rawInputValues**
Raw data for the NN input values. Format: TxNxIC (T: Time steps, N: Mini-batch size, IC: Input channels).

std::vector<float> **_rawInputGradients**
Raw data for the NN input gradients. Format: TxNxIC (T: Time steps, N: Mini-batch size, IC: Input channels).

std::vector<*std*::vector<float>> **_inputGradients**
Formatted data for the NN input gradients. Format: NxIC (N: Mini-batch size, IC: Input channels).

std::vector<float> **_rawOutputValues**
Raw data for the NN output values. Format: NxOC (N: Mini-batch size, OC: Output channels).

std::vector<float> **_rawOutputGradients**
Raw data for the NN output gradients. Format: NxOC (N: Mini-batch size, OC: Output channels).

std::vector<*std*::vector<float>> **_outputValues**
Formatted data for the NN output values. Format: NxOC (N: Mini-batch size, OC: Output channels).

std::vector<float> **_hyperparameterGradients**
F data for the NN hyperparameter gradients. Format: H (H: Hyperparameter count).

std::vector<size_t> **_inputBatchLastStep**
Remembers the position of the last timestep provided as input.

class *korali*::*solver*::*sampler*::**Leapfrog**
#include <leapfrog_base.hpp> Abstract base class used for time propagation according to hamiltonian dynamics via *Leapfrog* integration schemes.

Subclassed by *korali::solver::sampler::LeapfrogExplicit*, *korali::solver::sampler::LeapfrogImplicit*

Public Functions

inline Leapfrog(*std*::shared_ptr<*Hamiltonian*> *hamiltonian*)
Abstract base class constructor for explicit or implicit leapfrog stepper.

Parameters **hamiltonian** – *Hamiltonian* of the system.

virtual ~Leapfrog() = default
Default destructor.

virtual void **step**(*std*::vector<double> &*position*, *std*::vector<double> &*momentum*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*, **const** double *stepSize*) = 0
Purely virtual stepping function of the integrator.

Parameters

- **position** – Position which is evolved.
- **momentum** – Momentum which is evolved.
- **metric** – Current mentric.
- **inverseMetric** – Inverse metric.
- **stepSize** – Step Size used for Leap Frog Scheme.

Protected Attributes

std::shared_ptr<*Hamiltonian*> **_hamiltonian**
Pointer to hamiltonian object to calculate energies..

class *korali*::*solver*::*sampler*::**LeapfrogExplicit** : **public** *korali*::*solver*::*sampler*::*Leapfrog*
#include <leapfrog_explicit.hpp> Used for time propagation according to hamiltonian dynamics via explicit *Leapfrog* integration.

Public Functions

inline LeapfrogExplicit (*std*::shared_ptr<*Hamiltonian*> *hamiltonian*)
Constructor for explicit leapfrog stepper.

Parameters **hamiltonian** – *Hamiltonian* of the system.

virtual ~LeapfrogExplicit () = default
Default destructor.

inline virtual void **step** (*std*::vector<double> &*position*, *std*::vector<double> &*momentum*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*, **const** double *stepSize*) **override**
Explicit *Leapfrog* stepping scheme used for evolving *Hamiltonian* Dynamics.

Parameters

- **position** – Position which is evolved.
- **momentum** – Momentum which is evolved.
- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.
- **stepSize** – Step Size used for Leap Frog Scheme.

class *korali*::*solver*::*sampler*::**LeapfrogImplicit** : **public** *korali*::*solver*::*sampler*::*Leapfrog*
#include <leapfrog_implicit.hpp> Used for time propagation according to hamiltonian dynamics via implicit *Leapfrog* integration.

Public Functions

inline LeapfrogImplicit (size_t *maxNumFixedPointIter*, *std*::shared_ptr<*Hamiltonian*> *hamiltonian*)
Constructor for implicit leapfrog stepper.

Parameters

- **maxNumFixedPointIter** – Maximum fixed point iterations.
- **hamiltonian** – *Hamiltonian* of the system.

inline virtual void **step** (*std*::vector<double> &*position*, *std*::vector<double> &*momentum*, *std*::vector<double> &*metric*, *std*::vector<double> &*inverseMetric*, **const** double *stepSize*) **override**
Implicit *Leapfrog* stepping scheme used for evolving *Hamiltonian* Dynamics.

Parameters

- **position** – Position which is evolved.
- **momentum** – Momentum which is evolved.
- **metric** – Current metric.
- **inverseMetric** – Inverse of current metric.
- **stepSize** – Step Size used for Leap Frog Scheme.

Private Members

size_t **_maxNumFixedPointIter**
Maximum fixed point iterations during each step.

class *korali*::*neuralNetwork*::*layer*::**Linear** : **public** *korali*::*neuralNetwork*::*Layer*
#include <linear.hpp> Class declaration for module: *Linear*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **copyHyperparameterPointers**(*Layer* **dstLayer*) **override**
Replicates the pointers for the current layer onto a destination layer.

Parameters **dstLayer** – The destination layer onto which to copy the pointers

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual *std*::vector<float> **generateInitialHyperparameters**() **override**
Generates the initial weight/bias hyperparameters for the layer.

Returns The initial hyperparameters

virtual void **createHyperparameterMemory**() **override**
Initializes the layer's internal memory structures for hyperparameter storage.

virtual void **createForwardPipeline**() **override**
Initializes the layer's internal memory structures for the forward pipeline.

virtual void **createBackwardPipeline**() **override**
Initializes the internal memory structures for the backward pipeline.

virtual void **forwardData**(**const** size_t *t*) **override**
Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

virtual void **setHyperparameters**(**const** float **hyperparameters*) **override**
Updates layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (*Input*) Pointer to read the hyperparameters from.

virtual void **getHyperparameters**(float **hyperparameters*) **override**
Gets layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (*Output*) Pointer to write the hyperparameters to.

virtual void **getHyperparameterGradients**(float **gradient*) **override**
Gets the gradients of the layer's output wrt to is hyperparameters (e.g., weights and biases)

Parameters **gradient** – (*Output*) Pointer to write the hyperparameter gradients to.

virtual void **backwardData**(**const** size_t *t*) **override**
Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

virtual void **backwardHyperparameters**(**const** size_t *t*) **override**
Calculates the gradients of layer hyperparameters.

Parameters **t** – Indicates the current timestep

Public Members

float ***_weightValues**
Contains the values of the weights.

float ***_weightGradient**
Contains the gradients of the weights.

float ***_biasValues**
Contains the values of the bias.

float ***_biasGradient**
Contains the gradients of the bias.

class *korali*::**Logger**
#include <logger.hpp> *Logger* object for *Korali* Modules.

Public Functions

Logger(**const** *std*::string *verbosityLevel*, FILE **file* = stdout)
parametrized constructor for *Korali Logger*

Parameters

- **verbosityLevel** – The verbosity level above which nothing is printed.
- **file** – Output file (default: stdout)

size_t **getVerbosityLevel**(**const** *std*::string *verbosityLevel*)
Gets the numerical value of a verbosity level, given its string value.

Parameters **verbosityLevel** – specifies the verbosity level.

Returns Numerical value corresponding to verbosity level: { SILENT=0, MINIMAL=1, NORMAL=2, DETAILED=3 }

bool **isEnoughVerbosity**(**const** *std*::string *verbosityLevel*)
Checks whether the current verbosity level is enough to authorize the requested level. Serves to filter out non-important messages when low verbosity is chosen.

Parameters **verbosityLevel** – the requested verbosity level

Returns true, if it is enough; false, otherwise.

void **logData**(**const** *std*::string *verbosityLevel*, **const** char **format*, ...)
Outputs raw data to the console file.

Parameters

- **verbosityLevel** – the requested verbosity level.
- **format** – Format string of the data (printf-style)
- **...** – List of arguments for the format string

void **logInfo**(**const** *std*::string *verbosityLevel*, **const** char **format*, ...)
Outputs an information message to the console file.

Parameters

- **verbosityLevel** – the requested verbosity level.
- **format** – Format string of the data (printf-style)
- **...** – List of arguments for the format string

void **logWarning**(**const** *std*::string *verbosityLevel*, **const** char **format*, ...)
Outputs a warning message to the console file.

Parameters

- **verbosityLevel** – the requested verbosity level.
- **format** – Format string of the data (printf-style)
- **...** – List of arguments for the format string

Public Members

size_t **_verbosityLevel**
Global variable that contains the verbosity level for the current *Korali* experiment.

FILE *__outputFile**
Global variable that contains the output file for the current *Korali* experiment.

Public Static Functions

static void **logError**(**const** char **fileName*, **const** int *lineNumber*, **const** char **format*, ...)
Outputs an error message to the console file. Overrides any verbosity level, prints, and exits execution with error.

Parameters

- **fileName** – where the error occurred, given by the **FILE** macro
- **lineNumber** – number where the error occurred, given by the **LINE** macro
- **format** – Format string of the data (printf-style)
- **...** – List of arguments for the format string

class *korali*::*distribution*::*univariate*::**LogNormal** : **public** *korali*::*distribution*::*Univariate*
#include <logNormal.hpp> Class declaration for module: *LogNormal*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_mu**
[Conditional Variable Value] Check: https://en.wikipedia.org/wiki/Log-normal_distribution

std::string **_muConditional**
[Conditional Variable Reference] Check: https://en.wikipedia.org/wiki/Log-normal_distribution

double **_sigma**
[Conditional Variable Value] Check: https://en.wikipedia.org/wiki/Log-normal_distribution

std::string **_sigmaConditional**
[Conditional Variable Reference] Check: https://en.wikipedia.org/wiki/Log-normal_distribution

class *korali*::*neuralNetwork*::*layer*::*recurrent*::**LSTM** : **public** *korali*::*neuralNetwork*::*layer*::*Recurrent*
#include <lstm.hpp> Class declaration for module: *LSTM*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **createForwardPipeline**() **override**
Initializes the layer's internal memory structures for the forward pipeline.

virtual void **createBackwardPipeline**() **override**
Initializes the internal memory structures for the backward pipeline.

virtual void **forwardData**(**const** size_t *t*) **override**
Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

virtual void **backwardData**(**const** size_t *t*) **override**
Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

class *korali*::*solver*::*optimizer*::**MADGRAD** : **public** *korali*::*solver*::*Optimizer*
#include <MADGRAD.hpp> Class declaration for module: *MADGRAD*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **processResult**(double *evaluation*, *std*::vector<double> &*gradient*)
Takes a sample evaluation and its gradient and calculates the next set of parameters.

Parameters

- **evaluation** – The value of the objective function at the current set of parameters
- **gradient** – The gradient of the objective function at the current set of parameters

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

virtual void **runGeneration**() **override**
Runs the current generation.

virtual void **printGenerationBefore**() **override**
Prints solver information before the execution of the current generation.

virtual void **printGenerationAfter**() **override**
Prints solver information after the execution of the current generation.

Public Members

double **_eta**
Learning Rate (Step Size)

double **_weightDecay**
Smoothing factor for variable update.

double **_epsilon**
Term to facilitate numerical stability.

std::vector<double> **_currentVariable**
[Internal Use] Current value of parameters.

double **_scaledLearningRate**
[Internal Use] The learning rate of the current generation.

std::vector<double> **_initialParameter**
[Internal Use] Initial value of parameters.

std::vector<double> **_gradient**
[Internal Use] Gradient of Function with respect to Parameters.

std::vector<double> **_bestEverGradient**
[Internal Use] Gradient of function with respect to Best Ever Variables.

double **_gradientNorm**
[Internal Use] Norm of gradient of function with respect to Parameters.

std::vector<double> **_gradientSum**
[Internal Use] The sum of scaled gradients.

std::vector<double> **_squaredGradientSum**
[Internal Use] The sum of the scaled squares of the gradient.

double **_minGradientNorm**
[Termination Criteria] Specifies the minimal norm for the gradient of function with respect to Parameters.

double **_maxGradientNorm**
[Termination Criteria] Specifies the minimal norm for the gradient of function with respect to Parameters.

class *korali*::*solver*::*sampler*::**MCMC** : **public** *korali*::*solver*::*Sampler*
#include <MCMC.hpp> Class declaration for module: *MCMC*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

double **recursiveAlpha**(double &*denominator*, **const** double *leaderLoglikelihood*, **const** double **loglikelihoods*, size_t *N*) **const**
Calculates recursively acceptance probability. Recursion required for Delayed Rejection.

Parameters

- **denominator** – denominator of quotient (acceptance probability)

- **leaderLoglikelihood** – loglikelihood of current chain leader
- **loglikelihoods** – loglikelihoods of samples obtained after delay
- **N** – rescursion depth

Returns The acceptance probability

void **updateState**()
Updates internal state such as mean and covariance of chain.

void **generateCandidate**(size_t *sampleIdx*)
Generate new sample.

Parameters **sampleIdx** – Id of the sample to generate a candidate for

void **choleskyDecomp**(**const** *std*::vector<double> &*inC*, *std*::vector<double> &*outL*) **const**
Cholesky decomposition of chain covariance matrix.

Parameters

- **inC** – Input matrix
- **outL** – Output lower triangular decomposed matrix

void **finishSample**(size_t *sampleId*)
Process sample after evaluation.

Parameters **sampleId** – Id of the sample to process

virtual void **setInitialConfiguration**() **override**
Configures *MCMC*.

virtual void **finalize**() **override**
Final console output at termination.

virtual void **runGeneration**() **override**
Generate a sample and evaluate it.

virtual void **printGenerationBefore**() **override**
Console Output before generation runs.

virtual void **printGenerationAfter**() **override**
Console output after generation.

Public Members

size_t **_burnIn**
Specifies the number of preliminary *MCMC* steps before samples are being drawn. This may reduce effects from improper initialization.

size_t **_leap**
Generates a Markov Chain containing samples from every 'Leap'-th step. This will increase the overall Chain Length by a factor of 'Leap'.

size_t **_rejectionLevels**
Controls the number of accept-reject stages per *MCMC* step (by default, this value is set 1, for values greater 1 the delajed rejection algorithm is active.

int **_useAdaptiveSampling**
Specifies if covariance matrix of the proposal distribution is calculated from the samples.

size_t **_nonAdaptionPeriod**
: Number of steps (after Burn In steps) during which the initial covariance is used instead of the Chain Covariance. If 0 (default) is specified, this value is calibrated as $5%$ of the Max Chain Length (only relevant for Adaptive Sampling).

double **_chainCovarianceScaling**
: Learning rate of the Chain Covariance (only relevant for Adaptive Sampling).

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**
: [Internal Use] Normal random number generator.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
: [Internal Use] Uniform random number generator.

std::vector<double> **_choleskyDecompositionCovariance**
: [Internal Use] Cholesky Decomposition of Covariance for sampling (using a lower triangular matrix, with rest zeros).

std::vector<double> **_choleskyDecompositionChainCovariance**
: [Internal Use] Chain Cholesky Decomposition of Covariance for sampling (using a lower triangular matrix, with rest zeros).

std::vector<double> **_chainLeader**
: [Internal Use] Variables of the newest sample in the Markov chain.

double **_chainLeaderEvaluation**
: [Internal Use] The logLikelihood of the newest sample in the Markov chain.

std::vector<*std*::vector<double>> **_chainCandidate**
: [Internal Use] Candidate variables to be accepted or rejected after comparison with the Chain Leader.

std::vector<double> **_chainCandidatesEvaluations**
: [Internal Use] The loglikelihoods of the Chain Candidate Parameters.

std::vector<double> **_rejectionAlphas**
: [Internal Use] Placeholder for recursive calculation of delayed rejection schemes.

double **_acceptanceRate**
: [Internal Use] Ratio proposed to accepted samples (including Burn In period).

size_t **_acceptanceCount**
: [Internal Use] Number of accepted samples (including Burn In period).

size_t **_proposedSampleCount**
: [Internal Use] Number of proposed samples.

std::vector<*std*::vector<double>> **_sampleDatabase**
: [Internal Use] Parameters generated by *MCMC* and stored in the database.

std::vector<double> **_sampleEvaluationDatabase**
: [Internal Use] Evaluation associated with the parameters stored in the database.

std::vector<double> **_chainMean**
: [Internal Use] Mean of Markov Chain calculated from samples in Database.

std::vector<double> **_chainCovariancePlaceholder**
: [Internal Use] Placeholder for chain covariance calculation.

std::vector<double> **_chainCovariance**
: [Internal Use] Chain Covariance calculated from samples in Database.

size_t **_chainLength**
: [Internal Use] Current Chain Length (including Burn In and Leaped Samples).

size_t **_maxSamples**
[Termination Criteria] Number of Samples to Generate.

class *korali*::*solver*::*optimizer*::**MOCMAES** : **public** *korali*::*solver*::*Optimizer*
#include <MOCMAES.hpp> Class declaration for module: *MOCMAES*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **prepareGeneration**()
Prepares generation for the next set of evaluations.

void **sampleSingle**(size_t *sampleIdx*)
Evaluates a single sample.

Parameters **sampleIdx** – Index of the sample to evaluate

std::vector<int> **sortSampleIndices**(**const** *std*::vector<*std*::vector<double>> &*values*) **const**
Sort sample indeces based on non-dominance (primary) and contribution and contributing hypervolume (secondary).

Parameters **values** – Values to sort

Returns sorted indices

void **updateDistribution**()
Updates mean and covariance of Gaussian proposal distribution.

void **updateStatistics**()
Update statistics mostly for analysis.

virtual void **setInitialConfiguration**() **override**
Configures CMA-ES.

virtual void **runGeneration**() **override**
Executes sampling & evaluation generation.

virtual void **printGenerationBefore**() **override**
Console Output before generation runs.

virtual void **printGenerationAfter**() **override**
Console output after generation.

virtual void **finalize**() **override**
Final console output at termination.

Public Members

size_t **_populationSize**
Specifies the number of samples to evaluate per generation (preferably $4+3*log(N)$, where N is the number of variables).

size_t **_muValue**
Number of best samples (offspring) advancing to the next generation (by default it is half the *Sample* Count).

double **_evolutionPathAdaptionStrength**
Controls the learning rate of the conjugate evolution path (must be in (0,1], by default this variable is internally calibrated, variable Cc in reference).

double **_covarianceLearningRate**
Controls the learning rate of the covariance matrices (must be in (0,1], by default this variable is internally calibrated, variable Ccov in reference).

double **_targetSuccessRate**
Value that controls the updates of the covariance matrix and the evolution path (must be in (0,1], variable Psucc in reference).

double **_thresholdProbability**
Threshold that defines update scheme for the covariance matrix and the evolution path (must be in (0,1], variable Pthresh in reference).

double **_successLearningRate**
Learning Rate of success rates (must be in (0,1], by default this variable is internally calibrated, variable Cp in reference).

size_t **_numObjectives**
[Internal Use] The number of objective functions to optimize.

korali::*distribution*::*multivariate*::*Normal* ***_multinormalGenerator**
[Internal Use] Multinormal random number generator.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Uniform random number generator.

size_t **_currentNonDominatedSampleCount**
[Internal Use] Number of non dominated samples of current generation.

std::vector<*std*::vector<double>> **_currentValues**
[Internal Use] Objective function values.

std::vector<*std*::vector<double>> **_previousValues**
[Internal Use] Objective function values from previous generation.

std::vector<size_t> **_parentIndex**
[Internal Use] Tracking index of parent samples.

std::vector<*std*::vector<double>> **_parentSamplePopulation**
[Internal Use] *Sample* coordinate information of parents.

std::vector<*std*::vector<double>> **_currentSamplePopulation**
[Internal Use] *Sample* coordinate information.

std::vector<*std*::vector<double>> **_previousSamplePopulation**
[Internal Use] *Sample* coordinate information of previous offsprint.

std::vector<double> **_parentSigma**
[Internal Use] Step size of parent.

std::vector<double> **_currentSigma**
[Internal Use] Determines the step size.

std::vector<double> **_previousSigma**
[Internal Use] Previous step size.

std::vector<*std*::vector<double>> **_parentCovarianceMatrix**
[Internal Use] (Unscaled) covariance matrices of parents.

std::vector<*std*::vector<double>> **_currentCovarianceMatrix**
[Internal Use] (Unscaled) covariance matrices of proposal distributions.

std::vector<*std*::vector<double>> **_previousCovarianceMatrix**
[Internal Use] (Unscaled) covariance matrices of proposal distributions from previous offspring.

std::vector<*std*::vector<double>> **_parentEvolutionPaths**
[Internal Use] Evolution path of parents.

std::vector<*std*::vector<double>> **_currentEvolutionPaths**
[Internal Use] Evolution path of samples.

std::vector<*std*::vector<double>> **_previousEvolutionPaths**
[Internal Use] Evolution path of samples of previous offspring.

std::vector<double> **_parentSuccessProbabilities**
[Internal Use] Smoothed success probabilities of parents.

std::vector<double> **_currentSuccessProbabilities**
[Internal Use] Smoothed success probabilities.

std::vector<double> **_previousSuccessProbabilities**
[Internal Use] Smoothed success probabilities of previous generation.

size_t **_finishedSampleCount**
[Internal Use] Counter of evaluated samples to terminate evaluation.

std::vector<double> **_bestEverValues**
[Internal Use] Best value of each objective.

std::vector<*std*::vector<double>> **_bestEverVariablesVector**
[Internal Use] Samples associated with best ever objective values.

std::vector<double> **_previousBestValues**
[Internal Use] Best objectives from previous generation.

std::vector<*std*::vector<double>> **_previousBestVariablesVector**
[Internal Use] Samples associated with previous best objective values.

std::vector<double> **_currentBestValues**
[Internal Use] Best objectives from current generation.

std::vector<*std*::vector<double>> **_currentBestVariablesVector**
[Internal Use] Samples associated with current best objective values.

std::vector<*std*::vector<double>> **_sampleCollection**
[Internal Use] Candidate pareto optimal samples. Samples will be finalized at termination.

std::vector<*std*::vector<double>> **_sampleValueCollection**
[Internal Use] Model evaluations of pareto candidates.

size_t **_infeasibleSampleCount**
[Internal Use] Keeps count of the number of infeasible samples.

std::vector<double> **_currentBestValueDifferences**
[Internal Use] Value differences of current and previous best values found.

std::vector<double> **_currentBestVariableDifferences**
[Internal Use] L2 norm of previous and current best variable for each objective.

std::vector<double> **_currentMinStandardDeviations**
[Internal Use] Current minimum of any standard devs of a sample.

std::vector<double> **_currentMaxStandardDeviations**
[Internal Use] Current maximum of any standard devs of a sample.

double **_minMaxValueDifferenceThreshold**
[Termination Criteria] Specifies the min max fitness differential between two consecutive generations before stopping execution.

double **_minVariableDifferenceThreshold**
[Termination Criteria] Specifies the min L2 norm of the best samples between two consecutive generations before stopping execution.

double **_minStandardDeviation**
[Termination Criteria] Specifies the minimal standard deviation.

double **_maxStandardDeviation**
[Termination Criteria] Specifies the maximal standard deviation.

class *korali*::**Module**
#include <module.hpp> Represents the basic building block of all *Korali* modules.

Subclassed by *korali::Conduit*, *korali::Distribution*, *korali::Experiment*, *korali::fGradientBasedOptimizer*, *korali::NeuralNetwork*, *korali::neuralNetwork::Layer*, *korali::Problem*, *korali::Solver*

Public Functions

virtual ~Module() = default

virtual void **initialize**()
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **setEngine**(*korali*::*Engine* **engine*)
Sets pointer to the current *Korali* engine, if the module requires running samples.

Parameters **engine** – *Engine* pointer

virtual void **finalize**()
Finalizes *Module*. Deallocates memory and produces outputs.

virtual *std*::string **getType**()
Returns the module type.

Returns A string containing the exact type with which it was created.

virtual bool **checkTermination**()
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*)
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*)
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*)
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**()
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*)
Runs the operation specified in the operation field. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on
- **operation** – An operation accepted by this module or its parents

Returns True, if operation found and executed; false, otherwise.

Public Members

std::string **_type**
Stores the name of the module type selected. Determines which C++ class is constructed upon initialization.

korali::*Experiment* ***_k**
Stores a pointer to its containing experiment.

Public Static Functions

static *Module* ***getModule**(knlohmann::json &*js*, *korali*::*Experiment* **e*)
Instantiates the requested module class given its type and returns its pointer.

Parameters

- **js** – JSON file containing the module's configuration and type.
- **e** – *Korali Experiment* to serve as parent to the module.

Returns Pointer with the newly created module.

class *korali*::*solver*::*integrator*::**MonteCarlo** : **public** *korali*::*solver*::*Integrator*
#include <MonteCarlo.hpp> Class declaration for module: *MonteCarlo*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **launchSample**(size_t *sampleIndex*) **override**
Prepares and launches a sample to be evaluated.

Parameters **sampleIndex** – index of sample to be launched

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

Public Members

size_t **_numberOfSamples**
Specifies the number of randomly generated parameter to evaluate.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Uniform random number generator.

class *korali*::*distribution*::*specific*::**Multinomial** : **public** *korali*::*distribution*::*Specific*
#include <multinomial.hpp> Class declaration for module: *Multinomial*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **getSelections**(*std*::vector<double> &*p*, *std*::vector<unsigned int> &*n*, int *N*)
This function computes a random sample from the multinomial distribution.

Parameters

- **p** – Underlying probability distributions
- **n** – Random sample to draw
- **N** – Number of trials

class *korali*::*distribution*::**Multivariate** : **public** *korali*::*Distribution*
#include <multivariate.hpp> Class declaration for module: *Multivariate*.

Subclassed by *korali::distribution::multivariate::Normal*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **setProperty**(**const** *std*::string &*propertyName*, **const** *std*::vector<double> &*values*) = 0
Updates a specific property with a vector of values.

Parameters

- **propertyName** – The name of the property to update
- **values** – double Numerical values to assign.

virtual void **getDensity**(double *x, double *result, **const** size_t *n*) = 0
Gets the probability density of the distribution at points x.

Parameters

- **x** – points to evaluate
- **result** – P(x) at the given points
- **n** – number of points

virtual void **getLogDensity**(double *x, double *result, **const** size_t *n*) = 0
Gets Log probability density of the distribution at points x.

Parameters

- **x** – points to evaluate
- **result** – log(P(x)) at the given points

- **n** – number of points

virtual void **getRandomVector**(double *x, **const** size_t n) = 0
Draws and returns a random number vector from the distribution.

Parameters

- **x** – Random real number vector.
- **n** – Vector size

class *korali*::*solver*::*sampler*::**Nested** : **public** *korali*::*solver*::*Sampler*
#include <Nested.hpp> Class declaration for module: *Nested*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **setInitialConfiguration**() **override**
Configures *Sampler*.

virtual void **runGeneration**() **override**
Main solver loop.

virtual void **printGenerationBefore**() **override**
Console Output before generation runs.

virtual void **printGenerationAfter**() **override**
Console output after generation.

virtual void **finalize**() **override**
Final console output at termination.

Public Members

size_t **_numberLivePoints**
Number of live samples.

size_t **_batchSize**
Number of samples to discard and replace per generation, maximal number of parallel sample evaluation.

int **_addLivePoints**
Add live points to dead points.

std::string **_resamplingMethod**
Method to generate new candidates (can be set to either 'Box' or 'Ellipse', 'Multi Ellipse').

size_t **_proposalUpdateFrequency**
Frequency of resampling distribution update (e.g. ellipse rescaling for Ellipse).

double **_ellipsoidalScaling**
Scaling factor of ellipsoidal (only relevant for 'Ellipse' and 'Multi Ellipse' proposal).

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Uniform random number generator.

korali::*distribution*::*univariate*::*Normal* ***_normalGenerator**
[Internal Use] Gaussian random number generator.

korali::*distribution*::*multivariate*::*Normal* ***_multivariateGenerator**
[Internal Use] Random number generator with a multivariate normal distribution.

size_t **_acceptedSamples**
[Internal Use] Number of accepted samples.

size_t **_generatedSamples**
[Internal Use] Number of generated samples (after initialization).

double **_logEvidence**
[Internal Use] Accumulated LogEvidence.

double **_logEvidenceVar**
[Internal Use] Estimated accumulated variance of log evidence.

double **_logVolume**
[Internal Use] Remaining Prior Mass.

double **_boundLogVolume**
[Internal Use] Volume within bounds.

size_t **_lastAccepted**
[Internal Use] Number of generations past since a sample has been accepted.

size_t **_nextUpdate**
[Internal Use] Next time when bounds are being updated.

double **_information**
[Internal Use] Accumulated information.

double **_lStar**
[Internal Use] Likelihood constraint for sample acceptance.

double **_lStarOld**
[Internal Use] Previous likelihood constraint.

double **_logWeight**
[Internal Use] Log increment of evidence.

double **_expectedLogShrinkage**
[Internal Use] Expected volume shrinkage per sample.

double **_maxEvaluation**
[Internal Use] Largest sum of loglikelihood and logprior in live sample set.

double **_remainingLogEvidence**
[Internal Use] Estimated remaining log evidence.

double **_logEvidenceDifference**
[Internal Use] Difference of current and remaining log evidence.

double **_effectiveSampleSize**
[Internal Use] Number of effective Samples estimate.

double **_sumLogWeights**
[Internal Use] Sum of log weights in sample database.

double **_sumSquareLogWeights**
[Internal Use] Sum of squared log weights in sample database.

std::vector<double> **_priorLowerBound**
[Internal Use] Lower bound of unfirom prior.

std::vector<double> **_priorWidth**
[Internal Use] Width of uniform prior.

std::vector<*std*::vector<double>> **_candidates**
[Internal Use] *Sample* candidates to be evaluated in current generation.

std::vector<double> **_candidateLogLikelihoods**
[Internal Use] Loglikelihood evaluations of candidates.

std::vector<double> **_candidateLogPriors**
[Internal Use] The logpriors of the candidates.

std::vector<double> **_candidateLogPriorWeights**
[Internal Use] The logprior weights of the candidates.

std::vector<*std*::vector<double>> **_liveSamples**
[Internal Use] Samples to be processed and replaced in ascending order.

std::vector<double> **_liveLogLikelihoods**
[Internal Use] Loglikelihood evaluations of live samples.

std::vector<double> **_liveLogPriors**
[Internal Use] Logprior evaluations of live samples.

std::vector<double> **_liveLogPriorWeights**
[Internal Use] Logprior weights of live samples.

std::vector<size_t> **_liveSamplesRank**
[Internal Use] Ascending ranking of live samples (sorted based on likelihood and logprior weight).

size_t **_numberDeadSamples**
[Internal Use] Number of dead samples, which have been removed from the live samples.

std::vector<*std*::vector<double>> **_deadSamples**
[Internal Use] Dead samples stored in database.

std::vector<double> **_deadLogLikelihoods**
[Internal Use] Loglikelihood evaluations of dead samples.

std::vector<double> **_deadLogPriors**
[Internal Use] Logprior evaluations associated with dead samples.

std::vector<double> **_deadLogPriorWeights**
[Internal Use] Logprior weights associated with dead samples.

std::vector<double> **_deadLogWeights**
[Internal Use] Log weight (Priormass x Likelihood) of dead samples.

std::vector<double> **_covarianceMatrix**
[Internal Use] *Sample* covariance of the live samples.

double **_logDomainSize**
[Internal Use] Log of domain volumne given by uniform prior distribution.

std::vector<double> **_domainMean**
[Internal Use] Mean of the domain occupied by live samples.

std::vector<double> **_boxLowerBound**
[Internal Use] Lower bound of box constraint (only relevant for 'Box' resampling method).

std::vector<double> **_boxUpperBound**
[Internal Use] Upper bound of box constraint (only relevant for 'Box' resampling method).

std::vector<*std*::vector<double>> **_ellipseAxes**
[Internal Use] Axes of bounding ellipse (only relevant for 'Ellipse' resampling method).

double **_minLogEvidenceDelta**
[Termination Criteria] Minimal difference between estimated remaining log evidence and current logevidence.

size_t **_maxEffectiveSampleSize**
[Termination Criteria] Estimated maximal evidence gain smaller than accumulated evidence by given factor.

size_t **_maxLogLikelihood**
[Termination Criteria] Terminates if loglikelihood of sample removed from live set exceeds given value.

Private Functions

void **runFirstGeneration**()

void **updateBounds**()

void **priorTransform**(*std*::vector<double> &*sample*) **const**

void **generateCandidates**()

void **generateCandidatesFromBox**()
Generate new samples uniformly in Box.

void **generateSampleFromEllipse**(**const** *ellipse_t* &*ellipse*, *std*::vector<double> &*sample*) **const**
Generates a sample uniformly in Ellipse.

Parameters

- **ellipse** – Bounding ellipsoid from which to sample.
- **sample** – Generated sample.

void **generateCandidatesFromEllipse**()
Generate new samples uniformly in Ellipse.

void **generateCandidatesFromMultiEllipse**()
Generate new samples uniformly from multiple Ellipses.

bool **processGeneration**()

double **logPriorWeight**(*std*::vector<double> &*sample*)

void **setBoundsVolume**()

void **consumeLiveSamples**()

void **updateBox**()

void **sortLiveSamplesAscending**()

void **updateDeadSamples**(size_t *sampleIdx*)

void **generatePosterior**()

double **l2distance**(**const** *std*::vector<double> &*sampleOne*, **const** *std*::vector<double> &*sampleTwo*) **const**

bool **updateEllipse**(*ellipse_t* &*ellipse*) **const**

void **updateMultiEllipse**()

void **initEllipseVector**()

bool **kmeansClustering**(**const** *ellipse_t* &*parent*, size_t *maxIter*, *ellipse_t* &*childOne*, *ellipse_t* &*childTwo*) **const**

void **updateEllipseMean**(*ellipse_t* &*ellipse*) **const**

bool **updateEllipseCov**(*ellipse_t* &*ellipse*) **const**

bool **updateEllipseVolume**(*ellipse_t* &*ellipse*) **const**

double **mahalanobisDistance**(**const** *std*::vector<double> &*sample*, **const** *ellipse_t* &*ellipse*) **const**

void **updateEffectiveSamples**()

bool **insideUnitCube**(**const** *std*::vector<double> &*sample*) **const**

Private Members

size_t **_shuffleSeed**

std::vector<*ellipse_t*> **_ellipseVector**

class *korali*::**NeuralNetwork** : **public** *korali*::*Module*
#include <neuralNetwork.hpp> Class declaration for module: *NeuralNetwork*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

std::vector<float> **generateInitialHyperparameters**()
Generates the initial values for the hyperparameters.

Returns The generated hyperparameters

void **setHyperparameters**(**const** *std*::vector<float> &*hyperparameters*)
Updates the values of weights, biases configuration to the NN.

Parameters **hyperparameters** – The input hyperparameters

std::vector<float> **getHyperparameters**()
Gets the values of weights and biases configuration to the NN.

Returns The hyperparameters of the NN

void **forward**(**const** *std*::vector<*std*::vector<*std*::vector<float>>> &*inputValues*)
Forward-propagates the input values through the network.

Parameters **inputValues** – The input values. Format: TxNxIC (T: Time steps, N: Mini-batch, IC: Input channels).

void **backward**(**const** *std*::vector<*std*::vector<float>> &*outputGradients*)
Backward-propagates the gradients through the network.

Parameters **outputGradients** – Output gradients. Format: NxOC (N: Mini-batch size, OC: Output channels).

size_t **getBatchSizeIdx**(**const** size_t *batchSize*)
Returns the pipeline index corresponding to the batch size requested.

Parameters **batchSize** – Size of the batch to request

Returns Pipeline Id correspoding to the batch size

std::vector<*std*::vector<float>> &**getOutputValues**(**const** size_t *batchSize*)
Returns a reference to the output values corresponding to the batch size's pipeline.

Parameters **batchSize** – Size of the batch to request

Returns Reference to the output values

std::vector<*std*::vector<float>> &**getInputGradients**(**const** size_t *batchSize*)
Returns a reference to the input gradients corresponding to the batch size's pipeline.

Parameters **batchSize** – Size of the batch to request

Returns Reference to the input gradients

std::vector<float> &**getHyperparameterGradients**(**const** size_t *batchSize*)
Returns a reference to the hyperparameter gradients corresponding to the batch size's pipeline.

Parameters **batchSize** – Size of the batch to request

Returns Reference hyperparameter gradients

NeuralNetwork()
Creator that sets initialized flag to false.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

Public Members

std::string **_engine**
Specifies which Neural Network backend engine to use.

std::string **_mode**
Specifies the execution mode of the Neural Network.

knlohmann::json **_layers**
Complete description of the NN's layers.

size_t **_timestepCount**
Provides the sequence length for the input/output data.

std::vector<size_t> **_batchSizes**
Specifies the batch sizes.

float **_currentTrainingLoss**
[Internal Use] Current value of the training loss.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Uniform random number generator for setting the initial value of the weights and biases.

std::vector<*std*::vector<*layerPipeline_t*>> **_pipelines**
Layer pipelines, one per threadCount * BatchSize combination. These are all replicas of the user-defined layers that share the same hyperparameter space.

bool **_isInitialized**
Flag to make sure the NN is initialized before creating.

size_t **_hyperparameterCount**
Number of NN hyperparameters (weights/bias)

class *korali*::*distribution*::*univariate*::**Normal** : **public** *korali*::*distribution*::*Univariate*
#include <normal.hpp> Class declaration for module: *Normal*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_mean**
[Conditional Variable Value] The mean of the *Normal* (Gaussian) distribution.

std::string **_meanConditional**
[Conditional Variable Reference] The mean of the *Normal* (Gaussian) distribution.

double **_standardDeviation**
[Conditional Variable Value] The standard deviation of the *Normal* (Gaussian) distribution.

std::string **_standardDeviationConditional**
[Conditional Variable Reference] The standard deviation of the *Normal* (Gaussian) distribution.

Private Members

double **_normalization**

double **_logNormalization**

class *korali*::*distribution*::*multivariate*::**Normal** : **public** *korali*::*distribution*::*Multivariate*
#include <normal.hpp> Class declaration for module: *Normal*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual void **setProperty**(**const** *std*::string &*propertyName*, **const** *std*::vector<double> &*values*) **override**
Updates a specific property with a vector of values.

Parameters

- **propertyName** – The name of the property to update
- **values** – double Numerical values to assign.

virtual void **getDensity**(double **x*, double **result*, **const** size_t *n*) **override**
Gets the probability density of the distribution at points x.

Parameters

- **x** – points to evaluate
- **result** – P(x) at the given points
- **n** – number of points

virtual void **getLogDensity**(double **x*, double **result*, **const** size_t *n*) **override**
Gets Log probability density of the distribution at points x.

Parameters

- **x** – points to evaluate
- **result** – log(P(x)) at the given points
- **n** – number of points

virtual void **getRandomVector**(double *x, **const** size_t n) **override**
Draws and returns a random number vector from the distribution.

Parameters

- **x** – Random real number vector.
- **n** – Vector size

Public Members

std::vector<double> **_meanVector**
Means of the variables.

std::vector<double> **_sigma**
Cholesky Decomposition of the covariance matrix.

std::vector<double> **_workVector**
[Internal Use] Auxiliary work vector.

Private Members

gsl_matrix_view **_sigma_view**
Temporal storage for covariance matrix.

gsl_vector_view **_mean_view**
Temporal storage for variable means.

gsl_vector_view **_work_view**
Temporal storage for work.

class *korali*::*problem*::**Optimization** : **public** *korali*::*Problem*
#include <optimization.hpp> Class declaration for module: *Optimization*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

void **evaluate**(*korali*::*Sample* &*sample*)
Evaluates a single objective, given a set of parameters.

Parameters **sample** – A sample to process

void **evaluateMultiple**(*korali*::*Sample* &*sample*)
Evaluates multiple objectives, given a set of parameters.

Parameters **sample** – A sample to process

void **evaluateConstraints**(*korali*::*Sample* &*sample*)
Evaluates whether at least one of constraints have been met.

Parameters **sample** – A *Korali Sample*

void **evaluateWithGradients**(*korali*::*Sample* &*sample*)
Evaluates the F(x) and Gradient(x) of a sample, given a set of parameters.

Parameters **sample** – A sample to process

Public Members

size_t **_numObjectives**
Number of return values to expect from objective function.

std::uint64_t **_objectiveFunction**
Stores the function to evaluate.

std::vector<*std*::uint64_t> **_constraints**
Stores constraints to the objective function.

int **_hasDiscreteVariables**
[Internal Use] Flag indicating if at least one of the variables is discrete.

class *korali*::*solver*::**Optimizer** : **public** *korali*::*Solver*
#include <optimizer.hpp> Class declaration for module: *Optimizer*.

Subclassed by *korali::solver::optimizer::AdaBelief*, *korali::solver::optimizer::Adam*, *korali::solver::optimizer::CMAES*, *korali::solver::optimizer::DEA*, *korali::solver::optimizer::GridSearch*, *korali::solver::optimizer::MADGRAD*, *korali::solver::optimizer::MOCMAES*, *korali::solver::optimizer::Rprop*

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

bool **isSampleFeasible**(**const** *std*::vector<double> &*sample*)
Checks whether the proposed sample can be optimized.

Parameters **sample** – A *Korali Sample*

Returns True, if feasible; false, otherwise.

Public Members

double **_currentBestValue**
[Internal Use] Best model evaluation from current generation.

double **_previousBestValue**
[Internal Use] Best model evaluation from previous generation.

double **_bestEverValue**
[Internal Use] Best ever model evaluation.

std::vector<double> **_bestEverVariables**
[Internal Use] Variables associated to best ever value found.

size_t **_infeasibleSampleCount**
[Internal Use] Keeps count of the number of infeasible samples.

double **_maxValue**
[Termination Criteria] Specifies the maximum target fitness to stop maximization.

double **_minValueDifferenceThreshold**
[Termination Criteria] Specifies the minimum fitness differential between two consecutive generations before stopping execution.

size_t **_maxInfeasibleResamplings**
[Termination Criteria] Maximum number of resamplings per candidate per generation if sample is outside of Lower and Upper Bound.

class *korali*::*neuralNetwork*::*layer*::**Output** : **public** *korali*::*neuralNetwork*::*Layer*
#include <output.hpp> Class declaration for module: *Output*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **createForwardPipeline**() **override**
Initializes the layer's internal memory structures for the forward pipeline.

virtual void **createBackwardPipeline**() **override**
Initializes the internal memory structures for the backward pipeline.

virtual void **forwardData**(**const** size_t *t*) **override**
Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

virtual void **backwardData**(**const** size_t *t*) **override**
Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

Public Members

std::vector<*std*::string> **_transformationMask**
Indicates a transformation to be performed to the output at the last layer of the neural network. [Order of application on forward propagation: 1/3].

std::vector<float> **_scale**
Gives a scaling factor for each of the output values of the NN. [Order of application on forward propagation 2/3].

std::vector<float> **_shift**
Shifts the output of the NN by the values given. [Order of application on forward propagation 3/3].

float ***_srcOutputValues**
Contains the original output, before preprocessing.

float ***_dstOutputGradients**
Contains the postprocessed gradients.

std::vector<*transformation_t*> **_transformationVector**
Contains the description of the transformation to apply to each output element.

struct *korali*::**ParsedReactionString**
#include <reactionParser.hpp> Struct of reaction details constructed from reaction equation.

Public Members

std::vector<*std*::string> **reactantNames**
Vector containing all reactants in the reaction.

std::vector<int> **reactantSCs**
The stoichiometries asssociated to the reactants.

std::vector<*std*::string> **productNames**
Vector containing all products.

std::vector<int> **productSCs**
The stoichiometries asssociated to the products.

std::vector<bool> **isReactantReservoir**
Boolean vector indicating if a reactant is a reservoir and remains unchanged.

class *korali*::*distribution*::*univariate*::**Poisson** : **public** *korali*::*distribution*::*Univariate*
#include <poisson.hpp> Class declaration for module: *Poisson*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_mean**
[Conditional Variable Value] The mean and variance of the distribution.

std::string **_meanConditional**
[Conditional Variable Reference] The mean and variance of the distribution.

struct *korali*::*solver*::**policy_t**
#include <agent.hpp> Structure to store policy information.

Public Members

float **stateValue**
Contains state value (V) estimation for the given state / policy combination.

std::vector<float> **distributionParameters**
Contains the parameters that define the policy distribution used to produced the action. For continuous policies, it depends on the distribution selected. For discrete policies, it contains the categorical probability of every action.

size_t **actionIndex**
[Discrete] Stores the index of the selected experience

std::vector<float> **actionProbabilities**
[Discrete] Stores the action probabilities of the categorial distribution.

std::vector<size_t> **availableActions**
[Discrete] Flags the actions that are available at the current state.

std::vector<float> **unboundedAction**
[Continuous] Stores the Unbounded Actions of the Squashed Normal Policy *Distribution*

class *korali*::*neuralNetwork*::*layer*::**Pooling** : **public** *korali*::*neuralNetwork*::*Layer*
#include <pooling.hpp> Class declaration for module: *Pooling*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **createForwardPipeline**() **override**
Initializes the layer's internal memory structures for the forward pipeline.

virtual void **createBackwardPipeline**() **override**
Initializes the internal memory structures for the backward pipeline.

virtual void **forwardData**(**const** size_t *t*) **override**
Performs the forward propagation of the Wx+b operations.

Parameters **t** – Indicates the current timestep

virtual void **backwardData**(**const** size_t *t*) **override**
Performs the backward propagation of the data.

Parameters **t** – Indicates the current timestep

Public Members

std::string **_function**
Indicates the pooling function to apply.

ssize_t **_imageHeight**
Height of the incoming 2D image.

ssize_t **_imageWidth**
Width of the incoming 2D image.

ssize_t **_kernelHeight**
Height of the incoming 2D image.

ssize_t **_kernelWidth**
Width of the incoming 2D image.

ssize_t **_verticalStride**
Strides for the image on the vertical dimension.

ssize_t **_horizontalStride**
Strides for the image on the horizontal dimension.

ssize_t **_paddingLeft**
Paddings for the image left side.

ssize_t **_paddingRight**
Paddings for the image right side.

ssize_t **_paddingTop**
Paddings for the image top side.

ssize_t **_paddingBottom**
Paddings for the image Bottom side.

ssize_t **N**
Pre-calculated value for Mini-Batch Size.

ssize_t **IC**
Pre-calculated value for *Input* Channels.

ssize_t **IH**
Pre-calculated value for *Input* Image Height.

ssize_t **IW**
Pre-calculated value for *Input* Image Width.

ssize_t **OC**
Pre-calculated value for *Output* Channels.

ssize_t **OH**
Pre-calculated value for *Output* Image Height.

ssize_t **OW**
Pre-calculated value for *Output* Image Width.

ssize_t **KH**
Pre-calculated value for Kernel Image Height.

ssize_t **KW**
Pre-calculated value for Kernel Image Width.

ssize_t **PL**
Pre-calculated values for padding left.

ssize_t **PR**
Pre-calculated values for padding right.

ssize_t **PT**
Pre-calculated values for padding top.

ssize_t **PB**
Pre-calculated values for padding bottom.

ssize_t **SH**
Pre-calculated values for horizontal stride.

ssize_t **SV**
Pre-calculated values for vertical stride.

class *korali*::**Problem** : **public** *korali*::*Module*
#include <problem.hpp> Class declaration for module: *Problem*.

Subclassed by *korali::problem::Bayesian*, *korali::problem::Design*, *korali::problem::Hierarchical*, *korali::problem::Integration*, *korali::problem::Optimization*, *korali::problem::Propagation*, *korali::problem::Reaction*, *korali::problem::ReinforcementLearning*, *korali::problem::Sampling*, *korali::problem::SupervisedLearning*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

class *korali*::*problem*::**Propagation** : **public** *korali*::*Problem*
#include <propagation.hpp> Class declaration for module: *Propagation*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

void **execute**(*korali*::*Sample* &*sample*)
Produces an evaluation of the model, storing it in a file.

Parameters **sample** – A *Korali Sample*

Public Members

std::uint64_t **_executionModel**
Stores the function to evaluate.

size_t **_numberOfSamples**
Number of samples to draw from Prior distribution.

class *korali*::*problem*::*hierarchical*::**Psi** : **public** *korali*::*problem*::*Hierarchical*
#include <psi.hpp> Class declaration for module: *Psi*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **updateConditionalPriors**(*korali*::*Sample* &*sample*)
Updates the distribution parameters for the conditional priors, given variable values in the sample.

Parameters **sample** – A *Korali Sample*

virtual void **evaluateLogLikelihood**(*korali*::*Sample* &*sample*) **override**
Evaluates the log likelihood of the given sample, and stores it in sample["Log Likelihood"].

Parameters **sample** – A *Korali Sample*

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

Public Members

std::vector<knlohmann::json> **_subExperiments**
Provides results from previous *Bayesian* Inference sampling experiments.

std::vector<*std*::string> **_conditionalPriors**
List of conditional priors to use in the hierarchical problem.

std::vector<size_t> **_conditionalPriorIndexes**
Stores the indexes of conditional priors to *Experiment* variables.

Private Members

size_t **_subProblemsCount**
Stores the number of subproblems.

size_t **_subProblemsVariablesCount**
Stores the number of variables in the subproblems (all must be the same)

std::vector<*std*::vector<*std*::vector<double>>> **_subProblemsSampleCoordinates**
Stores the sample coordinates of all the subproblems.

std::vector<*std*::vector<double>> **_subProblemsSampleLogLikelihoods**
Stores the sample logLikelihoods of all the subproblems.

std::vector<*std*::vector<double>> **_subProblemsSampleLogPriors**
Stores the sample logPriors of all the subproblems.

std::vector<*conditionalPriorInfo*> **_conditionalPriorInfos**
Stores the precomputed conditional prior information, for performance.

class *korali*::*solver*::*integrator*::**Quadrature** : **public** *korali*::*solver*::*Integrator*
#include <Quadrature.hpp> Class declaration for module: *Quadrature*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **launchSample**(size_t *sampleIndex*) **override**
Prepares and launches a sample to be evaluated.

Parameters **sampleIndex** – index of sample to be launched

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

Public Members

std::string **_method**
The name of the quadrature rule.

std::vector<size_t> **_indicesHelper**
[Internal Use] Holds helper to calculate cartesian indices from linear index.

class *korali*::*problem*::**Reaction** : **public** *korali*::*Problem*
#include <reaction.hpp> Class declaration for module: *Reaction*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

double **computePropensity**(size_t *reactionIndex*, **const** *std*::vector<int> &*reactantNumbers*) **const**
Compute the propensity of a reaction.

Parameters

- **reactionIndex** – The index of the reaction
- **reactantNumbers** – Current number of reactants in simulation

Returns the propensity of reaction

double **computeGradPropensity**(size_t *reactionIndex*, **const** *std*::vector<int> &*reactantNumbers*, size_t *dI*) **const**
Compute the gradient of the propensity of a reaction wrt reactants.

Parameters

- **reactionIndex** – The index of the reaction
- **reactantNumbers** – Current number of reactants in simulation
- **dI** – reactantindex for gradient computation

Returns the gradient of the propensity of reaction

double **computeF**(size_t *reactionIndex*, size_t *otherReactionIndex*, **const** *std*::vector<int> &*reactantNumbers*)
Computes F value, the sum of weighted differentials of two reactions.

Parameters

- **reactionIndex** – The index of the reaction
- **otherReactionIndex** – the index of the second reaction, to access the state change values
- **reactantNumbers** – Current number of reactants in simulation

Returns value F

double **calculateMaximumAllowedFirings**(size_t *reactionIndex*, **const** *std*::vector<int> &*reactantNumbers*) **const**

Calculate the maximum allowed firings of a reactant in a reaction.

Parameters

- **reactionIndex** – The index of the reaction
- **reactantNumbers** – Current number of reactants in simulation

Returns maximum allowed firings of reaction

void **setStateChange**(size_t *numReactants*)

Initializes the state change matrix.

Parameters **numReactants** – number of reactants in reaction experiment

void **applyChanges**(size_t *reactionIndex*, *std*::vector<int> &*reactantNumbers*, int *numFirings* = 1) **const**

Apply changes to reactants based on reaction.

Parameters

- **reactionIndex** – The index of the reaction
- **reactantNumbers** – Current number of reactants in simulation
- **numFirings** – Number of repeated firings of reaction

Public Members

knlohmann::json **_reactions**

[Internal Use] Complete description of all reactions.

Class for the reaction problem type based on the implementation by Luca Amoudruz https://github.com/amlucas/SSM

std::map<*std*::string, int> **_reactantNameToIndexMap**

[Internal Use] Maps the reactants name to an internal index.

std::vector<int> **_initialReactantNumbers**

[Internal Use] Maps the reactants name to an internal index.

std::vector<*std*::vector<int>> **_stateChange**

[Internal Use] TODO

std::vector<*reaction_t*> **_reactionVector**

Container for all reactions.

struct *korali*::*problem*::**reaction_t**

#include <reaction.hpp> Structure to store reaction information.

Public Functions

inline reaction_t (double *rate*, *std*::vector<int> *reactantIds*, *std*::vector<int> *reactantSCs*, *std*::vector<int> *productIds*, *std*::vector<int> *productSCs*, *std*::vector<bool> *isReactantReservoir*)
Constructor for type *reaction_t*.

Parameters

- **rate** – the rate of the reaction
- **reactantIds** – ids of reactants
- **reactantSCs** – stoichiometry coefficients of reactants
- **productIds** – ids of products
- **productSCs** – stoichiometry coefficients of products
- **isReactantReservoir** – indicators if reactant is reservoir

Public Members

double **rate**
The rate of the reaction.

std::vector<int> **reactantIds**
Ids of the reactants.

std::vector<int> **reactantStoichiometries**
Stoichiometries of the reactants.

std::vector<int> **productIds**
Ids of the products of the reaction.

std::vector<int> **productStoichiometries**
Stoichiometries of the products.

std::vector<bool> **isReactantReservoir** = {}
Flag to declare reactants as reservois (remain unchanged).

class *korali*::*neuralNetwork*::*layer*::**Recurrent** : **public** *korali*::*neuralNetwork*::*Layer*
#include <recurrent.hpp> Class declaration for module: *Recurrent*.

Subclassed by *korali::neuralNetwork::layer::recurrent::GRU*, *korali::neuralNetwork::layer::recurrent::LSTM*

Public Functions

virtual void **getConfiguration** (knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration** (knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults** (knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **createHyperparameterMemory**() **override**
Initializes the layer's internal memory structures for hyperparameter storage.

virtual void **backwardHyperparameters**(**const** size_t *t*) **override**
Calculates the gradients of layer hyperparameters.

Parameters **t** – Indicates the current timestep

virtual void **createBackwardPipeline**() **override**
Initializes the internal memory structures for the backward pipeline.

virtual void **copyHyperparameterPointers**(*Layer* **dstLayer*) **override**
Replicates the pointers for the current layer onto a destination layer.

Parameters **dstLayer** – The destination layer onto which to copy the pointers

virtual *std*::vector<float> **generateInitialHyperparameters**() **override**
Generates the initial weight/bias hyperparameters for the layer.

Returns The initial hyperparameters

virtual void **setHyperparameters**(**const** float **hyperparameters*) **override**
Updates layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (*Input*) Pointer to read the hyperparameters from.

virtual void **getHyperparameters**(float **hyperparameters*) **override**
Gets layer's hyperparameters (e.g., weights and biases)

Parameters **hyperparameters** – (*Output*) Pointer to write the hyperparameters to.

virtual void **getHyperparameterGradients**(float **gradient*) **override**
Gets the gradients of the layer's output wrt to is hyperparameters (e.g., weights and biases)

Parameters **gradient** – (*Output*) Pointer to write the hyperparameter gradients to.

Public Members

size_t **_depth**
The number of copies of this layer. This has a better performance than just defining many of these layers manually since it is optimized by the underlying engine.

size_t **_gateCount**
Indicates the number of recurrent gates (depends on the architecture)

class *korali*::*problem*::*bayesian*::**Reference** : **public** *korali*::*problem*::*Bayesian*
#include <reference.hpp> Class declaration for module: *Reference*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **evaluateLoglikelihood**(*korali*::*Sample* &*sample*) **override**
Evaluates the log likelihood of the given sample, and stores it in sample["Log Likelihood"].

Parameters **sample** – A *Korali Sample*

virtual void **evaluateLoglikelihoodGradient**(*korali*::*Sample* &*sample*) **override**
Evaluates the gradient of the logLikelikood w.r.t. to the variables, and stores it in sample["logLikelihood Gradient"].

Parameters **sample** – A *Korali Sample*

virtual void **evaluateLogLikelihoodHessian**(*korali*::*Sample* &*sample*) **override**
Evaluates the gradient of the logLikelikood w.r.t. to the variables, and stores it in sample["logLikelihood Hessian"].

Parameters **sample** – A *Korali Sample*

virtual void **evaluateFisherInformation**(*korali*::*Sample* &*sample*) **override**
Evaluates the empirical Fisher information.

Parameters **sample** – A *Korali Sample*

Public Members

std::uint64_t **_computationalModel**
Stores the computational model. It should the evaluation of the model at the given reference data points.

std::vector<double> **_referenceData**
Reference data required to calculate likelihood. Model evaluations are compared against these data.

std::string **_likelihoodModel**
Specifies the likelihood model to approximate the reference data to.

Private Functions

double **compute_normalized_sse**(*std*::vector<double> *f*, *std*::vector<double> *g*, *std*::vector<double> *y*)
Precomputes the square distance between two vectors (f and y) of the same size normalized by a third vector (g)

Parameters

- **f** – Vector f
- **g** – Vector g, the normalization vector
- **y** – Vector y

Returns Normalized square distance of the vectors

void **loglikelihoodNormal**(*korali*::*Sample* &*sample*)
An implementation of the normal likelihood y~N(f,g), where f ang g are provided by the user.

Parameters **sample** – A *Korali Sample*

void **loglikelihoodPositiveNormal**(*korali*::*Sample* &*sample*)
An implementation of the normal likelihood y~N(f,g) truncated at zero, where f ang g are provided by the user.

Parameters **sample** – A *Korali Sample*

void **loglikelihoodStudentT**(*korali*::*Sample* &*sample*)
An implementation of the student's t loglikelihood y~T(v), where v>0 (degrees of freedom) is provided by the user.

Parameters **sample** – A *Korali Sample*

void **loglikelihoodPositiveStudentT**(*Sample* &*sample*)
An implementation of the student's t loglikelihood y~T(v) truncated at zero, where v>0 (degrees of freedom) is provided by the user.

Parameters **sample** – A *Korali Sample*

void **loglikelihoodPoisson**(*korali*::*Sample* &*sample*)
Poisson likelihood parametrized by mean.

Parameters **sample** – A *Korali Sample*

void **loglikelihoodGeometric**(*korali*::*Sample* &*sample*)
Geometric likelihood parametrized by mean. Parametrization of number of trials before success used.

Parameters **sample** – A *Korali Sample*

void **loglikelihoodNegativeBinomial**(*korali*::*Sample* &*sample*)
Negative Binomial likelihood parametrized by mean and dispersion.

Parameters **sample** – A *Korali Sample*

void **gradientLoglikelihoodNormal**(*korali*::*Sample* &*sample*)
Calculates the gradient of the Normal loglikelihood model.

Parameters **sample** – A *Korali Sample*

void **gradientLoglikelihoodPositiveNormal**(*korali*::*Sample* &*sample*)
Calculates the gradient of the Positive Normal (truncated at 0) loglikelihood model.

Parameters **sample** – A *Korali Sample*

void **gradientLoglikelihoodNegativeBinomial** (*korali*::*Sample* &*sample*)
Calculates the gradient of the Negative Binomial loglikelihood model.

Parameters **sample** – A *Korali Sample*

void **hessianLogLikelihoodNormal** (*korali*::*Sample* &*sample*)
Calculates the Hessian of the Normal logLikelihood model.

Parameters **sample** – A *Korali Sample*

void **hessianLogLikelihoodPositiveNormal** (*korali*::*Sample* &*sample*)
Calculates the Hessian of the Positive Normal logLikelihood model.

Parameters **sample** – A *Korali Sample*

void **hessianLogLikelihoodNegativeBinomial** (*korali*::*Sample* &*sample*)
Calculates the Hessian of the Negative Binomial logLikelihood model.

Parameters **sample** – A *Korali Sample*

void **fisherInformationLoglikelihoodNormal** (*korali*::*Sample* &*sample*)
Calculates the Fisher information matrix of the Normal likelihood model.

Parameters **sample** – A *Korali Sample*

void **fisherInformationLoglikelihoodPositiveNormal** (*korali*::*Sample* &*sample*)
Calculates the Fisher information matrix of the Positive Normal (truncated at 0) likelihood model.

Parameters **sample** – A *Korali Sample*

void **fisherInformationLoglikelihoodNegativeBinomial** (*korali*::*Sample* &*sample*)
Calculates the Fisher information matrix of the Negative Binomial likelihood model.

Parameters **sample** – A *Korali Sample*

Private Members

const double **_log2pi** = 1.837877066409345339081937709124 76

class *korali*::*problem*::**ReinforcementLearning** : **public** *korali*::*Problem*
#include <reinforcementLearning.hpp> Class declaration for module: *ReinforcementLearning*.

Subclassed by *korali::problem::reinforcementLearning::Continuous*, *korali::problem::reinforcementLearning::Discrete*

Public Functions

virtual void **getConfiguration** (knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration** (knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults** (knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

void **runTrainingEpisode**(*korali*::*Sample* &*agent*)
Runs an episode of the agent within the environment with actions produced by the policy + exploratory noise. If the reward exceeds the threshold, it also runs testing episodes.

Parameters **agent** – *Sample* containing current agent/state information.

void **runTestingEpisode**(*korali*::*Sample* &*agent*)
Runs an episode of the agent within the environment with actions produced by the policy only.

Parameters **agent** – *Sample* containing current agent/state information.

void **initializeEnvironment**(*korali*::*Sample* &*agent*)
Initializes the environment and agent configuration.

Parameters **agent** – *Sample* containing current agent/state information.

void **finalizeEnvironment**()
Finalizes the environemnt (frees resources)

void **runEnvironment**(*Sample* &*agent*)
Runs/resumes the execution of the environment.

Parameters **agent** – *Sample* containing current agent/state information.

void **requestNewPolicy**(*Sample* &*agent*)
Communicates with the *Engine* to get the latest policy.

Parameters **agent** – *Sample* containing current agent/state information.

void **getAction**(*Sample* &*agent*)
Runs the policy on the current state to get the action.

Parameters **agent** – *Sample* containing current agent/state information.

Public Members

size_t **_agentsPerEnvironment**
Number of agents in a given environment .

size_t **_policiesPerEnvironment**
Number of policies in a given environment. All agents share the same policy or all have individual policy.

size_t **_environmentCount**
Maximum number of different types of environments.

std::uint64_t **_environmentFunction**
Function to initialize and run an episode in the environment.

size_t **_actionsBetweenPolicyUpdates**
Number of actions to take before requesting a new policy.

size_t **_testingFrequency**
Number of episodes after which the policy will be tested.

size_t **_policyTestingEpisodes**
Number of test episodes to run the policy (without noise) for, for which the average sum of rewards will serve to evaluate the termination criteria.

knlohmann::json **_customSettings**
Any used-defined settings required by the environment.

size_t **_actionVectorSize**
[Internal Use] Stores the dimension of the action space.

size_t **_stateVectorSize**
[Internal Use] Stores the dimension of the state space.

std::vector<size_t> **_actionVectorIndexes**
[Internal Use] Stores the indexes of the variables that constitute the action vector.

std::vector<size_t> **_stateVectorIndexes**
[Internal Use] Stores the indexes of the variables that constitute the action vector.

size_t **_actionCount**
[Internal Use] The maximum number of actions an agent can take (only relevant for discrete).

std::vector<*std*::vector<float>> **_stateRescalingMeans**
Contains the state rescaling means.

std::vector<*std*::vector<float>> **_stateRescalingSdevs**
Contains the state rescaling sigmas.

double **_agentPolicyEvaluationTime**
[Profiling] Stores policy evaluation time per episode

double **_agentComputationTime**
[Profiling] Stores environment evaluation time per episode

double **_agentCommunicationTime**
[Profiling] Stores communication time per episode

class *korali*::*solver*::*optimizer*::**Rprop** : **public** *korali*::*solver*::*Optimizer*
#include <Rprop.hpp> Class declaration for module: *Rprop*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

virtual void **runGeneration**() **override**
Runs the current generation.

virtual void **printGenerationBefore**() **override**
Prints solver information before the execution of the current generation.

virtual void **printGenerationAfter**() **override**
Prints solver information after the execution of the current generation.

Public Members

double **_delta0**
Initial Delta.

double **_deltaMin**
Minimum Delta, parameter for step size calibration.

double **_deltaMax**
Maximum Delta, parameter for step size calibration.

double **_etaMinus**
Parameter for step size calibration.

double **_etaPlus**
Parameter for step size calibration.

std::vector<double> **_currentVariable**
[Internal Use] Current value of parameters.

std::vector<double> **_bestEverVariable**
[Internal Use] Best value of parameters.

std::vector<double> **_delta**
[Internal Use] Gradient scaling factor

std::vector<double> **_currentGradient**
[Internal Use] Gradient of parameters.

std::vector<double> **_previousGradient**
[Internal Use] Old gradient of parameters.

std::vector<double> **_bestEverGradient**
[Internal Use] Gradient of function with respect to Best Ever Variables.

double **_normPreviousGradient**
[Internal Use] Norm of old gradient.

double **_maxStallCounter**
[Internal Use] Counts the number the algorithm has been stalled in function evaluation bigger than the best one.

double **_xDiff**
[Internal Use] Norm of variable update.

double **_maxGradientNorm**
[Termination Criteria] Maximum value of the norm of the gradient.

size_t **_maxStallGenerations**
[Termination Criteria] Maximum times stalled with function evaluation bigger than the best one.

double **_parameterRelativeTolerance**
[Termination Criteria] Relative tolerance in parameter difference between generations.

Private Functions

void **evaluateFunctionAndGradient**(*Sample* &*sample*)

void **performUpdate**(void)

class *korali*::**Sample**
#include <sample.hpp> Contains input/output data to computational models.

Public Functions

Sample()
Constructs *Sample*. Stores its own pointer, sets ID to zero, state as uninitialized, and isAllocated to false.

~Sample()

void **run**(size_t *functionPosition*)
Runs a computational model by reinterpreting a numerical pointer to a function(sample) object to an actual function pointer and calls it.

Parameters **functionPosition** – Number containing a pointer to a function.

void **sampleLauncher**()
Handles the execution thread of individual samples on the worker's side.

void **update**()
Returns results to the worker without finishing the execution of the computational model.

knlohmann::json &**globals**()
Returns global parameters broadcasted by the problem.

Returns The global parameters

bool **contains**(**const** *std*::string &*key*)
Checks whether the sample contains the given key.

Parameters **key** – Key (String) to look for.

Returns True, if it is contained; false, otherwise.

knlohmann::json &**operator[]**(**const** *std*::string &*key*)
Accesses the value of a given key in the sample.

Parameters **key** – Key (String) to look for.

Returns JSON object for the given key.

knlohmann::json &**operator[]**(**const** unsigned long int &*key*)
Accesses the value of a given key in the sample.

Parameters **key** – Key (number) to look for.

Returns JSON object for the given key.

pybind11::object **getItem**(**const** pybind11::object *key*)
Gets the value of a given key in the sample.

Parameters **key** – Key (pybind11 object) to look for.

Returns Pybind11 object for the given key.

void **setItem**(**const** pybind11::object *key*, **const** pybind11::object *val*)
Sets the value of a given key in the sample.

Parameters

- **val** – Value to assign.
- **key** – Key (pybind11 object) to look for.

bool **retrievePendingMessage**(knlohmann::json &*message*)
Gets and dequeues a pending message, if exists.

Parameters **message** – The message (json object) to overwrite, if a message exists.

Returns True, if message found; false, if no message was found.

template<class **T**, typename ...**Key**>
inline *T* **get**(**const** char *fileName*[], int *lineNumber*, **const** *Key*&... *key*)
Retrieves an element from the sample information.

Parameters

- **fileName** – where the error occurred, given by the **FILE** macro
- **lineNumber** – number where the error occurred, given by the **LINE** macro
- **key** – a list of keys describing the full path to traverse

Returns Requested value

Public Members

Sample ***_self**
Pointer to the C++ object containing the sample. Necessary for integration with Python, because Python only passes objects by reference, and we need to access the original pointer when working on the C++ side. Therefore, we need to store the pointer as a variable.

std::queue<knlohmann::json> **_messageQueue**
Queue of messages sent from the sample to the engine.

SampleState **_state**
Current state of the sample.

cothread_t **_sampleThread**
User-Level thread (coroutine) containing the CPU execution state of the current *Sample*.

cothread_t **_workerThread**
User-Level thread (coroutine) containing the CPU execution state of the calling worker.

size_t **_workerId**
Storage to keep the iD of the worker processing this sample.

KoraliJson **_js**
JSON object containing the sample's configuration and input/output data.

std::vector<*std*::vector<float>> **_rawData**
Container for sending big raw data.

class *korali*::*solver*::**Sampler** : **public** *korali*::*Solver*
#include <sampler.hpp> Class declaration for module: *Sampler*.

Subclassed by *korali::solver::sampler::HMC*, *korali::solver::sampler::MCMC*, *korali::solver::sampler::Nested*, *korali::solver::sampler::TMCMC*

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

class *korali*::*problem*::**Sampling** : **public** *korali*::*Problem*
#include <sampling.hpp> Class declaration for module: *Sampling*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **runOperation**(*std*::string *operation*, *korali*::*Sample* &*sample*) **override**
Runs the operation specified on the given sample. It checks recursively whether the function was found by the current module or its parents.

Parameters

- **sample** – *Sample* to operate on. Should contain in the 'Operation' field an operation accepted by this module or its parents.
- **operation** – Should specify an operation type accepted by this module or its parents.

Returns True, if operation found and executed; false, otherwise.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **evaluate**(*korali*::*Sample* &*sample*)
Evaluates a function at the given parameters.

Parameters **sample** – *Sample* to evaluate

virtual void **evaluateGradient**(*korali*::*Sample* &*sample*)
Evaluates the gradient of a function at the given parameters.

Parameters **sample** – *Sample* to evaluate

virtual void **evaluateHessian**(*korali*::*Sample* &*sample*)
Evaluates the Heassian of a function at the given parameters.

Parameters **sample** – *Sample* to evaluate

Public Members

std::uint64_t **_probabilityFunction**
Stores the probability distribution function to evaluate.

class *korali*::*conduit*::**Sequential** : **public** *korali*::*Conduit*
#include <sequential.hpp> Class declaration for module: *Sequential*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual bool **isRoot**() **const override**
Determines whether the caller rank/thread/process is root.

Returns True, if it is root; false, otherwise.

virtual void **initServer**() **override**
Initializes the worker/server bifurcation in the conduit.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

virtual void **terminateServer**() **override**
Finalizes the workers.

virtual void **stackEngine**(*Engine* **engine*) **override**
Stacks a new *Engine* into the engine stack.

Parameters **engine** – A *Korali Engine*

virtual void **popEngine**() **override**
Pops the current *Engine* from the engine stack.

virtual void **listenWorkers**() **override**
(*Engine* <- Worker) Receives all pending incoming messages and stores them into the corresponding sample's message queue.

virtual void **broadcastMessageToWorkers**(knlohmann::json &*message*) **override**
(*Engine* -> Worker) Broadcasts a message to all workers

Parameters **message** – JSON object with information to broadcast

virtual void **sendMessageToEngine**(knlohmann::json &*message*) **override**
(*Sample* -> *Engine*) Sends an update to the engine to provide partial information while the sample is still active

Parameters **message** – Message to send to engine

virtual knlohmann::json **recvMessageFromEngine**() **override**
(*Sample* <- *Engine*) Blocking call that waits until any message incoming from the engine.

Returns message from the engine.

virtual void **sendMessageToSample**(*Sample* &*sample*, knlohmann::json &*message*) **override**
(*Engine* -> *Sample*) Sends an update to a still active sample

Parameters

- **sample** – The sample from which to receive an update
- **message** – Message to send to the sample.

virtual size_t **getProcessId**() **const override**
Returns the identifier corresponding to the executing process (to differentiate their random seeds)

Returns The executing process id

virtual size_t **getWorkerCount**() **const override**
Get total *Korali* worker count in the conduit.

Returns The number of workers

Public Members

cothread_t **_workerThread**
User-Level thread (coroutine) containing the CPU execution state of the single worker.

std::queue<knlohmann::json> **_workerMessageQueue**
Queue of messages sent from the engine to the worker.

class *korali*::**Solver** : **public** *korali*::*Module*
#include <solver.hpp> Class declaration for module: *Solver*.

Subclassed by *korali::solver::Agent*, *korali::solver::DeepSupervisor*, *korali::solver::Designer*, *korali::solver::Executor*, *korali::solver::Integrator*, *korali::solver::Optimizer*, *korali::solver::Sampler*, *korali::solver::SSM*

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **printGenerationBefore**()
Prints solver information before the execution of the current generation.

virtual void **printGenerationAfter**()
Prints solver information after the execution of the current generation.

virtual void **runGeneration**() = 0
Runs the current generation.

virtual void **setInitialConfiguration**()
Initializes the solver with starting values for the first generation.

Public Members

size_t **_variableCount**
[Internal Use] Number of variables.

size_t **_modelEvaluationCount**
[Internal Use] Keeps track on the number of calls to the computational model.

size_t **_maxModelEvaluations**
[Termination Criteria] Specifies the maximum allowed evaluations of the computational model.

size_t **_maxGenerations**
[Termination Criteria] Determines how many solver generations to run before stopping execution. Execution can be resumed at a later moment.

std::vector<*std*::string> **_terminationCriteria**
Stores termination criteria for the module.

class *korali*::*distribution*::**Specific** : **public** *korali*::*Distribution*
#include <specific.hpp> Class declaration for module: *Specific*.

Subclassed by *korali::distribution::specific::Multinomial*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

class *korali*::*solver*::*ssm*::**SSA** : **public** *korali*::*solver*::*SSM*
#include <SSA.hpp> Class declaration for module: *SSA*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **advance**() **override**
Simulates a trajectory for all reactants based on provided reactions.

Public Members

std::vector<double> **_cumPropensities**
Storage for cumulative propensities during each step.

class *korali*::*solver*::**SSM** : **public** *korali*::*Solver*
#include <SSM.hpp> Class declaration for module: *SSM*.

Subclassed by *korali::solver::ssm::SSA*, *korali::solver::ssm::TauLeaping*

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **reset**(*std*::vector<int> *numReactants*, double *time* = 0.)
Resets the initial conditions of a new trajectory simulation.

Parameters

- **numReactants** – initial reactants for new simulation
- **time** – starting time of new simulation

virtual void **advance**() = 0
Simulates a trajectory for all reactants based on provided reactions.

void **updateBins**()
Updates the values of the binned trajectories for each reactant.

virtual void **setInitialConfiguration**()
Initializes the solver with starting values for the first generation.

virtual void **runGeneration**() **override**
Runs the current generation.

virtual void **printGenerationBefore**() **override**
Prints solver information before the execution of the current generation.

virtual void **printGenerationAfter**() **override**
Prints solver information after the execution of the current generation.

virtual void **finalize**() **override**
Finalizes *Module*. Deallocates memory and produces outputs.

Public Members

double **_simulationLength**
Total duration of a stochastic reaction simulation.

size_t **_diagnosticsNumBins**
Number of bins to calculate the mean trajectory at termination.

size_t **_simulationsPerGeneration**
Number of trajectory simulations per *Korali* generation (checkpoints are generated between generations).

double **_time**
[Internal Use] The current time of the simulated trajectory.

size_t **_numReactions**
[Internal Use] The number of reactions to simulate.

std::vector<int> **_numReactants**
[Internal Use] The current number of each reactant in the simulated trajectory.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Uniform random number generator.

std::vector<double> **_binTime**
[Internal Use] The simulation time associated to each bin.

std::vector<*std*::vector<int>> **_binCounter**
[Internal Use] Stores the number of reactants per bin for each trajectory and reactant.

std::vector<*std*::vector<*std*::vector<int>>> **_binnedTrajectories**
[Internal Use] Stores the number of reactants per bin for each trajectory and reactant.

size_t **_completedSimulations**
[Internal Use] Counter that keeps track of completed simulations.

size_t **_maxNumSimulations**
[Termination Criteria] Max number of trajectory simulations.

problem::*Reaction* ***_problem**
Storage for the pointer to the reaction problem.

class *korali*::*problem*::**SupervisedLearning** : **public** *korali*::*Problem*
#include <supervisedLearning.hpp> Class declaration for module: *SupervisedLearning*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

void **verifyData**()
Checks whether the input data has the correct shape.

Public Members

size_t **_trainingBatchSize**
Stores the batch size of the training dataset.

size_t **_testingBatchSize**
Stores the batch size of the testing dataset.

size_t **_maxTimesteps**
Stores the length of the sequence for recurrent neural networks.

std::vector<*std*::vector<*std*::vector<float>>> **_inputData**
Provides the input data with layout T*N*IC, where T is the sequence length, N is the batch size and IC is the vector size of the input.

size_t **_inputSize**
Indicates the vector size of the input (IC).

std::vector<*std*::vector<float>> **_solutionData**
Provides the solution for one-step ahead prediction with layout N*OC, where N is the batch size and OC is the vector size of the output.

size_t **_solutionSize**
Indicates the vector size of the output (OC).

class *korali*::*solver*::*ssm*::**TauLeaping** : **public** *korali*::*solver*::*SSM*
#include <TauLeaping.hpp> Class declaration for module: *TauLeaping*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

double **estimateLargestTau**()
Estimate time step such that that many reaction events occur, but small enough that no propensity functin changes significantly.

Returns tau time step duration

void **ssaAdvance**()
SSA advance step if leap step rejected.

virtual void **setInitialConfiguration**() **override**
Initializes the solver with starting values for the first generation.

virtual void **advance**() **override**
Simulates a trajectory for all reactants based on provided reactions.

Public Members

int **_nc**
TODO.

double **_epsilon**
Error control parameter, larger epsilon leads to larger tau steps and errors.

double **_acceptanceFactor**
Multiplicator of inverse total propensity, to calculate acceptance criterion of tau step.

int **_numSSASteps**
Number of intermediate *SSA* steps if leap step rejected.

korali::*distribution*::*univariate*::*Poisson* ***_poissonGenerator**
[Internal Use] Poisson random number generator.

std::vector<double> **_mu**
[Internal Use] Estimated means of the expected change of reactants.

std::vector<double> **_variance**
[Internal Use] Estimated variance of the expected change of reactants.

std::vector<double> **_propensities**
Storage for propensities of reactions during each step.

std::vector<double> **_cumPropensities**
Storage for cumulative propensities during each *SSA* step.

std::vector<int> **_numFirings**
Storage for number of firings per reaction per tau leap step.

std::vector<double> **_isCriticalReaction**
Storage for critical reaction marker during each step.

std::vector<int> **_candidateNumReactants**
Storage for candidate reactants per leap step.

class *korali*::*problem*::*hierarchical*::**Theta** : **public** *korali*::*problem*::*Hierarchical*
#include <theta.hpp> Class declaration for module: *Theta*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **evaluateLogLikelihood**(*korali*::*Sample* &*sample*) **override**
Evaluates the log likelihood of the given sample, and stores it in sample["Log Likelihood"].

Parameters **sample** – A *Korali Sample*

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

Public Members

knlohmann::json **_subExperiment**
Results from one previously executed *Bayesian* experiment.

knlohmann::json **_psiExperiment**
Results from the hierarchical problem (*Psi*).

Private Members

korali::*Experiment* **_psiExperimentObject**
Stores the actual *Korali* object for the psi experiment.

korali::*Experiment* **_subExperimentObject**
Stores the actual *Korali* object for the theta experiment.

size_t **_psiVariableCount**
Stores the number of variables defined in the *Psi* problem.

size_t **_psiProblemSampleCount**
Stores the number of samples in the *Psi* problem experiment to use as input.

std::vector<*std*::vector<double>> **_psiProblemSampleCoordinates**
Stores the sample coordinates of the *Psi Problem*.

std::vector<double> **_psiProblemSampleLogLikelihoods**
Stores the sample logLikelihoods of the *Psi Problem*.

std::vector<double> **_psiProblemSampleLogPriors**
Stores the sample logPriors of the *Psi Problem*.

korali::*problem*::*hierarchical*::*Psi* ***_psiProblem**
Stores the *Problem* module of the *Psi* problem experiment to use as input.

size_t **_subProblemVariableCount**
Stores the number of variables defined in the Sub problem.

size_t **_subProblemSampleCount**
Stores the number of samples in the sub problem experiment to use as input.

std::vector<*std*::vector<double>> **_subProblemSampleCoordinates**
Stores the sample coordinates of the sub *Problem*.

std::vector<double> **_subProblemSampleLogLikelihoods**
Stores the sample logLikelihoods of the sub *Problem*.

std::vector<double> **_subProblemSampleLogPriors**
Stores the sample logPriors of the sub *Problem*.

std::vector<double> **_precomputedLogDenominator**
Stores the precomputed log denomitator to speed up calculations.

class *korali*::*problem*::*hierarchical*::**ThetaNew** : **public** *korali*::*problem*::*Hierarchical*
#include <thetaNew.hpp> Class declaration for module: *ThetaNew*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual void **evaluateLogLikelihood**(*korali*::*Sample* &*sample*) **override**
Evaluates the log likelihood of the given sample, and stores it in sample["Log Likelihood"].

Parameters **sample** – A *Korali Sample*

void **evaluateThetaLikelihood**(*korali*::*Sample* &*sample*)
Evaluates the theta log likelihood of the given sample.

Parameters **sample** – A *Korali Sample*

virtual void **initialize**() **override**
Initializes *Module* upon creation. May allocate memory, set initial states, and initialize external code.

Public Members

knlohmann::json **_psiExperiment**
Results from the hierarchical *Psi* experiment.

Private Members

korali::*Experiment* **_psiExperimentObject**
Stores the actual *Korali* object for the psi experiment.

size_t **_psiProblemSampleCount**
Stores the number of samples in the *Psi* problem experiment to use as input.

std::vector<*std*::vector<double>> **_psiProblemSampleCoordinates**
Stores the sample coordinates of the *Psi Problem*.

std::vector<double> **_psiProblemSampleLogLikelihoods**
Stores the sample logLikelihoods of the *Psi Problem*.

std::vector<double> **_psiProblemSampleLogPriors**
Stores the sample logPriors of the *Psi Problem*.

class *korali*::*solver*::*sampler*::**TMCMC** : **public** *korali*::*solver*::*Sampler*

#include <TMCMC.hpp> Class declaration for module: *TMCMC*.

Public Functions

virtual bool **checkTermination**() **override**

Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**

Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**

Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**

Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**

Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **setBurnIn**()

Sets the burn in steps per generation.

void **prepareGeneration**()

Prepare Generation before evaluation.

void **processGeneration**()

Process Generation after receiving all results.

void **minSearch**(double **const** **fj*, size_t *fn*, double *pj*, double *objTol*, double &*xmin*, double &*fmin*)

Helper function for annealing exponent update/.

Parameters

- **fj** – Pointer to exponentiated probability values.
- **fn** – Current exponent.
- **pj** – Number of values in fj array.
- **objTol** – Tolerance
- **xmin** – Location of minimum, the new exponent.
- **fmin** – Found minimum in search.

void **processCandidate**(**const** size_t *sampleId*)

Collects results after sample evaluation.

Parameters **sampleId** – Id of the sample to process

void **calculateGradients**(*std*::vector<*Sample*> &*samples*)

Calculate gradients of loglikelihood (only relevant for mTMCMC).

Parameters **samples** – Samples to calculate gradients for

void **calculateProposals**(*std*::vector<*Sample*> &*samples*)
Calculate sample wise proposal distributions (only relevant for mTMCMC).

Parameters **samples** – Samples to calculate proposal distributions for

void **generateCandidate**(**const** size_t *sampleId*)
Generate candidate from leader.

Parameters **sampleId** – Id of the sample to generate

void **updateDatabase**(**const** size_t *sampleId*)
Add leader into sample database.

Parameters **sampleId** – Id of the sample to update the database with

double **calculateAcceptanceProbability**(**const** size_t *sampleId*)
Calculate acceptance probability.

Parameters **sampleId** – Id of the sample to calculate acceptance probability

Returns The acceptance probability of the given sample

virtual void **setInitialConfiguration**() **override**
Configures *TMCMC*.

virtual void **runGeneration**() **override**
Main solver loop.

virtual void **printGenerationBefore**() **override**
Console Output before generation runs.

virtual void **printGenerationAfter**() **override**
Console output after generation.

virtual void **finalize**() **override**
Final console output at termination.

Public Members

std::string **_version**
Indicates which variant of the *TMCMC* algorithm to use.

size_t **_populationSize**
Specifies the number of samples drawn from the posterior distribution at each generation.

size_t **_maxChainLength**
Chains longer than Max Chain Length will be broken and samples will be duplicated (replacing samples associated with a chain length of 0). Max Chain Length of 1 corresponds to the BASIS algorithm [Wu2018].

size_t **_burnIn**
Specifies the number of additional *TMCMC* steps per chain per generation (except for generation 0 and 1).

std::vector<size_t> **_perGenerationBurnIn**
Specifies the number of additional *TMCMC* steps per chain at specified generations (this property will overwrite Default Burn In at specified generations). The first entry of the vector corresponds to the 2nd *TMCMC* generation.

double **_targetCoefficientOfVariation**
Target coefficient of variation of the plausibility weights to update the annealing exponent :math:`\rho` (by default, this value is 1.0 as suggested in [Ching2007]).

double **_covarianceScaling**
Scaling factor :math:\beta^2 of Covariance Matrix (by default, this value is 0.04 as suggested in [Ching2007]).

double **_minAnnealingExponentUpdate**
Minimum increment of the exponent :math:\rho. This parameter prevents *TMCMC* from stalling.

double **_maxAnnealingExponentUpdate**
Maximum increment of the exponent :math:\rho (by default, this value is 1.0 (inactive)).

double **_stepSize**
Scaling factor of gradient and proposal distribution (only relevant for mTMCMC).

double **_domainExtensionFactor**
Defines boundaries for eigenvalue adjustments of proposal distribution (only relevant for mTMCMC).

korali::*distribution*::*specific*::*Multinomial* ***_multinomialGenerator**
[Internal Use] Random number generator with a multinomial distribution.

korali::*distribution*::*multivariate*::*Normal* ***_multivariateGenerator**
[Internal Use] Random number generator with a multivariate normal distribution.

korali::*distribution*::*univariate*::*Uniform* ***_uniformGenerator**
[Internal Use] Random number generator with a uniform distribution.

size_t **_currentBurnIn**
[Internal Use] Actual placeholder for burn in steps per generation, calculated from Burn In Default, Burn In and Current Generation.

std::vector<int> **_chainPendingEvaluation**
[Internal Use] Indicates that the model evaluation for the chain is pending.

std::vector<int> **_chainPendingGradient**
[Internal Use] Indicates that the gradient evaluation for the chain is pending (only relevant for mTMCMC).

std::vector<*std*::vector<double>> **_chainCandidates**
[Internal Use] All candidates of all chains to evaluate in order to advance the markov chains.

std::vector<double> **_chainCandidatesLogLikelihoods**
[Internal Use] The loglikelihoods of the chain candidates.

std::vector<double> **_chainCandidatesLogPriors**
[Internal Use] The logpriors of the chain candidates.

std::vector<*std*::vector<double>> **_chainCandidatesGradients**
[Internal Use] Candidate gradient of statistical model wrt. sample variables.

std::vector<int> **_chainCandidatesErrors**
[Internal Use] Shows if covariance calculation successfully terminated for candidate (only relevant for mTMCMC).

std::vector<*std*::vector<double>> **_chainCandidatesCovariance**
[Internal Use] Candidates covariance of normal proposal distribution.

std::vector<*std*::vector<double>> **_chainLeaders**
[Internal Use] Leading parameters of all chains to be accepted.

std::vector<double> **_chainLeadersLogLikelihoods**
[Internal Use] The loglikelihoods of the chain leaders.

std::vector<double> **_chainLeadersLogPriors**
[Internal Use] The logpriors of the chain leaders.

std::vector<*std*::vector<double>> **_chainLeadersGradients**
[Internal Use] Leader gradient of statistical model wrt. sample variables.

std::vector<int> **_chainLeadersErrors**
[Internal Use] Shows if covariance calculation successfully terminated for leader (only relevant for mTMCMC).

std::vector<*std*::vector<double>> **_chainLeadersCovariance**
[Internal Use] Leader covariance of normal proposal distribution.

size_t **_finishedChainsCount**
[Internal Use] Number of finished chains.

std::vector<size_t> **_currentChainStep**
[Internal Use] The current execution step for every chain.

std::vector<size_t> **_chainLengths**
[Internal Use] Lengths for each of the chains.

double **_coefficientOfVariation**
[Internal Use] Actual coefficient of variation after :math:`\rho` has beed updated.

size_t **_chainCount**
[Internal Use] Unique selections after resampling stage.

double **_annealingExponent**
[Internal Use] Exponent of the likelihood. If :math:`\rho` equals 1.0, *TMCMC* converged.

double **_previousAnnealingExponent**
[Internal Use] Previous Exponent of the likelihood. If :math:`\rho` equals 1.0, *TMCMC* converged.

size_t **_numFinitePriorEvaluations**
[Internal Use] Number of finite prior evaluations per gerneration.

size_t **_numFiniteLikelihoodEvaluations**
[Internal Use] Number of finite likelihood evaluations per gerneration.

size_t **_acceptedSamplesCount**
[Internal Use] Accepted candidates after proposal.

double **_currentAccumulatedLogEvidence**
[Internal Use] The current accumulated logEvidence. At termination, this is the logEvidence of the model.

double **_proposalsAcceptanceRate**
[Internal Use] Acceptance rate calculated from accepted samples.

double **_selectionAcceptanceRate**
[Internal Use] Acceptance rate calculated from unique samples (chain count) after recombination.

std::vector<double> **_covarianceMatrix**
[Internal Use] *Sample* covariance of the current leaders updated at every generation.

double **_maxLoglikelihood**
[Internal Use] Max Loglikelihood found in current generation.

std::vector<double> **_meanTheta**
[Internal Use] Mean of the current leaders updated at every generation.

std::vector<*std*::vector<double>> **_sampleDatabase**
[Internal Use] Parameters stored in the database (taken from the chain leaders).

std::vector<double> **_sampleLogLikelihoodDatabase**
[Internal Use] LogLikelihood Evaluation of the parameters stored in the database.

std::vector<double> **_sampleLogPriorDatabase**
[Internal Use] Log priors of the samples stored in the database.

std::vector<*std*::vector<double>> **_sampleGradientDatabase**
[Internal Use] Gradients stored in the database (taken from the chain leaders, only mTMCMC).

std::vector<int> **_sampleErrorDatabase**
[Internal Use] Shows if covariance calculation successfully terminated for sample (only relevant for mTMCMC).

std::vector<*std*::vector<double>> **_sampleCovarianceDatabase**
[Internal Use] Gradients stored in the database (taken from the chain leaders, only mTMCMC).

std::vector<double> **_upperExtendedBoundaries**
[Internal Use] Calculated upper domain boundaries (only relevant for mTMCMC).

std::vector<double> **_lowerExtendedBoundaries**
[Internal Use] Calculated lower domain boundaries (only relevant for mTMCMC).

size_t **_numLUDecompositionFailuresProposal**
[Internal Use] Number of failed LU decompositions (only relevan for mTMCMC).

size_t **_numEigenDecompositionFailuresProposal**
[Internal Use] Number of failed Eigenvalue problems (only relevan for mTMCMC).

size_t **_numInversionFailuresProposal**
[Internal Use] Number of failed FIM inversions (only relevan for mTMCMC).

size_t **_numNegativeDefiniteProposals**
[Internal Use] Number of Fisher information matrices with negative eigenvalues (only relevan for mTMCMC).

size_t **_numCholeskyDecompositionFailuresProposal**
[Internal Use] Number of failed chol. decomp. during proposal step (only relevant for mTMCMC).

size_t **_numCovarianceCorrections**
[Internal Use] Number of covariance adaptions (only relevant for mTMCMC).

double **_targetAnnealingExponent**
[Termination Criteria] Determines the annealing exponent :math:`\rho` to achieve before termination. *TMCMC* converges if :math:`\rho` equals 1.0.

size_t **N**
Number of variables to sample.

Public Static Functions

static double **calculateSquaredCVDifference**(double *x*, **const** double **loglike*, size_t *Ns*, double *exponent*, double *targetCV*)
Helper function to calculate the squared difference between (CVaR) for min search.

Parameters

- **x** – Alternative exponent
- **loglike** – Vector of loglikelihood values
- **Ns** – Size of loglike array
- **exponent** – Current rho
- **targetCV** – Target CV

Returns The squared CV difference

static double **calculateSquaredCVDifferenceOptimizationWrapper**(**const** gsl_vector *v, void *param)

Helper function for minimization procedure to find the target CV.

Parameters

- **v** – Input GSL vector containing loglikelihood values
- **param** – Input parameter for method 'calculateSquaredCVDifference'

Returns The squared CV difference

struct *korali*::*solver*::*sampler*::**TreeHelper**

#include <tree_helper_base.hpp> Abstract helper class for long argument list of buildTree.

Subclassed by *korali::solver::sampler::TreeHelperEuclidean*, *korali::solver::sampler::TreeHelperRiemannian*

Public Functions

virtual bool **computeCriterion**(**const** *Hamiltonian* &*hamiltonian*) **const** = 0

Computes No U-Turn Sampling (NUTS) criterion.

Parameters **hamiltonian** – *Hamiltonian* object of system.

Returns Returns of tree should be built further.

virtual bool **computeCriterion**(**const** *Hamiltonian* &*hamiltonian*, **const** *std*::vector<double> &*momentumStart*, **const** *std*::vector<double> &*momentumEnd*, **const** *std*::vector<double> &*inverseMetric*, **const** *std*::vector<double> &*rho*) **const** = 0

Purely virtual function, computes No U-Turn Sampling (NUTS) criterion.

Parameters

- **hamiltonian** – *Hamiltonian* object of system.
- **momentumStart** – Starting momentum of trajectory.
- **momentumEnd** – Ending momentum of trajectory.
- **inverseMetric** – Inverse of current metric.
- **rho** – Sum of momenta encountered in trajectory.

Returns Returns of tree should be built further.

virtual **~TreeHelper**() = default

Default destructor.

Public Members

std::vector<double> **qIn**
Position input.

std::vector<double> **pIn**
Momentum input.

double **logUniSampleIn**
Log of uni sample input.

int **directionIn**
Direction in which to propagate input.

double **rootHIn**
Energy of root of binary tree (i.e. starting position) input.

std::vector<double> **qLeftOut**
Leftmost position output.

std::vector<double> **pLeftOut**
Leftmost momentum output.

std::vector<double> **qRightOut**
Rightmost position output.

std::vector<double> **pRightOut**
Rightmost momentum output.

std::vector<double> **qProposedOut**
Proposed position output.

double **numValidLeavesOut**
Number of valid leaves output (needed for acceptance probability).

bool **buildCriterionOut**
No U-Turn Termination Sampling (NUTS) criterion output.

double **alphaOut**
Acceptance probability output.

size_t **numLeavesOut**
Number of valid leaves encountererd (needed for adaptive time stepping).

struct *korali*::*solver*::*sampler*::**TreeHelperEuclidean** : **public** *korali*::*solver*::*sampler*::*TreeHelper*
#include <tree_helper_euclidean.hpp> Euclidean helper class for long argument list of buildTree.

Public Functions

inline virtual bool **computeCriterion**(**const** *Hamiltonian* *&hamiltonian*) **const override**
Computes No U-Turn Sampling (NUTS) criterion.

Parameters **hamiltonian** – *Hamiltonian* object of system

Returns Returns of tree should be built further.

inline virtual bool **computeCriterion**(**const** *Hamiltonian* *&hamiltonian*, **const** *std*::vector<double> *&momentumStart*, **const** *std*::vector<double> *&momentumEnd*, **const** *std*::vector<double> *&inverseMetric*, **const** *std*::vector<double> *&rho*) **const override**

Computes No U-Turn Sampling (NUTS) criterion.

Parameters

- **hamiltonian** – *Hamiltonian* object of system.
- **momentumStart** – Starting momentum of trajectory.
- **momentumEnd** – Ending momentum of trajsectory.
- **inverseMetric** – Inverse of current metric.
- **rho** – Sum of momenta encountered in trajectory.

Returns Returns of tree should be built further.

struct *korali*::*solver*::*sampler*::**TreeHelperRiemannian** : **public** *korali*::*solver*::*sampler*::*TreeHelper*
#include <tree_helper_riemannian.hpp> Riemmanian helper class for long argument list of buildTree.

Public Functions

inline virtual bool **computeCriterion**(**const** *Hamiltonian* &*hamiltonian*) **const override**
Computes No U-Turn Sampling (NUTS) criterion.

Parameters **hamiltonian** – *Hamiltonian* object of system.

Returns Returns of tree should be built further.

inline virtual bool **computeCriterion**(**const** *Hamiltonian* &*hamiltonian*, **const** *std*::vector<double> &*momentumStart*, **const** *std*::vector<double> &*momentumEnd*, **const** *std*::vector<double> &*inverseMetric*, **const** *std*::vector<double> &*rho*) **const override**
Computes No U-Turn Sampling (NUTS) criterion.

Parameters

- **hamiltonian** – *Hamiltonian* object of system.
- **momentumStart** – Starting momentum of trajectory.
- **momentumEnd** – Ending momentum of trajsectory.
- **inverseMetric** – Inverse of current metric.
- **rho** – Sum of momenta encountered in trajectory.

Returns Returns of tree should be built further.

class *korali*::*distribution*::*univariate*::**TruncatedNormal** : **public** *korali*::*distribution*::*Univariate*
#include <truncatedNormal.hpp> Class declaration for module: *TruncatedNormal*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_mu**
[Conditional Variable Value] The mean of the untruncated *Normal* distribution.

std::string **_muConditional**
[Conditional Variable Reference] The mean of the untruncated *Normal* distribution.

double **_sigma**
[Conditional Variable Value] The standard deviation of the untruncated *Normal* distribution.

std::string **_sigmaConditional**
[Conditional Variable Reference] The standard deviation of the untruncated *Normal* distribution.

double **_minimum**
[Conditional Variable Value] The lower bound of the truncated *Normal* distribution.

std::string **_minimumConditional**
[Conditional Variable Reference] The lower bound of the truncated *Normal* distribution.

double **_maximum**
[Conditional Variable Value] The upper bound of the truncated *Normal* distribution.

std::string **_maximumConditional**
[Conditional Variable Reference] The upper bound of the truncated *Normal* distribution.

Private Members

double **_normalization**

double **_logNormalization**

class *korali*::*distribution*::*univariate*::**Uniform** : **public** *korali*::*distribution*::*Univariate*
#include <uniform.hpp> Class declaration for module: *Uniform*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_minimum**
[Conditional Variable Value] The lower bound of the uniform distribution.

std::string **_minimumConditional**
[Conditional Variable Reference] The lower bound of the uniform distribution.

double **_maximum**
[Conditional Variable Value] The upper bound of the uniform distribution.

std::string **_maximumConditional**
[Conditional Variable Reference] The upper bound of the uniform distribution.

class *korali*::*distribution*::*univariate*::**UniformRatio** : **public** *korali*::*distribution*::*Univariate*
#include <uniformratio.hpp> Class declaration for module: *UniformRatio*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_minimumX**
[Conditional Variable Value] Lower bound of the first (dividend) uniform distribution.

std::string **_minimumXConditional**
[Conditional Variable Reference] Lower bound of the first (dividend) uniform distribution.

double **_maximumX**
[Conditional Variable Value] Upper bound of the first (divident) uniform distribution.

std::string **_maximumXConditional**
[Conditional Variable Reference] Upper bound of the first (divident) uniform distribution.

double **_minimumY**
[Conditional Variable Value] Lower bound of the second (divisor) uniform distribution.

std::string **_minimumYConditional**
[Conditional Variable Reference] Lower bound of the second (divisor) uniform distribution.

double **_maximumY**
[Conditional Variable Value] Upper bound of the second (divisor) uniform distribution.

std::string **_maximumYConditional**
[Conditional Variable Reference] Upper bound of the second (divisor) uniform distribution.

class *korali*::*distribution*::**Univariate** : **public** *korali*::*Distribution*
#include <univariate.hpp> Class declaration for module: *Univariate*.

Subclassed by *korali::distribution::univariate::Beta*, *korali::distribution::univariate::Cauchy*, *korali::distribution::univariate::Exponential*, *korali::distribution::univariate::Gamma*, *korali::distribution::univariate::Geometric*, *korali::distribution::univariate::Igamma*, *korali::distribution::univariate::Laplace*, *korali::distribution::univariate::LogNormal*, *korali::distribution::univariate::Normal*, *korali::distribution::univariate::Poisson*, *korali::distribution::univariate::TruncatedNormal*, *korali::distribution::univariate::Uniform*, *korali::distribution::univariate::UniformRatio*, *korali::distribution::univariate::Weibull*

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double **getDensity**(**const** double *x*) **const** = 0
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(**const** double *x*) **const** = 0
Gets the log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

inline virtual double **getLogDensityGradient**(**const** double *x*) **const**
Gets the gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

inline virtual double **getLogDensityHessian**(**const** double *x*) **const**
Gets the second derivative of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate H(log(P(x)))

Returns Hessian of log of probability density.

virtual double **getRandomNumber**() = 0
Draws and returns a random number from the distribution.

Returns Random real number.

class *korali*::*solver*::*agent*::*continuous*::**VRACER** : **public** *korali*::*solver*::*agent*::*Continuous*
#include <VRACER.hpp> Class declaration for module: *VRACER*.

Public Functions

virtual bool **checkTermination**() **override**
Determines whether the module can trigger termination of an experiment run.

Returns True, if it should trigger termination; false, otherwise.

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

void **updateVtbc**(size_t *expId*)
Update the V-target or current and previous experiences in the episode.

Parameters **expId** – Current Experience Id

void **calculatePolicyGradients**(**const** *std*::vector<*std*::pair<size_t, size_t>> &*miniBatch*, **const** size_t *policyIdx*)
Calculates the gradients for the policy/critic neural network.

Parameters

- **miniBatch** – The indexes of the experience mini batch
- **policyIdx** – The indexes of the policy to compute the gradient for

virtual float **calculateStateValue**(**const** *std*::vector<*std*::vector<float>> &*stateSequence*, size_t *policyIdx* = 0) **override**

Function to pass a state time series through the NN and calculates the action probabilities, along with any additional information.

Parameters

- **stateSequence** – The batch of state time series (Format: BxTxS, B is batch size, T is the time series lenght, and S is the state size)
- **policyIdx** – The index for the policy for which the state-value is computed

Returns A JSON object containing the information produced by the policies given the current state series

virtual void **runPolicy**(**const** *std*::vector<*std*::vector<*std*::vector<float>>> &*stateSequenceBatch*, *std*::vector<*policy_t*> &*policy*, size_t *policyIdx* = 0) **override**

Function to pass a state time series through the NN and calculates the action probabilities, along with any additional information.

Parameters

- **stateSequenceBatch** – The batch of state time series (Format: BxTxS, B is batch size, T is the time series lenght, and S is the state size)
- **policy** – Vector with policy objects that is filled after forwarding the policy
- **policyIdx** – The index for the policy for which the state-value is computed

virtual knlohmann::json **getPolicy**() **override**

Obtains the policy hyperaparamters from the learner for the agent to generate new actions.

Returns The current policy hyperparameters

virtual void **setPolicy**(**const** knlohmann::json &*hyperparameters*) **override**

Updates the agent's hyperparameters.

Parameters **hyperparameters** – The hyperparameters to update the agent.

virtual void **trainPolicy**() **override**

Trains the policy, based on the new experiences.

virtual void **printInformation**() **override**

Prints information about the training policy.

virtual void **initializeAgent**() **override**

Initializes the internal state of the policy.

Public Members

std::vector<float> **_miniBatchPolicyMean**
[Statistics] Keeps track of the mu of the current minibatch for each action variable

std::vector<float> **_miniBatchPolicyStdDev**
[Statistics] Keeps track of the sigma of the current minibatch for each action variable

class *korali*::*distribution*::*univariate*::**Weibull** : **public** *korali*::*distribution*::*Univariate*
#include <weibull.hpp> Class declaration for module: *Weibull*.

Public Functions

virtual void **getConfiguration**(knlohmann::json &*js*) **override**
Obtains the entire current state and configuration of the module.

Parameters **js** – JSON object onto which to save the serialized state of the module.

virtual void **setConfiguration**(knlohmann::json &*js*) **override**
Sets the entire state and configuration of the module, given a JSON object.

Parameters **js** – JSON object from which to deserialize the state of the module.

virtual void **applyModuleDefaults**(knlohmann::json &*js*) **override**
Applies the module's default configuration upon its creation.

Parameters **js** – JSON object containing user configuration. The defaults will not override any currently defined settings.

virtual void **applyVariableDefaults**() **override**
Applies the module's default variable configuration to each variable in the *Experiment* upon creation.

virtual double ***getPropertyPointer**(**const** *std*::string &*property*) **override**
Retrieves the pointer of a conditional value of a distribution property.

Parameters **property** – Name of the property to find.

Returns The pointer to the property..

virtual void **updateDistribution**() **override**
Updates the parameters of the distribution based on conditional variables.

virtual double **getDensity**(**const** double *x*) **const override**
Gets the probability density of the distribution at point x.

Parameters **x** – point to evaluate P(x)

Returns Value of the probability density.

virtual double **getLogDensity**(double *x*) **const override**
Gets the Log probability density of the distribution at point x.

Parameters **x** – point to evaluate log(P(x))

Returns Log of probability density.

virtual double **getLogDensityGradient**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getLogDensityHessian**(double *x*) **const override**
Gets the Gradient of the log probability density of the distribution wrt. to x.

Parameters **x** – point to evaluate grad(log(P(x)))

Returns Gradient of log of probability density.

virtual double **getRandomNumber**() **override**
Draws and returns a random number from the distribution.

Returns Random real number.

Public Members

double **_shape**
[Conditional Variable Value] The shape of the *Weibull* distribution.

std::string **_shapeConditional**
[Conditional Variable Reference] The shape of the *Weibull* distribution.

double **_scale**
[Conditional Variable Value] The scale of the *Weibull* distribution.

std::string **_scaleConditional**
[Conditional Variable Reference] The scale of the *Weibull* distribution.

namespace agent
Namespace declaration for modules of type: agent.

namespace bayesian
Namespace declaration for modules of type: bayesian.

namespace conduit
Namespace declaration for modules of type: conduit.

namespace continuous
Namespace declaration for modules of type: continuous.

namespace discrete
Namespace declaration for modules of type: discrete.

namespace distribution
Namespace declaration for modules of type: distribution.

namespace Eigen

namespace hierarchical
Namespace declaration for modules of type: hierarchical.

namespace integrator
Namespace declaration for modules of type: integrator.

namespace Korali
The *Korali* namespace includes all Korali-specific functions, variables, and modules.

namespace korali
The *Korali* namespace includes all Korali-specific functions, variables, and modules.

Namespace declaration for modules of type: korali.

Typedefs

typedef uint8_t **crc**
Special type for CRC calculation.

typedef long int **MPI_Comm**
Dummy communicator storage for the current *Korali* Worker.

Enums

enum SampleState
Execution states of a given sample.

Values:

enumerator uninitialized

enumerator initialized

enumerator running

enumerator waiting

enumerator finished

Functions

void **mkdir**(**const** *std*::string *dirPath*)
Creates a new folder and builds the entire path, if necessary.

Parameters **dirPath** – relative path to the new folder.

bool **dirExists**(**const** *std*::string *dirPath*)
Lists all files within within a given folder path.

Parameters **dirPath** – relative path to the folder to list.

Returns A list with the path of all files found.

bool **isEmpty**(**const** knlohmann::json &*js*)
Checks whether the JSON object is empty.

Parameters **js** – The JSON object to check.

Returns true, if it's empty; false, otherwise.

bool **isElemental**(**const** knlohmann::json &*js*)
Checks whether the JSON object is of elemental type (number or string).

Parameters **js** – The JSON object to check.

Returns true, if it's elemental; false, otherwise.

void **mergeJson**(knlohmann::json &*dest*, **const** knlohmann::json &*defaults*)
Merges the values of two JSON objects recursively and applying priority.

Parameters

- **dest** – the JSON object onto which the changes will be made. Values here have priority (are not replaced).
- **defaults** – the JSON object that applies onto the other. Values here have no priority (they will not replace)

bool **loadJsonFromFile**(knlohmann::json &*dst*, **const** char **fileName*)
Loads a JSON object from a file.

Parameters

- **dst** – The JSON object to overwrite.
- **fileName** – The path to the json file to load and parse.

Returns true, if file was found; false, otherwise.

int **saveJsonToFile**(**const** char **fileName*, **const** knlohmann::json &*js*)
Saves a JSON object to a file.

Parameters

- **fileName** – The path to the file onto which to save the JSON object.
- **js** – The input JSON object.

Returns 0 if successful, otherwise if not.

template<typename **T**>
T ***getPointer**(*T* &*x*)
Function made exclusively made to avoid warnings on getting the last element of variadic template arguments.

Parameters **x** – is the element to get the pointer from

Returns The element's pointer

template<typename **T**, typename ...**Key**>
void **eraseValue**(*T* &*js*, **const** *Key*&... *key*)
Deletes a value on a given JS given a string containing the full path.

Parameters

- **js** – The JSON object to modify.
- **key** – a list of keys describing the full path to traverse

template<typename **T**, typename ...**Key**>
bool **isDefined**(*T* &*js*, **const** *Key*&... *key*)
Checks whether a given key is present in the JSON object.

Parameters

- **js** – The JSON object to check.
- **key** – a list of keys describing the full path to traverse

Returns true, if the path defined by settings is found; false, otherwise.

template<typename **T**, typename ...**Key**>
T **getValue**(*T* &*js*, **const** *Key*&... *key*)
Returns a value on a given object given a string containing the full path.

Parameters

- **js** – The source object to read from.
- **key** – a list of keys describing the full path to traverse

Returns Object of the requested path

template<typename ...**Key**>

std::string **getPath**(**const** *Key*&... *key*)
Returns a string out of a list of keys showing.

Parameters **key** – a list of keys describing the full path to traverse

Returns The string with a printed key sequence

std::string **toLower**(**const** *std*::string &*input*)
Generates lower case string of provided string.

Parameters **input** – Input string

Returns The lower case varient of the string

bool **iCompare**(**const** *std*::string &*a*, **const** *std*::string &*b*)
Generates upper case string of provided string.

Parameters

- **a** – Input string
- **b** – Input string

Returns The upper case variant of the string

bool **isanynan**(**const** *std*::vector<double> &*x*)
Checks whether at least one of the elements in the vector is not a number.

Parameters **x** – vector of xi values

Returns True, if found at least one NaN: false, otherwise.

double **vectorNorm**(**const** *std*::vector<double> &*x*)
Computes the L2 norm of a vector.

Parameters **x** – vector of xi values

Returns The L2 norm of the vector.

std::string **getTimestamp**()
Obtains the timestamp containing the current data and time.

Returns String containing the timestamp.

size_t **getTimehash**()
Obtains the hash function of timestamp containing the current data and time, for seed initialization purposes.

Returns Unsigned integer containing the hashed timestamp.

char **decimalToHexChar**(**const** uint8_t *byte*)
Converts a decimal byte to its hexadecimal equivalent.

Parameters **byte** – single byte containing a number from 0 to 15

Returns The hexadecimal letter/number for the value

uint8_t **hexCharToDecimal**(**const** char *x*)
Converts a hexadecimal letter/number to integer.

Parameters **x** – the letter/number to convert

Returns A byte with the corresponding integer from 0 to 15

uint8_t **hexPairToByte**(**const** char **src*)
Converts a hexadecimal string pair to integer.

Parameters **src** – the source hexadecimal string format 0xFF to convert

Returns A byte with the corresponding integer from 0 to 255

void **byteToHexPair**(char **dst*, **const** uint8_t *byte*)
Converts an integer to its equivalent hexadecimal string.

Parameters

- **dst** – pointer to string to save the hex string with format 0xFF.
- **byte** – integer containing values from 0 to 255.

template<typename **T**>
double **sign**(*T val*)
Returns the sign of a given signed item.

Parameters **val** – The input signed item.

Returns -1, if val is negative; +1, if val is positive; 0, if neither.

template<typename **T**>
bool **approximatelyEqual**(*T a*, *T b*, *T epsilon*)
Check if both arguments are approximately equal up to given precision.

Parameters

- **a** – Value a
- **b** – Value b
- **epsilon** – Precision parameter

Returns The inverse of the error function

bool **secondSmaller**(**const** *std*::pair<size_t, size_t> &*a*, **const** *std*::pair<size_t, size_t> &*b*)
Check if second element of tuple is smaller.

Parameters

- **a** – Tuple to check if second element is smaller than b
- **b** – Value b

Returns boolean indicating if a.second < b.second

template<typename **T**>
bool **definitelyGreaterThan**(*T a*, *T b*, *T epsilon*)
Check if the first argument is surely greater than the second argument up to given precision.

Parameters

- **a** – First argument, to be checked if greater than b
- **b** – Value b
- **epsilon** – Precision parameter

Returns The inverse of the error function

template<typename **T**>
bool **definitelyLessThan**(*T a*, *T b*, *T epsilon*)
Check if the first argument is surely smaller than the second argument up to given precision.

Parameters

- **a** – First argument, to be checked if smaller than b
- **b** – Value b

- **epsilon** – Precision parameter

Returns The inverse of the error function

template<typename **T**>
T **ierf**(*T x*)

Approximates the inverse of the error function.

Parameters **x** – Argument to the inverse error function

Returns The inverse of the error function

template<typename **T**>
T **safeLogPlus**(*T x*, *T y*)

Safely computes log(exp(x)+exp(y)) and avoids overflows.

Parameters

- **x** – a variable
- **y** – a variable

Returns The result of log(exp(x)+exp(y))

template<typename **T**>
T **safeLogMinus**(*T x*, *T y*)

Safely computes log(exp(x)-exp(y)) and avoids overflows.

Parameters

- **x** – a variable, x > y
- **y** – a variable

Returns The result of log(exp(x)-exp(y))

template<typename **T**>
T **logSumExp**(**const** *T *logValues*, **const** size_t &*n*)

Computes: log sum_{i=1}^N x_i using the log-sum-exp trick: https://en.wikipedia.org/wiki/LogSumExp#log-sum-exp_trick_for_log-domain_calculations.

Parameters

- **logValues** – vector of log(x_i) values
- **n** – size of the vector

Returns The LSE function of the input.

template<typename **T**>
T **logSumExp**(**const** *std*::vector<*T*> &*logValues*)

Computes: log sum_{i=1}^N x_i using the log-sum-exp trick: https://en.wikipedia.org/wiki/LogSumExp#log-sum-exp_trick_for_log-domain_calculations.

Parameters **logValues** – vector of log(x_i) values

Returns The LSE function of the input.

template<typename **T**>
T **dotProduct**(**const** *std*::vector<*T*> &*x*, **const** *std*::vector<*T*> &*y*)

Computes the dot product between two vectors.

Parameters

- **x** – vector of xi values
- **y** – vector of yi values

Returns The x . y product

template<typename **T**>
T **normalLogDensity**(**const** *T &x*, **const** *T &mean*, **const** *T &sigma*)
Computes the log density of a normal distribution.

Parameters

- **x** – denisty evaluation point
- **mean** – Mean of normal distribution
- **sigma** – Standard Deviation of normal distribution

Returns The log density

template<typename **T**>
T **normalCDF**(**const** *T &x*, **const** *T &mean*, **const** *T &sigma*)
Computes the cumulative distribution function of a normal distribution.

Parameters

- **x** – evaluation point
- **mean** – Mean of normal distribution
- **sigma** – Standard Deviation of normal distribution

Returns The log of the CDF

template<typename **T**>
T **normalLogCDF**(**const** *T &x*, **const** *T &mean*, **const** *T &sigma*)
Computes the log of the cumulative distribution function of a normal distribution.

Parameters

- **x** – evaluation point
- **mean** – Mean of normal distribution
- **sigma** – Standard Deviation of normal distribution

Returns The log of the CDF

template<typename **T**>
T **normalCCDF**(**const** *T &x*, **const** *T &mean*, **const** *T &sigma*)
Computes the tail distribution of a normal distribution (complementary cumulative distribution).

Parameters

- **x** – evaluation point
- **mean** – Mean of normal distribution
- **sigma** – Standard Deviation of normal distribution

Returns The log of the CDF

template<typename **T**>
T **normalLogCCDF**(**const** *T &x*, **const** *T &mean*, **const** *T &sigma*)
Computes the log of the tail distribution of a normal distribution (complementary cumulative distribution).

Parameters

- **x** – evaluation point
- **mean** – Mean of normal distribution

- **sigma** – Standard Deviation of normal distribution

Returns The log of the CDF

template<typename **T**>
T **squashedNormalLogDensity**(**const** *T* &*px*, **const** *T* &*mean*, **const** *T* &*sigma*, **const** *T* &*scale*)

Computes the log density of a squashed normal distribution.

Parameters

- **px** – density evaluation point
- **mean** – Mean of normal distribution
- **sigma** – Standard Deviation of normal distribution
- **scale** – The scale used after the tanh normalization

Returns The log density

template<typename **T**>
T **truncatedNormalPdf**(*T x*, *T mu*, *T sigma*, *T a*, *T b*)

Computes the density of the truncated normal distribution.

Parameters

- **x** – density evaluation point
- **mu** – Mean of normal distribution
- **sigma** – Standard Deviation of normal distribution
- **a** – Lower bound of truncated normal
- **b** – Upper bound of truncated normal

Returns The log density

template<typename **T**>
T **betaLogDensity**(**const** *T* &*x*, **const** *T* &*alpha*, **const** *T* &*beta*)

Computes the log density of the beta distribution.

Parameters

- **x** – denisty evaluation point
- **alpha** – Shape of Beta distribution
- **beta** – Shape of Beta distribution

Returns The log density

template<typename **T**>
std::tuple<*T*, *T*> **betaParamTransformAlt**(**const** *T* &*mean*, **const** *T* &*varcof*, **const** *T* &*lb*, **const** *T* &*ub*)

Transforms mean and varcof to alpha and beta for the shifted and scaled beta distribution.

Parameters

- **mean** – Mean of beta distribution
- **varcof** – Variance coefficient (var=mu*(1-mu)*varcof
- **lb** – Lower bound of distribution
- **ub** – Upper bound of distribution

Returns tuple containing alpha and beta

template<typename **T**>
std::tuple<*T*, *T*, *T*, *T*> **derivativesBetaParamTransformAlt**(**const** *T* &*mean*, **const** *T* &*varcof*, **const** *T* &*lb*, **const** *T* &*ub*)

Calculates derivatives of Beta params (alpba,beta) wrt. the params of the alternative parametrization.

Parameters

- **mean** – Mean of alt beta distribution
- **varcof** – Variance coefficient (var=mu*(1-mu)*varcof
- **lb** – Lower bound of distribution
- **ub** – Upper bound of distribution

Returns tuple containing dalpha/dmean, dalpha/dvarcof, dbeta/dmean, dbeta/dvarcof

template<typename **T**>
T **betaLogDensityAlt**(**const** *T* &*x*, **const** *T* &*mean*, **const** *T* &*varcof*, **const** *T* &*lb*, **const** *T* &*ub*)

Computes the log density of the shifted and scaled beta distribution using an alternative four param parametrization.

Parameters

- **x** – denisty evaluation point
- **mean** – Mean of beta distribution
- **varcof** – Variance coefficient (var=mu*(1-mu)*varcof
- **lb** – Lower bound of distribution
- **ub** – Upper bound of distribution

Returns The log density

template<typename **T**>
T **ranBetaAlt**(**const** gsl_rng **rng*, **const** *T* &*mean*, **const** *T* &*varcof*, **const** *T* &*lb*, **const** *T* &*ub*)

Generates a random number from the shifted and scaled beta distribution using an alternative four param parametrization.

Parameters

- **rng** – Gsl random number generator
- **mean** – Mean of beta distribution
- **varcof** – Variance coefficient (var=mu*(1-mu)*varcof
- **lb** – Lower bound of distribution
- **ub** – Upper bound of distribution

Returns a random number

double **vectorDistance**(**const** *std*::vector<double> &*x*, **const** *std*::vector<double> &*y*)

Computes the norm of the difference between two vectors.

Parameters

- **x** – vector of xi values
- **y** – vector of yi values

Returns The L2 norm of the distance of vectors x and y.

void **crcInit**(void)
Initializes the CRC function.

crc **crcFast**(uint8_t **const** *message*[], size_t *nBytes*)
Calculates CRC value of the given byte array.

Parameters

- **message** – Pointer to the start of the byte array
- **nBytes** – Size of the byte array

Returns CRC value of the message

size_t **checksum**(void **buffer*, size_t *len*, unsigned int *seed*)
Checksum function that takes an array of bytes and calculates its CRC given a specific initialization seed.

Parameters

- **buffer** – pointer to the start of the byte array.
- **len** – size of the buffer.
- **seed** – initialization seed for the CRC calculation

Returns The checksum (CRC) of the buffer.

int **setKoraliMPIComm**(...)
Error handler for when MPI is not defined.

Parameters **...** – accepts any parameters since it will fail anyway

Returns Error code -1

void ***getWorkerMPIComm**()
Error handler for when MPI is not defined.

Returns A NULL pointer

static inline *std*::vector<*std*::string> **splitStr**(*std*::string *s*, *std*::string *delim*)
Helper method to split strings.

Parameters

- **s** – the string to split
- **delim** – the delimiter

Returns splitted strings

static inline *std*::string **trimSpaces**(*std*::string *s*)
Helper method to remove spaces let and right.

Parameters **s** – the string to trim

Returns trimed string

static *std*::tuple<*std*::vector<*std*::string>, *std*::vector<int>> **parseSpeciesAndStoichiometricCoeffs**(*std*::string *s*)
Helper method to parse species and stoichiometry coefficients.

Parameters **s** – the string to parse

Returns tuple of species names and stoichiometry coefficients

static inline bool **isReservoir**(*std*::string *name*)
Helper method to check if a variable name is a reservoir.

Parameters **name** – variable name

Returns true if reservoir, else false

static inline bool **containsSpaces** (*std*::string *name*)
Helper method to check if a variable name contains spaces.

Parameters **name** – variable name

Returns true if spaces exist, else false

ParsedReactionString **parseReactionString** (*std*::string *s*)
Parses a string and creates a struct of type *ParsedReactionString*.

Parameters **s** – the reaction equation.

Returns struct containing reaction details.

void **threadWrapper** ( )
Function for the initialization of new coroutine threads.

Variables

const double **NaN** = *std*::numeric_limits<double>::quiet_NaN()
Korali's definition of a non-number.

const double **Inf** = *std*::numeric_limits<double>::infinity()
Korali's definition of Infinity.

const double **Lowest** = *std*::numeric_limits<double>::lowest()
Korali's definition of lowest representable double.

const double **Max** = *std*::numeric_limits<double>::max()
Korali's definition of maximum representable double.

const double **Min** = *std*::numeric_limits<double>::min()
Korali's definition of minimum representable double.

const double **Eps** = *std*::numeric_limits<double>::epsilon()
Korali's definition of minimum representable difference between two numbers.

std::stack<*Engine**> **_engineStack**
Stack storing pointers to different *Engine* execution levels.

bool **isPythonActive** = 0
Flag indicating that *Korali* has been called from *Korali*.

Sample ***_currentSample**
Temporary storage to hold the pointer to the current sample to process.

Experiment ***__expPointer**
Pointer to the current experiment in execution.

cothread_t **__returnThread**
Pointer to the calling thread.

knlohmann::json **__profiler**
Storage for profiling information.

std::chrono::time_point<*std*::chrono::high_resolution_clock> **_startTime**
Start time for the current *Korali* run.

std::chrono::time_point<*std*::chrono::high_resolution_clock> **_endTime**
End time for the current *Korali* run.

double **_cumulativeTime**
Cumulative time for all *Korali* runs during the current application execution.

std::vector<*std*::function<void (*Sample*&) >*> **_functionVector**
Stores all functions inserted as parameters to experiment's configuration.

namespace *korali*::**conduit**

Functions

void **_workerWrapper**()

Variables

Sequential ***_currentConduit**
Temporary storage to hold the pointer to the current conduit.

namespace distribution

namespace multivariate

namespace specific

namespace univariate

namespace neuralNetwork

namespace *korali*::*neuralNetwork*::**layer**

Enums

enum transformation_t
This enumerator details all possible transformations. It is used in lieu of string comparison to accelerate the application of this layer.

Values:

enumerator t_identity
No transformation.

enumerator t_absolute
Apply absolute mask.

enumerator t_softplus
Apply softplus mask.

enumerator t_tanh
Apply tanh mask.

enumerator t_sigmoid
Apply sigmoid mask.

namespace recurrent

namespace *korali*::**problem**

Functions

void **__environmentWrapper**()
Thread wrapper to run an environment.

Variables

Sample ***__currentSample**
Pointer to the current agent, it is immediately copied as to avoid concurrency problems.

size_t **__envFunctionId**
Identifier of the current environment function Id.

solver::*Agent* ***_agent**
Pointer to the agent (*Korali* solver module)

Conduit ***_conduit**
Pointer to the engine's conduit.

cothread_t **_envThread**
Stores the environment thread (coroutine).

size_t **_launchId**
Stores the current launch Id for the current sample.

namespace bayesian

namespace hierarchical

namespace reinforcementLearning

namespace *korali*::**solver**

Enums

enum termination_t
This enumerator details all possible termination statuses for a given episode's experience.

Values:

enumerator e_nonTerminal
The experience is non-terminal.

enumerator e_terminal
This is the terminal experience in a normally executed episode.

enumerator e_truncated
This is the terminal experience in a truncated episode. (i.e., should have continued, but it was artificially truncated to limit running time)

namespace agent

namespace continuous

namespace discrete

namespace integrator

namespace optimizer

namespace *korali*::*solver*::**sampler**

Typedefs

typedef struct *korali*::*solver*::*sampler*::*fparam_s* **fparam_t**
Struct for *TMCMC* optimization operations.

Enums

enum Metric
Enum to set metric type.

Values:

enumerator Static
Static Metric type.

enumerator Euclidean
Euclidean Metric type.

enumerator Riemannian
Riemannian Metric type.

enumerator Riemannian_Const
Const Riemannian Metric type.

namespace ssm

namespace layer
Namespace declaration for modules of type: layer.

namespace multivariate
Namespace declaration for modules of type: multivariate.

namespace neuralNetwork
Namespace declaration for modules of type: *neuralNetwork*.

namespace optimizer
Namespace declaration for modules of type: optimizer.

namespace problem
Namespace declaration for modules of type: problem.

namespace recurrent
Namespace declaration for modules of type: recurrent.

namespace reinforcementLearning
Namespace declaration for modules of type: *reinforcementLearning*.

namespace Rtnorm

namespace sampler
Namespace declaration for modules of type: sampler.

namespace solver
Namespace declaration for modules of type: solver.

namespace specific
Namespace declaration for modules of type: specific.

namespace ssm
Namespace declaration for modules of type: ssm.

namespace std

namespace univariate
Namespace declaration for modules of type: univariate.

file **cbuffer.hpp**
#include <>#include <> Implements a circular buffer with automatic overwrite on full by Sergio Martin (2020), partially based on the implementation by Jose Herrera https://gist.github.com/xstherrera1987/3196485.

file **cudaUtils.hpp**
Contains auxiliar error reporting functions to CUDA and cuDNN Credits to Motoki Sato (https://gist.github.com/aonotas)

file **fs.cpp**
*#include "*auxiliar/fs.hpp*"#include ""#include <>#include <>#include <>*

file **fs.hpp**
#include <>#include <> Contains auxiliar code for file system (files and folders) manipulation.

file **jsonInterface.cpp**
*#include "*auxiliar/jsonInterface.hpp*"#include "*auxiliar/logger.hpp*"#include <>#include <>* Contains auxiliar functions for JSON object manipulation.

file **jsonInterface.hpp**
*#include ""#include "*auxiliar/logger.hpp*"#include <>* Contains auxiliar functions for JSON object manipulation.

file **kcache.hpp**
#include <>#include <> Implements an LRU cache that returns a pre-calculated value if it is not too old. Age is determined by an external timer. Mutual exclusion mechanisms have been added for thread-safe access. by Sergio Martin (2020)

file **koraliJson.cpp**
*#include "*auxiliar/koraliJson.hpp*"#include "*auxiliar/logger.hpp*"#include ""#include "*sample/sample.hpp*"*

file **koraliJson.hpp**
*#include "*auxiliar/jsonInterface.hpp*"#include <>#include <>#include <>#include <>* Contains the koraliJson class, which supports JSON objects within *Korali* classes.

file **kstring.cpp**
*#include "*kstring.hpp*"#include <>#include <>#include <>*

file **kstring.hpp**
#include <> Auxiliary library for *Korali*'s essential string operations.

file **logger.cpp**
*#include "*logger.hpp*"#include <>#include <>#include <>#include <>*

file **logger.hpp**
#include <> Contains functions to manage file and console output, verbosity levels, and error reporting.

Defines

KORALI_LOG_ERROR(...)
Terminates execution, printing an error message and indicates file name and line number where the error occurred.

file **math.cpp**
*#include "*math.hpp*"#include "*auxiliar/logger.hpp*"#include <>#include <>#include <>#include <>*

file **math.hpp**
#include <>#include <>#include <>#include <>#include <>#include <>#include <>#include <>

<>*#include* <>*#include* <>*#include* <>*#include* <> Auxiliary library for *Korali*'s essential math and time manipulation operations.

Defines

_USE_MATH_DEFINES
This definition enables the use of M_PI.

KORALI_EPSILON
Epsilon to add to log or division operations to prevent numerical instabilities.

POLYNOMIAL
Polynomial for CRC calculation.

WIDTH
Width of CRC calculation.

TOPBIT
Most significant bit of a CRC calculation.

file **MPIUtils.cpp**
#include <auxiliar/MPIUtils.hpp>*#include* <auxiliar/logger.hpp> Contains the helper definitions for MPI.

file **MPIUtils.hpp**
#include <>*#include* <>*#include* <>*#include* <> Contains the helper definitions for MPI.

file **reactionParser.cpp**
#include "reactionParser.hpp"*#include* "logger.hpp"*#include* <>

file **reactionParser.hpp**
#include <>*#include* <>*#include* <> Implements a parser for reaction equations based on the implementation by Luca Amoudruz https://github.com/amlucas/SSM.

file **engine.cpp**
#include "engine.hpp"*#include* "auxiliar/fs.hpp"*#include* "auxiliar/koraliJson.hpp"*#include* "modules/conduit/conduit.hpp"*#include* "modules/experiment/experiment.hpp"*#include* "modules/conduit/distributed/distributed.hpp"*#include* "modules/problem/problem.hpp"*#include* "modules/solver/solver.hpp"*#include* "sample/sample.hpp"*#include* <>*#include* <>

Functions

PYBIND11_MODULE(libkorali, m)

file **engine.hpp**
#include ""*#include* "auxiliar/MPIUtils.hpp"*#include* "modules/conduit/conduit.hpp"*#include* "modules/conduit/distributed/distributed.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>*#include* <>*#include* <>*#include* <> Include header for the *Korali* Engine.

file **korali.hpp**
#include ""*#include* "engine.hpp"*#include* "modules/conduit/distributed/distributed.hpp"*#include* "modules/experiment/experiment.hpp"*#include* "sample/sample.hpp" Include header for C++ applications linking with *Korali*.

file **concurrent.cpp**
#include "engine.hpp"*#include* "modules/conduit/concurrent/concurrent.hpp"*#include* "modules/experiment/experiment.hpp"*#include* "modules/problem/problem.hpp"*#include* "modules/solver/solver.hpp"*#include* "sample/sample.hpp"*#include* <>*#include* <>*#include* <>*#include* <>

Defines

BUFFERSIZE

file **concurrent.hpp**
#include “modules/conduit/conduit.hpp”*#include* <>*#include* <>*#include* <> Header file for module: Concurrent.

file **conduit.cpp**
#include “engine.hpp”*#include* “modules/conduit/conduit.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “sample/sample.hpp”*#include* <>

file **conduit.hpp**
#include “modules/module.hpp”*#include* <>*#include* <> Header file for module: Conduit.

file **distributed.cpp**
#include “auxiliar/MPIUtils.hpp”*#include* “engine.hpp”*#include* “modules/conduit/distributed/distributed.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/problem/problem.hpp”*#include* “modules/solver/solver.hpp”*#include* “sample/sample.hpp”

file **distributed.hpp**
#include “auxiliar/MPIUtils.hpp”*#include* “”*#include* “modules/conduit/conduit.hpp”*#include* <>*#include* <>*#include* <> Header file for module: Distributed.

file **sequential.cpp**
#include “engine.hpp”*#include* “modules/conduit/sequential/sequential.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/problem/problem.hpp”*#include* “modules/solver/solver.hpp”*#include* “sample/sample.hpp”*#include* <>*#include* <>*#include* <>*#include* <>

file **sequential.hpp**
#include “”*#include* “modules/conduit/conduit.hpp”*#include* <>*#include* <>*#include* <>*#include* <> Header file for module: Sequential.

file **distribution.cpp**
#include “modules/distribution/distribution.hpp”*#include* “modules/experiment/experiment.hpp”*#include* <>*#include* <>*#include* <>*#include* <>

file **distribution.hpp**
#include “modules/module.hpp”*#include* <>*#include* <> Header file for module: Distribution.

file **multivariate.cpp**
#include “modules/distribution/multivariate/multivariate.hpp”*#include* “modules/experiment/experiment.hpp”

file **multivariate.hpp**
#include “modules/distribution/distribution.hpp” Header file for module: Multivariate.

file **normal.cpp**
#include “modules/distribution/multivariate/normal/normal.hpp”*#include* “modules/experiment/experiment.hpp”*#include* <auxiliar/logger.hpp>*#include* <>*#include* <>*#include* <>

file **normal.cpp**
#include “modules/distribution/univariate/normal/normal.hpp”*#include* “modules/experiment/experiment.hpp”*#include* <>*#include* <>*#include* <>

file **normal.hpp**
#include “modules/distribution/multivariate/multivariate.hpp”*#include* <>*#include* <> Header file for module: Normal.

file **normal.hpp**
#include “modules/distribution/univariate/univariate.hpp” Header file for module: Normal.

file **multinomial.cpp**
#include "modules/distribution/specific/multinomial/multinomial.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>

file **multinomial.hpp**
#include "modules/distribution/specific/specific.hpp" Header file for module: Multinomial.

file **specific.cpp**
#include "modules/distribution/specific/specific.hpp"*#include* "modules/experiment/experiment.hpp"

file **specific.hpp**
#include "modules/distribution/distribution.hpp" Header file for module: Specific.

file **beta.cpp**
#include "modules/distribution/univariate/beta/beta.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>

file **beta.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Beta.

file **cauchy.cpp**
#include "modules/distribution/univariate/cauchy/cauchy.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>*#include* <>

file **cauchy.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Cauchy.

file **exponential.cpp**
#include "modules/distribution/univariate/exponential/exponential.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>

file **exponential.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Exponential.

file **gamma.cpp**
#include "modules/distribution/univariate/gamma/gamma.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>

file **gamma.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Gamma.

file **geometric.cpp**
#include "modules/distribution/univariate/geometric/geometric.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>

file **geometric.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Geometric.

file **igamma.cpp**
#include "modules/distribution/univariate/igamma/igamma.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>

file **igamma.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Igamma.

file **laplace.cpp**
#include "modules/distribution/univariate/laplace/laplace.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>*#include* <>

file **laplace.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Laplace.

file **logNormal.cpp**
#include "modules/distribution/univariate/logNormal/logNormal.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>*#include* <>

file **logNormal.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: LogNormal.

file **poisson.cpp**
#include "modules/distribution/univariate/poisson/poisson.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>*#include* <>

file **poisson.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Poisson.

file **truncatedNormal.cpp**
#include "modules/distribution/univariate/truncatedNormal/truncatedNormal.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>*#include* <>*#include* ""

file **truncatedNormal.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: TruncatedNormal.

file **uniform.cpp**
#include "modules/distribution/univariate/uniform/uniform.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>*#include* <>

file **uniform.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Uniform.

file **uniformratio.cpp**
#include "modules/distribution/univariate/uniformratio/uniformratio.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>*#include* <>

file **uniformratio.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: UniformRatio.

file **univariate.cpp**
#include "modules/distribution/univariate/univariate.hpp"*#include* "modules/experiment/experiment.hpp"

file **univariate.hpp**
#include "modules/distribution/distribution.hpp" Header file for module: Univariate.

file **weibull.cpp**
#include "modules/distribution/univariate/weibull/weibull.hpp"*#include* "modules/experiment/experiment.hpp"*#include* <>*#include* <>

file **weibull.hpp**
#include "modules/distribution/univariate/univariate.hpp" Header file for module: Weibull.

file **experiment.cpp**
#include "auxiliar/fs.hpp"*#include* "auxiliar/koraliJson.hpp"*#include* "engine.hpp"*#include* "modules/conduit/conduit.hpp"*#include* "modules/conduit/distributed/distributed.hpp"*#include* "modules/experiment/experiment.hpp"*#include* "modules/problem/problem.hpp"*#include* "modules/solver/agent/agent.hpp"*#include* "modules/solver/deepSupervisor/deepSupervisor.hpp"*#include* "modules/solver/solver.hpp"*#include* "sample/sample.hpp"*#include* <>*#include* <>*#include* <>*#include* <>

file **experiment.hpp**
#include "auxiliar/koraliJson.hpp"*#include* ""*#include* ""*#include* "modules/module.hpp"*#include* ""*#include* <>*#include* <>*#include* <> Header file for module: Experiment.

file **module.cpp**
#include "module.hpp"*#include* "conduit/concurrent/concurrent.hpp"*#include* "conduit/distributed/distributed.hpp"*#include* "conduit/sequential/sequential.hpp"*#include* "distribu-

tion/distribution.hpp"#include "distribution/multivariate/normal/normal.hpp"#include "distribution/specific/multinomial/multinomial.hpp"#include "distribution/specific/specific.hpp"#include "distribution/univariate/beta/beta.hpp"#include "distribution/univariate/cauchy/cauchy.hpp"#include "distribution/univariate/exponential/exponential.hpp"#include "distribution/univariate/gamma/gamma.hpp"#include "distribution/univariate/geometric/geometric.hpp"#include "distribution/univariate/igamma/igamma.hpp"#include "distribution/univariate/laplace/laplace.hpp"#include "distribution/univariate/logNormal/logNormal.hpp"#include "distribution/univariate/normal/normal.hpp"#include "distribution/univariate/poisson/poisson.hpp"#include "distribution/univariate/truncatedNormal/truncatedNormal.hpp"#include "distribution/univariate/uniform/uniform.hpp"#include "distribution/univariate/uniformratio/uniformratio.hpp"#include "distribution/univariate/weibull/weibull.hpp"#include "experiment/experiment.hpp"#include "neuralNetwork/layer/activation/activation.hpp"#include "neuralNetwork/layer/convolution/convolution.hpp"#include "neuralNetwork/layer/deconvolution/deconvolution.hpp"#include "neuralNetwork/layer/pooling/pooling.hpp"#include "neuralNetwork/layer/input/input.hpp"#include "neuralNetwork/layer/layer.hpp"#include "neuralNetwork/layer/linear/linear.hpp"#include "neuralNetwork/layer/output/output.hpp"#include "neuralNetwork/layer/recurrent/gru/gru.hpp"#include "neuralNetwork/layer/recurrent/lstm/lstm.hpp"#include "neuralNetwork/neuralNetwork.hpp"#include "problem/bayesian/custom/custom.hpp"#include "problem/bayesian/reference/reference.hpp"#include "problem/design/design.hpp"#include "problem/hierarchical/psi/psi.hpp"#include "problem/hierarchical/theta/theta.hpp"#include "problem/hierarchical/thetaNew/thetaNew.hpp"#include "problem/integration/integration.hpp"#include "problem/optimization/optimization.hpp"#include "problem/problem.hpp"#include "problem/propagation/propagation.hpp"#include "problem/reinforcementLearning/continuous/continuous.hpp"#include "problem/reinforcementLearning/discrete/discrete.hpp"#include "problem/sampling/sampling.hpp"#include "problem/supervisedLearning/supervisedLearning.hpp"#include "problem/reaction/reaction.hpp"#include "solver/agent/continuous/VRACER/VRACER.hpp"#include "solver/agent/continuous/continuous.hpp"#include "solver/agent/discrete/dVRACER/dVRACER.hpp"#include "solver/agent/discrete/discrete.hpp"#include "solver/designer/designer.hpp"#include "solver/executor/executor.hpp"#include "solver/integrator/integrator.hpp"#include "solver/integrator/montecarlo/MonteCarlo.hpp"#include "solver/integrator/quadrature/Quadrature.hpp"#include "solver/deepSupervisor/deepSupervisor.hpp"#include "solver/deepSupervisor/optimizers/fAdam/fAdam.hpp"#include "solver/deepSupervisor/optimizers/fAdaBelief/fAdaBelief.hpp"#include "solver/deepSupervisor/optimizers/fMadGrad/fMadGrad.hpp"#include "solver/deepSupervisor/optimizers/fAdaGrad/fAdaGrad.h#include "solver/optimizer/AdaBelief/AdaBelief.hpp"#include "solver/optimizer/Adam/Adam.hpp"#include "solver/optimizer/CMAES/CMAES.hpp"#include "solver/optimizer/DEA/DEA.hpp"#include "solver/optimizer/MADGRAD/MADGRAD.hpp"#include "solver/optimizer/MOCMAES/MOCMAES.hpp"#include "solver/optimizer/Rprop/Rprop.hpp"#include "solver/optimizer/gridSearch/gridSearch.hpp"#include "solver/optimizer/optimizer.hpp"#include "solver/sampler/HMC/HMC.hpp"#include "solver/sampler/MCMC/MCMC.hpp"#include "solver/sampler/Nested/Nested.hpp"#include "solver/sampler/TMCMC/TMCMC.hpp"#include "solver/sampler/sampler.hpp"#include "solver/SSM/SSA/SSA.hpp"#include "solver/SSM/TauLeaping/TauLeaping.hpp"#include "solver/SSM/SSM.hpp"

file **module.hpp**

#include "auxiliar/koraliJson.hpp"#include "auxiliar/kstring.hpp"#include "auxiliar/logger.hpp"#include "auxiliar/math.hpp"#include <> Header file for the base *Korali* Module class.

file **activation.cpp**

#include "modules/neuralNetwork/layer/activation/activation.hpp"#include "modules/neuralNetwork/neuralNetwork.hpp"#include <>

file **activation.hpp**

#include "modules/neuralNetwork/layer/layer.hpp" Header file for module: Activation.

file **convolution.cpp**

#include "modules/neuralNetwork/layer/convolution/convolution.hpp"#include "modules/neuralNetwork/neuralNetwork.hpp"#include <>

file **convolution.hpp**
#include “modules/neuralNetwork/layer/layer.hpp” Header file for module: Convolution.

file **deconvolution.cpp**
#include “modules/neuralNetwork/layer/deconvolution/deconvolution.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* <>

file **deconvolution.hpp**
#include “modules/neuralNetwork/layer/layer.hpp” Header file for module: Deconvolution.

file **input.cpp**
#include “modules/neuralNetwork/layer/input/input.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* <>

file **input.hpp**
#include “modules/neuralNetwork/layer/layer.hpp” Header file for module: Input.

file **layer.cpp**
#include “modules/neuralNetwork/layer/layer.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* <>

file **layer.hpp**
#include “”*#include* “modules/distribution/univariate/uniform/uniform.hpp”*#include* “modules/module.hpp” Header file for module: Layer.

file **linear.cpp**
#include “modules/neuralNetwork/layer/linear/linear.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* <>

file **linear.hpp**
#include “modules/neuralNetwork/layer/layer.hpp” Header file for module: Linear.

file **output.cpp**
#include “modules/neuralNetwork/layer/output/output.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* <>

file **output.hpp**
#include “modules/neuralNetwork/layer/layer.hpp” Header file for module: Output.

file **pooling.cpp**
#include “modules/neuralNetwork/layer/pooling/pooling.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* <>

file **pooling.hpp**
#include “modules/neuralNetwork/layer/layer.hpp” Header file for module: Pooling.

file **gru.cpp**
#include “modules/neuralNetwork/layer/recurrent/gru/gru.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* <>

file **gru.hpp**
#include “modules/neuralNetwork/layer/recurrent/recurrent.hpp” Header file for module: GRU.

file **lstm.cpp**
#include “modules/neuralNetwork/layer/recurrent/lstm/lstm.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* <>

file **lstm.hpp**
#include “modules/neuralNetwork/layer/recurrent/recurrent.hpp” Header file for module: LSTM.

file **recurrent.cpp**
#include “modules/neuralNetwork/layer/recurrent/recurrent.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* <>

file **recurrent.hpp**
#include “modules/neuralNetwork/layer/layer.hpp” Header file for module: Recurrent.

file **neuralNetwork.cpp**
#include “modules/experiment/experiment.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”

file **neuralNetwork.hpp**
#include “”*#include* “modules/experiment/experiment.hpp”*#include* “modules/module.hpp”*#include* “modules/neuralNetwork/layer/layer.hpp”*#include* “modules/solver/solver.hpp” Header file for module: NeuralNetwork.

file **bayesian.cpp**
#include “modules/problem/bayesian/bayesian.hpp”*#include* “sample/sample.hpp”

file **bayesian.hpp**
#include “modules/problem/problem.hpp” Header file for module: Bayesian.

file **custom.cpp**
#include “modules/conduit/conduit.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/problem/bayesian/custom/custom.hpp”*#include* “sample/sample.hpp”

file **custom.hpp**
#include “modules/problem/bayesian/bayesian.hpp” Header file for module: Custom.

file **reference.cpp**
#include “modules/conduit/conduit.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/problem/bayesian/reference/reference.hpp”*#include* “sample/sample.hpp”*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>

Defines

STDEV_EPSILON

file **reference.hpp**
#include “modules/problem/bayesian/bayesian.hpp”*#include* <> Header file for module: Reference.

file **design.cpp**
#include “modules/problem/design/design.hpp”*#include* “sample/sample.hpp”

file **design.hpp**
#include “modules/problem/problem.hpp” Header file for module: Design.

file **hierarchical.cpp**
#include “modules/problem/hierarchical/hierarchical.hpp”*#include* “sample/sample.hpp”

file **hierarchical.hpp**
#include “modules/problem/problem.hpp” Header file for module: Hierarchical.

file **psi.cpp**
#include “modules/conduit/conduit.hpp”*#include* “modules/distribution/univariate/normal/normal.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/problem/hierarchical/psi/psi.hpp”*#include* “sample/sample.hpp”

file **psi.hpp**
#include “modules/distribution/distribution.hpp”*#include* “modules/problem/hierarchical/hierarchical.hpp” Header file for module: Psi.

file **theta.cpp**
#include "modules/conduit/conduit.hpp"*#include* "modules/problem/hierarchical/theta/theta.hpp"*#include* "sample/sample.hpp"

file **theta.hpp**
#include "modules/problem/bayesian/bayesian.hpp"*#include* "modules/problem/hierarchical/psi/psi.hpp" Header file for module: Theta.

file **thetaNew.cpp**
#include "modules/conduit/conduit.hpp"*#include* "modules/problem/hierarchical/thetaNew/thetaNew.hpp"*#include* "sample/sample.hpp"

file **thetaNew.hpp**
#include "modules/problem/hierarchical/hierarchical.hpp"*#include* "modules/problem/hierarchical/psi/psi.hpp" Header file for module: ThetaNew.

file **integration.cpp**
#include "modules/problem/integration/integration.hpp"*#include* "sample/sample.hpp"

file **integration.hpp**
#include "modules/problem/problem.hpp" Header file for module: Integration.

file **optimization.cpp**
#include "modules/problem/optimization/optimization.hpp"*#include* "sample/sample.hpp"

file **optimization.hpp**
#include "modules/problem/problem.hpp" Header file for module: Optimization.

file **problem.cpp**
#include "modules/problem/problem.hpp"

file **problem.hpp**
#include "modules/experiment/experiment.hpp"*#include* "modules/module.hpp" Header file for module: Problem.

file **propagation.cpp**
#include "modules/problem/propagation/propagation.hpp"*#include* "sample/sample.hpp"

file **propagation.hpp**
#include "modules/problem/problem.hpp" Header file for module: Propagation.

file **reaction.cpp**
#include "modules/problem/reaction/reaction.hpp"*#include* "sample/sample.hpp"

file **reaction.hpp**
#include "auxiliar/reactionParser.hpp"*#include* "modules/problem/problem.hpp" Header file for module: Reaction.

file **continuous.cpp**
#include "modules/problem/reinforcementLearning/continuous/continuous.hpp"*#include* "modules/solver/agent/continuous/continuous.hpp"*#include* "sample/sample.hpp"

file **continuous.cpp**
#include "engine.hpp"*#include* "modules/solver/agent/continuous/continuous.hpp"*#include* "sample/sample.hpp"*#include* <>

file **continuous.hpp**
#include "modules/distribution/univariate/normal/normal.hpp"*#include* "modules/problem/reinforcementLearning/reinforcementLearning.hpp" Header file for module: Continuous.

file **continuous.hpp**
#include "modules/distribution/univariate/beta/beta.hpp"*#include* "modules/problem/reinforcementLearning/continuous/continuo
"modules/solver/agent/agent.hpp" Header file for module: Continuous.

file **discrete.cpp**
#include "modules/problem/reinforcementLearning/discrete/discrete.hpp"*#include* "modules/solver/agent/discrete/discrete.hpp"*#include* "sample/sample.hpp"

file **discrete.cpp**
#include "engine.hpp"*#include* "modules/solver/agent/discrete/discrete.hpp"*#include* "sample/sample.hpp"

file **discrete.hpp**
#include "modules/problem/reinforcementLearning/reinforcementLearning.hpp" Header file for module: Discrete.

file **discrete.hpp**
#include "modules/problem/reinforcementLearning/discrete/discrete.hpp"*#include* "modules/solver/agent/agent.hpp" Header file for module: Discrete.

file **reinforcementLearning.cpp**
#include "engine.hpp"*#include* "modules/problem/reinforcementLearning/reinforcementLearning.hpp"*#include*
"modules/solver/agent/agent.hpp"*#include* "sample/sample.hpp"

file **reinforcementLearning.hpp**
#include "modules/distribution/univariate/uniform/uniform.hpp"*#include* "modules/neuralNetwork/neuralNetwork.hpp"*#include* "modules/problem/problem.hpp" Header file for module: ReinforcementLearning.

file **sampling.cpp**
#include "modules/problem/sampling/sampling.hpp"*#include* "sample/sample.hpp"

file **sampling.hpp**
#include "modules/problem/problem.hpp" Header file for module: Sampling.

file **supervisedLearning.cpp**
#include "modules/problem/supervisedLearning/supervisedLearning.hpp"

file **supervisedLearning.hpp**
#include "modules/problem/problem.hpp" Header file for module: SupervisedLearning.

file **agent.cpp**
#include "auxiliar/fs.hpp"*#include* "engine.hpp"*#include* "modules/solver/agent/agent.hpp"*#include* "sample/sample.hpp"*#include* <>

file **agent.hpp**
#include "auxiliar/cbuffer.hpp"*#include* "modules/problem/reinforcementLearning/reinforcementLearning.hpp"*#include*
"modules/problem/supervisedLearning/supervisedLearning.hpp"*#include* "modules/solver/deepSupervisor/deepSupervisor.hpp"*#include* "sample/sample.hpp"*#include* <>*#include* <>
Header file for module: Agent.

file **VRACER.cpp**
#include "engine.hpp"*#include* "modules/solver/agent/continuous/VRACER/VRACER.hpp"*#include* "sample/sample.hpp"*#include* <>

file **VRACER.hpp**
#include "modules/distribution/univariate/normal/normal.hpp"*#include* "modules/problem/reinforcementLearning/continuous/continuous.hpp"*#include* "modules/solver/agent/continuous/continuous.hpp" Header file for module: VRACER.

file **dVRACER.cpp**

#include “engine.hpp”*#include* “modules/solver/agent/discrete/dVRACER/dVRACER.hpp”*#include* “sample/sample.hpp”

file **dVRACER.hpp**
#include “modules/distribution/univariate/normal/normal.hpp”*#include* “modules/problem/reinforcementLearning/discrete/discrete.hpp”*#include* “modules/solver/agent/discrete/discrete.hpp” Header file for module: dVRACER.

file **deepSupervisor.cpp**
#include “engine.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/solver/deepSupervisor/deepSupervisor.hpp”*#include* “sample/sample.hpp”

file **deepSupervisor.hpp**
#include “modules/experiment/experiment.hpp”*#include* “modules/neuralNetwork/neuralNetwork.hpp”*#include* “modules/problem/supervisedLearning/supervisedLearning.hpp”*#include* “modules/solver/deepSupervisor/optimizers/fGradientBasedOptimizer.hpp” Header file for module: DeepSupervisor.

file **fAdaBelief.cpp**
#include “modules/solver/deepSupervisor/optimizers/fAdaBelief/fAdaBelief.hpp”

file **fAdaBelief.hpp**
#include “modules/solver/deepSupervisor/optimizers/fAdam/fAdam.hpp”*#include* “modules/solver/deepSupervisor/optimizers/fGradientBasedOptimizer.hpp” Header file for module: fAdaBelief.

file **fAdaGrad.cpp**
#include “modules/solver/deepSupervisor/optimizers/fAdaGrad/fAdaGrad.hpp”

file **fAdaGrad.hpp**
#include “modules/solver/deepSupervisor/optimizers/fGradientBasedOptimizer.hpp” Header file for module: fAdaGrad.

file **fAdam.cpp**
#include “modules/solver/deepSupervisor/optimizers/fAdam/fAdam.hpp”

file **fAdam.hpp**
#include “modules/solver/deepSupervisor/optimizers/fGradientBasedOptimizer.hpp” Header file for module: fAdam.

file **fGradientBasedOptimizer.cpp**
#include “modules/solver/deepSupervisor/optimizers/fGradientBasedOptimizer.hpp”

file **fGradientBasedOptimizer.hpp**
#include “auxiliar/logger.hpp”*#include* “modules/module.hpp”*#include* <>*#include* <>*#include* <>*#include* <> Header file for module: fGradientBasedOptimizer.

file **fMadGrad.cpp**
#include “modules/solver/deepSupervisor/optimizers/fMadGrad/fMadGrad.hpp”

file **fMadGrad.hpp**
#include “modules/solver/deepSupervisor/optimizers/fGradientBasedOptimizer.hpp” Header file for module: fMadGrad.

file **designer.cpp**
#include “engine.hpp”*#include* “modules/solver/designer/designer.hpp”

file **designer.hpp**
#include “modules/distribution/univariate/normal/normal.hpp”*#include* “modules/problem/design/design.hpp”*#include* “modules/solver/solver.hpp”*#include* “sample/sample.hpp” Header file for module: Designer.

file **executor.cpp**
#include “engine.hpp”*#include* “modules/solver/executor/executor.hpp”*#include* “sample/sample.hpp”

file **executor.hpp**
#include “modules/solver/solver.hpp” Header file for module: Executor.

file **integrator.cpp**
#include “engine.hpp”*#include* “modules/solver/integrator/integrator.hpp”

file **integrator.hpp**
#include “modules/solver/solver.hpp”*#include* “sample/sample.hpp” Header file for module: Integrator.

file **MonteCarlo.cpp**
#include “engine.hpp”*#include* “modules/solver/integrator/montecarlo/MonteCarlo.hpp”

file **MonteCarlo.hpp**
#include “modules/distribution/univariate/uniform/uniform.hpp”*#include* “modules/solver/integrator/integrator.hpp”*#include* “modules/solver/solver.hpp” Header file for module: MonteCarlo.

file **Quadrature.cpp**
#include “engine.hpp”*#include* “modules/solver/integrator/quadrature/Quadrature.hpp”

file **Quadrature.hpp**
#include “modules/solver/integrator/integrator.hpp”*#include* “modules/solver/solver.hpp” Header file for module: Quadrature.

file **AdaBelief.cpp**
#include “engine.hpp”*#include* “modules/solver/optimizer/AdaBelief/AdaBelief.hpp”*#include* “sample/sample.hpp”

file **AdaBelief.hpp**
#include “modules/solver/optimizer/optimizer.hpp” Header file for module: AdaBelief.

file **Adam.cpp**
#include “engine.hpp”*#include* “modules/solver/optimizer/Adam/Adam.hpp”*#include* “sample/sample.hpp”

file **Adam.hpp**
#include “modules/solver/optimizer/optimizer.hpp” Header file for module: Adam.

file **CMAES.cpp**
#include “engine.hpp”*#include* “modules/solver/optimizer/CMAES/CMAES.hpp”*#include* “sample/sample.hpp”*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>

file **CMAES.hpp**
#include “modules/distribution/univariate/normal/normal.hpp”*#include* “modules/distribution/univariate/uniform/uniform.hpp”*#include* “modules/solver/optimizer/optimizer.hpp”*#include* <> Header file for module: CMAES.

file **DEA.cpp**
#include “engine.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/problem/problem.hpp”*#include* “modules/solver/optimizer/DEA/DEA.hpp”*#include* “sample/sample.hpp”*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>

file **DEA.hpp**
#include “modules/distribution/univariate/normal/normal.hpp”*#include* “modules/distribution/univariate/uniform/uniform.hpp”*#include* “modules/solver/optimizer/optimizer.hpp”*#include* <> Header file for module: DEA.

file **gridSearch.cpp**
#include “engine.hpp”*#include* “modules/solver/optimizer/gridSearch/gridSearch.hpp”*#include* “sample/sample.hpp”

file **gridSearch.hpp**
#include "modules/solver/optimizer/optimizer.hpp" Header file for module: GridSearch.

file **MADGRAD.cpp**
#include "engine.hpp"*#include* "modules/solver/optimizer/MADGRAD/MADGRAD.hpp"*#include* "sample/sample.hpp"

file **MADGRAD.hpp**
#include "modules/solver/optimizer/optimizer.hpp" Header file for module: MADGRAD.

file **MOCMAES.cpp**
#include "engine.hpp"*#include* "modules/problem/optimization/optimization.hpp"*#include* "modules/solver/optimizer/MOCMAES/MOCMAES.hpp"*#include* "sample/sample.hpp"*#include* <>

file **MOCMAES.hpp**
#include "modules/distribution/multivariate/normal/normal.hpp"*#include* "modules/distribution/univariate/uniform/uniform.hpp"*#include* "modules/solver/optimizer/optimizer.hpp"*#include* <> Header file for module: MOCMAES.

file **optimizer.cpp**
#include "modules/solver/optimizer/optimizer.hpp"

file **optimizer.hpp**
#include "modules/solver/solver.hpp" Header file for module: Optimizer.

file **Rprop.cpp**
#include "engine.hpp"*#include* "modules/experiment/experiment.hpp"*#include* "modules/problem/problem.hpp"*#include* "modules/solver/optimizer/Rprop/Rprop.hpp"*#include* "sample/sample.hpp"*#include* <>

file **Rprop.hpp**
#include "modules/solver/optimizer/optimizer.hpp"*#include* <> Header file for module: Rprop.

file **hamiltonian_base.hpp**
#include "modules/conduit/conduit.hpp"*#include* "engine.hpp"*#include* "modules/experiment/experiment.hpp"*#include* "modules/problem/bayesian/bayesian.hpp"*#include* "modules/problem/bayesian/reference/reference.hpp"*#include* "modules/problem/problem.hpp"*#include* "modules/problem/sampling/sampling.hpp"*#include* "sample/sample.hpp"

file **hamiltonian_euclidean_base.hpp**
#include "hamiltonian_base.hpp"*#include* "modules/conduit/conduit.hpp"*#include* "engine.hpp"*#include* "modules/experiment/experiment.hpp"*#include* "modules/problem/problem.hpp"*#include* "modules/solver/sampler/MCMC/MCMC.hpp"*#include* "sample/sample.hpp"

file **hamiltonian_euclidean_dense.hpp**
#include "hamiltonian_euclidean_base.hpp"*#include* "modules/distribution/multivariate/normal/normal.hpp"*#include* <>*#include* <>*#include* <>*#include* <>

file **hamiltonian_euclidean_diag.hpp**
#include "hamiltonian_euclidean_base.hpp"*#include* "modules/distribution/univariate/normal/normal.hpp"

file **hamiltonian_riemannian_base.hpp**
#include "hamiltonian_base.hpp"*#include* "modules/conduit/conduit.hpp"*#include* "engine.hpp"*#include* "modules/experiment/experiment.hpp"*#include* "modules/problem/problem.hpp"*#include* "modules/solver/sampler/MCMC/MCMC.hpp"*#include* "sample/sample.hpp"

file **hamiltonian_riemannian_const_dense.hpp**
#include "hamiltonian_riemannian_base.hpp"*#include* "modules/distribution/multivariate/normal/normal.hpp"*#include* <>*#include* <>

file **hamiltonian_riemannian_const_diag.hpp**
#include “hamiltonian_riemannian_base.hpp”*#include* “modules/distribution/univariate/normal/normal.hpp”

file **hamiltonian_riemannian_diag.hpp**
#include “hamiltonian_riemannian_base.hpp”*#include* “modules/distribution/univariate/normal/normal.hpp”

file **leapfrog_base.hpp**
#include “hamiltonian_base.hpp”*#include* <>*#include* <>

file **leapfrog_explicit.hpp**
#include “hamiltonian_base.hpp”*#include* “leapfrog_base.hpp”

file **leapfrog_implicit.hpp**
#include “engine.hpp”*#include* “hamiltonian_base.hpp”*#include* “hamiltonian_riemannian_base.hpp”*#include* “leapfrog_base.hpp”*#include* <>

file **tree_helper_base.hpp**
#include <>

file **tree_helper_euclidean.hpp**
#include “tree_helper_base.hpp”

file **tree_helper_riemannian.hpp**
#include “tree_helper_base.hpp”

file **HMC.cpp**
#include “auxiliar/math.hpp”*#include* “modules/conduit/conduit.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/problem/problem.hpp”*#include* “modules/solver/sampler/HMC/HMC.hpp”*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>

file **HMC.hpp**
#include “modules/distribution/multivariate/normal/normal.hpp”*#include* “modules/distribution/univariate/normal/normal.hpp”*#include* “modules/distribution/univariate/uniform/uniform.hpp”*#include* “modules/solver/sampler/sampler.hpp”*#include* <>*#include* <>*#include* “modules/solver/sampler/HMC/helpers/hamiltonian_euclidean_dense.hpp”*#include* “modules/solver/sampler/HMC/helpers/hamiltonian_euclidean_diag.hpp”*#include* “modules/solver/sampler/HMC/helpers/hamiltonian_riemannian_const_dense.hpp”*#include* “modules/solver/sampler/HMC/helpers/hamiltonian_riemannian_const_diag.hpp”*#include* “modules/solver/sampler/HMC/helpers/hamiltonian_riemannian_diag.hpp”*#include* “modules/solver/sampler/HMC/helpers/leapfrog_explicit.hpp”*#include* “modules/solver/sampler/HMC/helpers/leapfrog_implicit.hpp”*#include* “modules/solver/sampler/HMC/helpers/tree_helper_euclidean.
“modules/solver/sampler/HMC/helpers/tree_helper_riemannian.hpp” Header file for module: HMC.

file **MCMC.cpp**
#include “engine.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/problem/problem.hpp”*#include* “modules/solver/sampler/MCMC/MCMC.hpp”*#include* “sample/sample.hpp”*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>

file **MCMC.hpp**
#include “modules/distribution/univariate/normal/normal.hpp”*#include* “modules/distribution/univariate/uniform/uniform.hpp”*#include* “modules/solver/sampler/sampler.hpp”*#include* <> Header file for module: MCMC.

file **Nested.cpp**
#include “engine.hpp”*#include* “modules/distribution/univariate/uniform/uniform.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/solver/sampler/Nested/Nested.hpp”*#include* “sample/sample.hpp”*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>

file **Nested.hpp**
#include "modules/distribution/multivariate/normal/normal.hpp"*#include* "modules/distribution/univariate/normal/normal.hpp"*#include* "modules/distribution/univariate/uniform/uniform.hpp"*#include* "modules/solver/sampler/sampler.hpp"*#include* <> Header file for module: Nested.

file **sampler.cpp**
#include "modules/solver/sampler/sampler.hpp"

file **sampler.hpp**
#include "modules/solver/solver.hpp" Header file for module: Sampler.

file **TMCMC.cpp**
#include "engine.hpp"*#include* "modules/experiment/experiment.hpp"*#include* "modules/solver/sampler/TMCMC/TMCMC.hpp"*#include* "sample/sample.hpp"*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>*#include* <>

file **TMCMC.hpp**
#include "modules/distribution/distribution.hpp"*#include* "modules/distribution/multivariate/normal/normal.hpp"*#include* "modules/distribution/specific/multinomial/multinomial.hpp"*#include* "modules/distribution/univariate/uniform/uniform.hpp"*#include* "modules/solver/sampler/sampler.hpp"*#include* <> Header file for module: TMCMC.

file **solver.cpp**
#include "modules/solver/solver.hpp"

file **solver.hpp**
#include ""*#include* "modules/experiment/experiment.hpp"*#include* "modules/module.hpp"*#include* "sample/sample.hpp"*#include* <>*#include* <> Header file for module: Solver.

Defines

KORALI_START (*SAMPLE*)
Macro to start the processing of a sample.

KORALI_WAIT (*SAMPLE*)
Macro to wait for the finishing of a sample.

KORALI_WAITANY (*SAMPLES*)
Macro to wait for any of the given samples.

KORALI_WAITALL (*SAMPLES*)
Macro to wait for all of the given samples.

KORALI_SEND_MSG_TO_SAMPLE (*SAMPLE*, *MSG*)
Macro to send a message to a sample.

KORALI_RECV_MSG_FROM_SAMPLE (*SAMPLE*)
Macro to receive a message from a sample (blocking)

KORALI_LISTEN (*SAMPLES*)
(Blocking) Receives all pending incoming messages (at least one) and stores them into the corresponding sample's message queue.

file **SSA.cpp**
#include "modules/solver/SSM/SSA/SSA.hpp"

file **SSA.hpp**
#include "modules/solver/SSM/SSM.hpp" Header file for module: SSA.

file **SSM.cpp**
#include “modules/solver/SSM/SSM.hpp”

file **SSM.hpp**
#include “modules/distribution/univariate/uniform/uniform.hpp”*#include* “modules/problem/reaction/reaction.hpp”*#include* “modules/solver/solver.hpp” Header file for module: SSM.

file **TauLeaping.cpp**
#include “modules/solver/SSM/TauLeaping/TauLeaping.hpp”

file **TauLeaping.hpp**
#include “modules/distribution/univariate/poisson/poisson.hpp”*#include* “modules/solver/SSM/SSM.hpp” Header file for module: TauLeaping.

file **sample.cpp**
#include “”*#include* “sample/sample.hpp”*#include* “”*#include* “engine.hpp”*#include* “modules/conduit/conduit.hpp”*#include* “modules/experiment/experiment.hpp”*#include* “modules/problem/problem.hpp”*#include* “modules/solver/solver.hpp”*#include* <>

file **sample.hpp**
#include “auxiliar/koraliJson.hpp”*#include* “”*#include* “auxiliar/logger.hpp”*#include* “”*#include* <>*#include* <> Contains the definition of a *Korali* Sample.

Defines

KORALI_GET (*TYPE*, *SAMPLE*, ...)
Macro to get information from a sample. Checks for the existence of the path and produces detailed information on failure.

KORALI_SEND_MSG_TO_ENGINE (*MESSAGE*)
Macro to send message updates to the engine.

KORALI_RECV_MSG_FROM_ENGINE ()
Macro to recv message updates from the engine.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Activation.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: AdaBelief.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Adam.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Agent.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/auxilia**
Contains auxiliar libraries and tools to run *Korali*.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Bayesian.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Beta.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Cauchy.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: CMAES.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Concurrent.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Conduit.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Continuous.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Continuous.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Convolution.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Custom.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: DEA.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Deconvolution.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: DeepSupervisor.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Design.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Designer.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Discrete.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Discrete.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Distributed.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Distribution.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: dVRACER.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Executor.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Experiment.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Exponential.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: fAdaBelief.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: fAdaGrad.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: fAdam.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: fMadGrad.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Gamma.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Geometric.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: GridSearch.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: GRU.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Hierarchical.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: HMC.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Igamma.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Input.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Integration.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Integrator.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Laplace.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Layer.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Linear.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: LogNormal.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: LSTM.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: MADGRAD.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: MCMC.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: MOCMAES.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains all the modules upon which a *Korali* application is created.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: MonteCarlo.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Multinomial.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Multivariate.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Nested.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: NeuralNetwork.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Normal.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Normal.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Optimization.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Optimizer.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: fGradientBasedOptimizer.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Output.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Poisson.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Pooling.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Problem.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Propagation.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Psi.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Quadrature.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Reaction.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Recurrent.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Reference.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: ReinforcementLearning.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Rprop.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/sample**
Contains the definition of a *Korali* Sample.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Sampler.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Sampling.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Sequential.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Solver.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source**
Contains source code for the *Korali* engine, experiment, and its modules.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Specific.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: SSA.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: SSM.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: SupervisedLearning.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: TauLeaping.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Theta.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: ThetaNew.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: TMCMC.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: TruncatedNormal.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Uniform.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: UniformRatio.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Univariate.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: VRACER.

dir **/home/docs/checkouts/readthedocs.org/user_builds/korali/checkouts/cleanup/source/modules**
Contains code, documentation, and scripts for module: Weibull.

- genindex

- modindex
- search

INDEX

Symbols

A

B

C

D

E

H

I

K

L

M

N

O

P

R

S

T

U

W