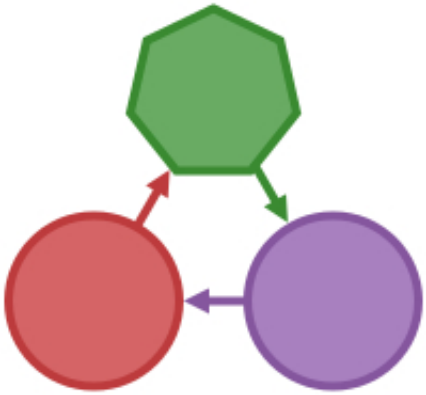

Klara User Guide

Release 0.5.3

JuliaStats Team

Oct 09, 2017

Contents

CHAPTER 1

Introduction

Main principles of development

The Julia `Klara` package provides an interoperable generic engine for a breadth of Markov Chain Monte Carlo (MCMC) methods.

The idea that there exists no unique optimal MCMC methodology for all purposes has been a development cornerstone. Along these lines, interest is in providing a wealth of Monte Carlo strategies and options, letting the user decide which algorithm suits their use case. Such "agnostic" approach to coding permeates `Klara` from top to bottom, offering a variety of methods and detailed configuration, ultimately leading to rich functionality. `Klara`'s wide range of functionality makes itself useful in applications and as a test bed for comparative methodological research. It also gives the flexibility to connect `Klara` with various other packages, exploiting different levels of ongoing developments in them.

In fact, interoperability has been another central principle of development. A high-level API enables the user to implement their application effectively, while it facilitates connectivity to Julia packages. Whenever deemed necessary, minor wrappers are provided to integrate `Klara` with other packages such as `ReverseDiffSource` and `ForwardDiff`.

The high-level API sits atop of a low-level one. The latter aims at providing an extensible codebase for developers interested in adding new functionality and offers an alternative interface for users who prefer more hands-on access to the underlying routines.

Speed of execution has been another motivation behind the low-level API. Passing from the higher to the lower level interface allows to exploit Julia's meta-programming capabilities by generating code dynamically and by substituting dictionaries by vectors internally. Memory footprint and garbage collection have been kept to a minimum without compromising ease of use thanks to the duality of higher and lower level APIs.

Features

A summary of `Klara`'s main features follows:

- *Graph-based model specification*. Representing the model as a graph widens the scope of accommodated models and enables exploiting graph algorithms from the `Graphs` package.
- *Diverse options for defining model parameters*. Parameters can be defined on the basis of a log-target or they can be introduced in a Bayesian fashion via their log-likelihood and log-prior. Parameter targets, likelihoods and priors can be specified via functions or distributions. `Klara`'s integration with the `Distributions` package facilitates parameter definition via distributions.
- *Job-centric simulations*. The concept of MCMC simulation has been separated from that of model specification. Job types indicate the context in which a model is simulated. For example, a `BasicMCJob` instance determines how to sample an MCMC chain for a model with a single parameter, whereas a `GibbsJob` provide Gibbs sampling for more complex models involving several parameters.
- *Customized job flow*. Job control flow comes in two flavors, as it can be set to ordinary loop-based flow or it can be managed by Julia's tasks (coroutines). Job management with tasks allows MCMC simulations to be suspended and resumed in a flexible manner.
- *Wide range of Monte Carlo samplers*. A range of MCMC samplers is available, including accept-reject and slice sampling, Metropolis-Hastings algorithm, No-U-Turn (NUTS) sampling, and geometric MCMC schemes, such as Riemann manifold Langevin and Hamiltonian Monte Carlo. Adaptive samplers and empirical tuning are included in `Klara` as a means to faster convergence. It is noted that most of these samplers need to be ported from the older version of Klara, which is work in progress.
- *MCMC summary statistics and convergence diagnostics*. Main routines for computing the effective sampling size and integrated autocorrelation time have been coded, while there is a roadmap to provide more convergence diagnostics tools (note to user; this functionality will also be ported soon from the older version of `Klara`).
- *States and chains*. Proposed Monte Carlo samples are organized systematically with the help of a state and chain type system. This way, values can be passed around and stored without re-allocating memory. At the same time, the state/chain type system offers scope for extending the current functionality if it is required to store less usual components.
- *Detailed configuration of output storage in memory or in file*. The chain resulting from a Monte Carlo simulation can be saved in memory or can be written directly to a file stream. Detailed output configuration is possible, allowing to select which elements to save and which to omit from the final output.
- *Automatic differentiation for MCMC sampling*. Some Monte Carlo methods require the gradient or higher order derivatives of the log-target. If these derivatives are not user-inputted explicitly, `Klara` can optionally compute them using reverse or forward mode automatic differentiation. For this purpose, `Klara` uses `ReverseDiffSource` and `ForwardDiff` under the hood.

Preliminary exposition of graph models

`Klara`'s graph model is presented concisely in the current section as a smooth introduction to subsequent elaborate chapters. A single model type, named `GenericModel`, serves as the sole entry point for defining any graph model in `Klara`. `GenericModel` has been inspired by and operates on par with `GenericGraph`, a versatile graph type of the `Graphs` package.

GenericModel can be conceptualized as a graph whose nodes represent the underlying model's variables and its edges specify the dependencies between these variables. As it becomes obvious, the main front end of `GenericModel` consists of its `vertices` and `edges` fields, which are of type `Vector{Variable}` and `Vector{Dependence}` respectively. Without going into details, each vertex is defined as constant, data, transformation or parameter, all being `Variable` subtypes. A single non-abstract `Dependence` type suffices to describe variable dependencies.

Typical probabilistic graphical models, such as directed acyclic graphs (DAGs) and factor graphs, are permitted in `GenericModel`. Similarly to `GenericGraph`, the `is_directed` field of `GenericModel` dictates whether

the model is directed or not. Practically, Gibbs sampling is not affected by the distinction between directed and non-directed graphs. However, effort has been made to define `GenericModel` generically in order to provide scope for future developments if the need arises to distinguish between DAGs and factor graphs in programming practice.

The graph-oriented definition of `GenericModel` finds its main utility in `Klara`'s optional declarative model specification. In other words, it is possible to delegate responsibility of variable ordering to `GenericModel` via topological sorting of the graph. Furthermore, the statistical model is easier to disseminate by visualizing `GenericModel` as a graph.

Topological sorting and graph visualization are achieved via "outsourcing". In particular, converting `GenericModel` to its corresponding `GenericGraph` allows to harness sorting routines in the `Graphs` package. Moreover, `GenericModel` is convertible to DOT format, thus making it possible to use the DOT graph description language for model visualization.

CHAPTER 2

Variable States

Rationale behind states

Being geared towards statistics, `Klara` variables include parameters, data, transformations and constants, the specifics of which will be delineated in *Variables*. A variable, be it stochastic or deterministic, can take a value, that is it can have a *state*. Variables and their states are maintained in two distinct type systems in `Klara`.

The functionality of a variable is enclosed by its type instance. For a example, a typical parameter consists of its probability distribution, log-likelihood or log-prior fields.

It is possible to store the value of a multivariate variable in a vector. However, it might be required to save additional information. For example, the value of a parameter's log-likelihood and associated gradient might be of interest. Variable state types exist to accommodate such states that comprise two or more entities.

From an object-oriented programming (OOP) standpoint, variable types correspond to methods while variable state types constitute data members. `Klara` does not merge the functional and data components into a single type, which would had been the analogous of a class in OOP terms. The main reasoning behind Klara's compartmentalization of variables and their states is code reusability. For instance, it becomes possible for different variables to share the same state type; more generally, adhering to Julia's multiple dispatch is facilitated.

Moreover, keeping states separate from variables helps cater to user-specific problems. Existing functionality can be deployed on user-defined states tailored to the problem at hand.

Built-in states

The state type system comprises two abstract and other non-abstract types. Listing 2.1 displays the hierarchy of built-in state types. The two abstract types are `VariableState` and its sub-type `ParameterState`. The non-abstract states put forward sensible defaults aimed at covering conventional use-cases. `Klara`'s existing functionality relies on these defaults, yet it is possible to extend the package by defining custom state types.

Variable states are categorized as univariate or multivariate. Parameter states are further classified as discrete or continuous. To make these distinctions, parametric abstract state types are employed by importing the `VariateForm` and `ValueSupport` classification scheme from `Distributions`. Every possible category is designated a unique

non-abstract state type; for instance, `BasicContMuvParameterState` hosts a continuous multivariate parameter state.

The most common field, appearing in all built-in non-abstract state types, is called `value`. For example, in the context of MCMC, `value` would hold the current or proposed state at each iteration of the sampler. Each state type and its associated methods will be elaborated in the following sections.

Listing 2.1: State type hierarchy in `Klara`

```
VariableState
|
+-- BasicUnvVariableState
|
+-- BasicMuvVariableState
|
+-- BasicMavVariableState
|
+-- ParameterState
    |
    +-- BasicDiscUnvParameterState
    |
    +-- BasicDiscMuvParameterState
    |
    +-- BasicContUnvParameterState
    |
    +-- BasicContMuvParameterState
```

Abstract states

The type system of `VariateForm` and `ValueSupport` from `Distributions` (see Listing 2.2 and associated documentation in `Distributions`) is used for parameterizing abstract states in `Klara`.

Listing 2.2: `VariateForm` and `ValueSupport` type system from `Distributions` package

```
abstract VariateForm
type Univariate    <: VariateForm end
type Multivariate  <: VariateForm end
type Matrixvariate <: VariateForm end

abstract ValueSupport
type Discrete   <: ValueSupport end
type Continuous <: ValueSupport end
```

`VariableState` is the root of `Klara`'s variable state type hierarchy. It is defined as

```
abstract VariableState{F<:VariateForm}
```

Being parameterized by `VariateForm`, the abstract type `VariableState` enables distinguishing between univariate, multivariate and matrix-variate variable states.

`ParameterState` is the root of `Klara`'s parameter state types and an abstract sub-type of `VariableState`. It is defined as

```
abstract ParameterState{S<:ValueSupport, F<:VariateForm} <: VariableState{F}
```

As seen from its parameterization, `ParameterState` makes it possible to organize parameter states by both the support of state space and the variate form.

Basic variable states

`Klara` ships with three so-called basic variable state types, namely `BasicUnvVariableState`, `BasicMuvVariableState` and `BasicMavVariableState`. These three types are used for encapsulating minimal information, that is the `value` of a variable state and possibly the associated `size` of `value`.

Each of these three basic state types corresponds to a specific variate form, whereas none of them is parameterized by the support of state space. Instead, each of them is parameterized by the type of `Number` of their `value` field, see Table 2.1.

Table 2.1: Basic variable states in `Klara`.

Variable state type	`value` type	`size` type
BasicUnvVariableState{N<:Number}	N	✗
BasicMuvVariableState{N<:Number}	Vector{N}	Int
BasicMavVariableState{N<:Number}	Matrix{N}	Tuple{Int, Int}

In what follows, constructors are provided for the three basic variable types.

BasicUnvVariableState

`BasicUnvVariableState{N<:Number}`(*value::N*)
Construct a basic univariate variable state with some `value`.

Examples:

```
state = BasicUnvVariableState(1.)
# Klara.BasicUnvVariableState{Float64}(1.0)

state.value
# 1.0
```

BasicMuvVariableState

`BasicMuvVariableState{N<:Number}`(*value::Vector{N}*)
Construct a basic multivariate variable state with some `value`.

Examples:

```
state = BasicMuvVariableState([1, 2])
# Klara.BasicMuvVariableState{Int64}([1, 2], 2)

state.value
#  2-element Array{Int64, 1}:
#   1
#   2

state.size
# 2
```

`BasicMuvVariableState{N<:Number}`(*size::Int*, *::Type{N}=Float64*)
Construct a basic multivariate variable state with a `value` of specified `size` and element type.

Examples:

```
BasicMuvVariableState(3, Float32)
# Klara.BasicMuvVariableState{Float32}(3-element Array{Float32, 1}, 2)
```

BasicMavVariableState

`BasicMavVariableState{N<:Number}` (*value::Matrix{N}*)
Construct a basic matrix-variate variable state with some `value`.

Examples:

```
state = BasicMavVariableState(eye(2))
# Klara.BasicMavVariableState{Float64}(2x2 Array{Float64, 2}, (2, 2))

state.value
#  2x2 Array{Float64, 2}:
#   1.0  0.0
#   0.0  1.0

state.size
# (2, 2)
```

`BasicMavVariableState{N<:Number}` (*size::Tuple*, *::Type{N}=Float64*)
Construct a basic matrix-variate variable state with a `value` of specified `size` and element type.

Examples:

```
BasicMavVariableState((3, 2), Float32)
# Klara.BasicMavVariableState{Int16}(3x2 Array{Float32, 2}, (3, 2))
```

Basic parameter states

Four basic parameter state types are made available by `Klara`, namely the discrete univariate `BasicDiscUnvParameterState`, discrete multivariate `BasicDiscMuvParameterState`, continuous univariate `BasicContUnvParameterState` and continuous multivariate `BasicContMuvParameterState`, see Table 2.2.

Table 2.2: Basic parameter states in `Klara`.

Parameter state type	ValueSupport	VariateForm
BasicDiscUnvParameterState{NI<:Integer, NR<:Real}	Discrete	Univariate
BasicDiscMuvParameterState{NI<:Integer, NR<:Real}	Discrete	Multivariate
BasicContUnvParameterState{NR<:Real}	Continuous	Univariate
BasicContMuvParameterState{NR<:Real}	Continuous	Multivariate

Both basic parameter states and basic variable states contain the state's `value` and value's `size`. Additionally, basic parameter states contain fields that hold information about the target distribution of the associated parameter and about sampling diagnostics, see Table 2.3.

The discrete states `BasicDiscUnvParameterState` and `BasicDiscMuvParameterState` are parameterized by the element type `NI<:Integer` of state `value` and by the element type `NR<:Real` of target-related fields. On the other hand, the continuous states `BasicContUnvParameterState` and `BasicContMuvParameterState` are parameterized by the common element type `NR<:Real` of state `value` and of target-related fields.

A parameter is characterized by its target, that is by its possibly unnormalized distribution. A target is specified via a `Distribution` or via a possibly unnormalized probability distribution `function` (PDF). Either way, the `state.logtarget` field of a parameter `state` stores the logarithm of the associated PDF evaluated at `state.value`.

A posterior target is proportional to a likelihood times a prior. Thus, if a parameter is specified via its posterior target, the `state.loglikelihood` and `state.logprior` fields of the associated parameter `state` enable storing the logarithm of the likelihood function and prior PDF evaluated at `state.value`. Apparently, `state.logtarget` is equal to the sum of `state.loglikelihood` and `state.logprior`.

Table 2.3: Fields of basic parameter state types in `Klara`. All four types are parameterized by `NI<:Integer` or `NR<:Real`.

	Basic[S/F]ParameterState{P}			
S<:ValueSupport	Discrete	Discrete	Continuous	Continuous
F<:VariateForm	Univariate	Multivariate	Univariate	Multivariate
P (Parameters)	NI, NR	NI, NR	NR	NR
Field	Field type			
value	NI	Vector{NI}	NR	Vector{NR}
loglikelihood	NR	NR	NR	NR
logprior	NR	NR	NR	NR
logtarget	NR	NR	NR	NR
gradloglikelihood	✗	✗	NR	Vector{NR}
gradlogprior	✗	✗	NR	Vector{NR}
gradlogtarget	✗	✗	NR	Vector{NR}
tensorloglikelihood	✗	✗	NR	Matrix{NR}
tensorlogprior	✗	✗	NR	Matrix{NR}
tensorlogtarget	✗	✗	NR	Matrix{NR}
dtensorloglikelihood	✗	✗	NR	Array{NR, 3}
dtensorlogprior	✗	✗	NR	Array{NR, 3}
dtensorlogtarget	✗	✗	NR	Array{NR, 3}
diagnosticvalues	Vector	Vector	Vector	Vector
size	✗	Int	✗	Int
diagnostickeys	Vector{Symbol}	Vector{Symbol}	Vector{Symbol}	Vector{Symbol}

The rest of target-related fields, prefixed by *grad*, *tensor* and *dtensor*, appear only in continuous parameter states and correspond to first, second and third degree derivatives of the target. Such target derivatives are utilized by various MCMC algorithms.

Fields starting with *grad* store the gradient of the prefixed function. For example, `state.gradlogtarget` stores the gradient of the log-target evaluated at `state.value`.

Fields starting with *tensor* refer to the metric tensor of the prefixed function. It is noted that the concept of metric tensor is used in an information theoretic context referring to distance between distributions *[1]*. For instance, `state.tensorloglikelihood` can be used for saving the expected Fisher information matrix, which is equal to the negative expected value of the second-order derivative of the log-likelihood evaluated at `state.value`. Moreover, `state.tensorlogprior` can be utilized for storing the negative Hessian of the log-prior evaluated at `state.value`. As for `state.dtensorlogtarget`, it is the metric tensor of the log-target, which equals the sum of `state.tensorloglikelihood` and `state.tensorlogprior`.

Fields prefixed by *dtensor* store all first-order derivatives of the metric tensor referred by the respective *tensor*-prefixed field, thus yielding third-order derivatives of the target. For example, `state.dtensorlogtarget` saves all first-order derivatives of `state.tensorlogtarget` evaluated at `state.value`.

`state.diagnosticvalues` is a `Vector` used for storing diagnostics pertaining to the sampling of a parameter `state`. The `state.diagnosticvalues` are labeled by an accordingly ordered `Vector{Symbol}` of `state.diagnostickeys`. Conceptually, `state.diagnostickeys` and `state.diagnosticvalues` can be seen

as the keys and values of a dictionary of diagnostics but are maintained in two separate vectors to improve MCMC performance. The two vectors are interfaced with a `diagnostics()` function, which zips them together and returns the resulting dictionary.

`diagnostics`(*state::ParameterState*)
Return the dictionary of `state` diagnostics arising from `state.diagnostickeys` and `state.diagnosticvalues`.

The constructors of `Klara`'s basic parameter state types are elucidated in the remaining of this section.

BasicDiscUnvParameterState

`BasicDiscUnvParameterState{NI, NR}(value::NI, <optional arguments>)`
Construct a basic discrete univariate parameter state with some `value`.

The parameterization is set as `NI<:Integer`, `NR<:Real`.

Optional arguments:

- `diagnostickeys::Vector{Symbol}=Symbol[]`: the diagnostic keys of the state.
- `::Type{NR}=Float64`: the element type of target-related fields.
- `diagnosticvalues::Vector=Array(Any, length(diagnostickeys))`: the diagnostic values of the state.

Examples:

```
state = BasicDiscUnvParameterState(2, [:accept], Float64, [true])
# Klara.BasicDiscUnvParameterState{Int64, Float64}(
#   2, NaN, NaN, NaN, Bool[true], [:accept]
# )

state.value
# 2

diagnostics(state)
# Dict{Symbol, Bool} with 1 entry:
#   :accept => true
```

BasicDiscMuvParameterState

`BasicDiscMuvParameterState{NI, NR}(value::Vector{NI}, <optional arguments>)`
Construct a basic discrete multivariate parameter state with some `value`.

The parameterization is set as `NI<:Integer`, `NR<:Real`.

Optional arguments:

- `diagnostickeys::Vector{Symbol}=Symbol[]`: the diagnostic keys of the state.
- `::Type{NR}=Float64`: the element type of target-related fields.
- `diagnosticvalues::Vector=Array(Any, length(diagnostickeys))`: the diagnostic values of the state.

Examples:

```
state = BasicDiscMuvParameterState(Int64[0, 1], [:accept], Float64, [false])
# Klara.BasicDiscMuvParameterState{Int64, Float64}(
#   [0, 1], NaN, NaN, NaN, Bool[false], 2, [:accept]
# )

state.value
# 2-element Array{Int64, 1}:
#  0
#  1

diagnostics(state)
# Dict{Symbol, Bool} with 1 entry:
#  :accept => false
```

BasicDiscMuvParameterState{NI, NR}(size::Int, <optional arguments>)
Construct a basic discrete multivariate parameter state with with a `value` of specified `size`.

The parameterization is set as `NI<:Integer`, `NR<:Real`.

Optional arguments:

- `diagnostickeys::Vector{Symbol}=Symbol[]`: the diagnostic keys of the state.
- `::Type{NI}=Int`: the element type of the state value.
- `::Type{NR}=Float64`: the element type of target-related fields.
- `diagnosticvalues::Vector=Array(Any, length(diagnostickeys))`: the diagnostic values of the state.

Examples:

```
BasicDiscMuvParameterState(3, [:accept], Int32, Float32, [true])
# Klara.BasicDiscMuvParameterState{Int32, Float32}(
#   3-element Array{Int32, 1}, NaN32, NaN32, NaN32, Bool[true], 3, [:accept]
# )
```

BasicContUnvParameterState

BasicContUnvParameterState{N<:Real} (*value::N*, <*optional arguments*>)
Construct a basic continuous univariate parameter state with some `value`.

Optional arguments:

- `diagnostickeys::Vector{Symbol}=Symbol[]`: the diagnostic keys of the state.
- `diagnosticvalues::Vector=Array(Any, length(diagnostickeys))`: the diagnostic values of the state.

Examples:

```
state = BasicContUnvParameterState(-1.25, [:accept], [false])
# Klara.BasicContUnvParameterState{Float64}(
#   -1.25,
#   NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN,
#   Bool[false], [:accept]
# )

state.value
# -1.25
```

```
diagnostics(state)
# Dict{Symbol, Bool} with 1 entry:
#   :accept => false
```

BasicContUnvParameterState{N<:Real} (*<optional arguments>*)
Construct a basic continuous univariate parameter state with an uninitialized `value` (`NaN`).

Optional arguments:

- `diagnostickeys::Vector{Symbol}=Symbol[]`: the diagnostic keys of the state.
- ::Type{N}=Float64:: the element type of the state value.
- `diagnosticvalues::Vector=Array(Any, length(diagnostickeys))`: the diagnostic values of the state.

Examples:

```
BasicContUnvParameterState()
# Klara.BasicContUnvParameterState{Float64}(
#   NaN,
#   NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN, NaN,
#   Any[], Symbol[]
# )
```

BasicContMuvParameterState

BasicContMuvParameterState{N<:Real} (*value::Vector{N}*, *<optional arguments>*)
Construct a basic continuous multivariate parameter state with some `value`.

Optional arguments:

- `monitor::Vector{Bool}=fill(false, 9)`: 9-element Boolean vector indicating which of the target-related fields are stored by the state.
- `diagnostickeys::Vector{Symbol}=Symbol[]`: the diagnostic keys of the state.
- `diagnosticvalues::Vector=Array(Any, length(diagnostickeys))`: the diagnostic values of the state.

Examples:

```
state = BasicContMuvParameterState(ones(Float32, 2))
# Klara.BasicContMuvParameterState{Float32}(
#   Float32[1.0f0, 1.0f0],
#   NaN32, NaN32, NaN32,
#   Float32[], Float32[], Float32[],
#   0x0 Array{Float32, 2}, 0x0 Array{Float32, 2}, 0x0 Array{Float32, 2},
#   0x0x0 Array{Float32, 3}, 0x0x0 Array{Float32, 3}, 0x0x0 Array{Float32, 3},
#   Any[], 2, Symbol[]
# )

state.value
# 2-element Array{Float32,1}:
#  1.0
#  1.0

size(state.gradloglikelihood)
# (0,)
```

```
size(state.gradlogtarget)
# (0,)

diagnostics(state)
# Dict{Symbol,Any} with 0 entries
```

BasicContMuvParameterState{N<:Real}(*value::Vector{N}*, *monitor::Vector{Symbol}*, *<optional arguments>*)

Construct a basic continuous multivariate parameter state with some `value` and tracked target-related fields specified by `monitor`.

Optional arguments:

- `diagnostickeys::Vector{Symbol}=Symbol[]`: the diagnostic keys of the state.
- `diagnosticvalues::Vector=Array(Any, length(diagnostickeys))`: the diagnostic values of the state.

Examples:

```
state = BasicContMuvParameterState(
  zeros(Float64, 2), [:logtarget, :gradlogtarget]
)
# Klara.BasicContMuvParameterState{Float64}(
#   [0.0, 0.0],
#   NaN, NaN, NaN,
#   Float64[], Float64[], 2-element Array{Float64, 1},
#   0x0 Array{Float64, 2}, 0x0 Array{Float64, 2}, 0x0 Array{Float64, 2},
#   0x0x0 Array{Float64, 3}, 0x0x0 Array{Float64, 3}, 0x0x0 Array{Float64, 3},
#   Any[], 2, Symbol[]
# )

size(state.gradloglikelihood)
# (0,)

size(state.gradlogtarget)
# (2,)
```

BasicContMuvParameterState{N<:Real}(*size::Int*, *<optional arguments>*)

Construct a basic continuous multivariate parameter state with a `value` of specified `size`.

Optional arguments:

- `monitor::Vector{Bool}=fill(false, 9)`: 9-element Boolean vector indicating which of the target-related fields are stored by the state.
- `diagnostickeys::Vector{Symbol}=Symbol[]`: the diagnostic keys of the state.
- `::Type{N}=Float64`: the element type of the state value.
- `diagnosticvalues::Vector=Array(Any, length(diagnostickeys))`: the diagnostic values of the state.

Examples:

```
state = BasicContMuvParameterState(3)
# Klara.BasicContMuvParameterState{Float64}(
#   3-element Array{Float64, 1},
#   NaN, NaN, NaN,
#   Float64[], Float64[], Float64[],
```

```
#   0x0 Array{Float64, 2}, 0x0 Array{Float64, 2}, 0x0 Array{Float64, 2},
#   0x0x0 Array{Float64, 3}, 0x0x0 Array{Float64, 3}, 0x0x0 Array{Float64, 3},
#   Any[], 3, Symbol[]
# )

state.size
# 3

diagnostics(state)
# Dict{Symbol,Any} with 0 entries
```

BasicContMuvParameterState{N<:Real}(*size::Int*, *monitor::Vector{Symbol}*, *<optional arguments>*)

Construct a basic continuous multivariate parameter state with a `value` of specified `size` and tracked target-related fields specified by `monitor`.

Optional arguments:

- `diagnostickeys::Vector{Symbol}=Symbol[]`: the diagnostic keys of the state.
- `::Type{N}=Float64`: the element type of the state value.
- `diagnosticvalues::Vector=Array(Any, length(diagnostickeys))`: the diagnostic values of the state.

Examples:

```
BasicContMuvParameterState(
  3, [:loglikelihood, :logtarget], [:accept], Float16, [true]
)
# Klara.BasicContMuvParameterState{Float16}(
#   3-element Array{Float16, 1}
#   NaN16, NaN16, NaN16,
#   Float16[], Float16[], Float16[],
#   0x0 Array{Float16, 2}, 0x0 Array{Float16, 2}, 0x0 Array{Float16, 2},
#   0x0x0 Array{Float16, 3}, 0x0x0 Array{Float16, 3}, 0x0x0 Array{Float16, 3},
#   Bool[true], 3, [:accept]
# )
```

CHAPTER 3

Variable NStates (Chains)

`Klara`'s `NState` type system is used for storing several variable states. An instance `nstate` of some `NState` type stores `nstate.n` number of states, each of which share the same type. `NStates` are useful for saving the output of Monte Carlo simulations. A Markov chain generated by a Monte Carlo algorithm consists of a certain number of states. An instance of some `NState` type can be used for saving the simulated Markov chain. For this reason, `Klara` provides `Chain` aliases for `NState` types.

CHAPTER 4

Variable IOStreams

To be written up soon.

CHAPTER 5

Variables

To be written up soon.

CHAPTER 6

Models

To be written up soon.

CHAPTER 7

Samplers

To be written up soon.

CHAPTER 8

Tuners

To be written up soon.

CHAPTER 9

Ranges

To be written up soon.

CHAPTER 10

Jobs

To be written up soon.

CHAPTER 11

MCMC Stats

To be written up soon.

CHAPTER 12

Automatic Differentiation

To be written up soon.

CHAPTER 13

Examples

To be written up soon.

Bibliography

[1] Mark Girolami and Ben Calderhead. Riemann manifold langevin and hamiltonian monte carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 73(2):123–214, 2011.

Index