dakotathon Documentation

Release 0.5

Mark Piper

Jul 11, 2019

Contents

Release 0.5

Date Jul 11, 2019

Dakotathon provides a Python API and BMI for the Dakota iterative systems analysis toolkit. The interface follows the documentation for the keywords used to configure a Dakota experiment.

CHAPTER 1

Contents

1.1 Dakota class

A Python interface to the Dakota iterative systems analysis toolkit.

class dakotathon.dakota.**Dakota**(*run_directory='/home/docs/checkouts/readthedocs.org/user_builds/csdms-dakota/checkouts/stable/docs/source'*, *configuration_file='dakota.yaml'*, *input_file='dakota.in'*, *output_file='dakota.out'*, *run_log='run.log'*, *error_log='stderr.log'*, *template_file=None*, *auxiliary_files=()*, ***kwargs*)

Bases: *dakotathon.experiment.Experiment*

Controller for configuring and running a Dakota experiment.

__init__(*run_directory='/home/docs/checkouts/readthedocs.org/user_builds/csdms-dakota/checkouts/stable/docs/source'*, *configuration_file='dakota.yaml'*, *input_file='dakota.in'*, *output_file='dakota.out'*, *run_log='run.log'*, *error_log='stderr.log'*, *template_file=None*, *auxiliary_files=()*, ***kwargs*)

Initialize a Dakota experiment.

Called with no parameters, a Dakota experiment with basic defaults (a vector parameter study with the built-in *rosenbrock* example) is created. Use `method` to set the Dakota analysis method in a new experiment.

run_directory [str, optional] The working directory in which Dakota is run, and output is placed (default is the current directory).

configuration_file [str, optional] A Dakota instance serialized to a YAML file (default is **dakota.yaml**).

input_file [str, optional] Name of Dakota input file (default is **dakota.in**).

output_file [str, optional] Name of Dakota output file (default is **dakota.out**).

run_log [str, optional] Name of Dakota log file (default is **run.log***)

error_log [str, optional] Name of Dakota error log file (default is **stderr.log***)

template_file [str, optional] The Dakota template file, formed from the input file of the model to study, but with study variables replaced by descriptors in braces; e.g., `{total_annual_precipitation}` (default is None).

auxiliary_files [str or tuple or list of str, optional] Additional input files used by the model being studied.

****kwargs** Arbitrary keyword arguments.

Create a generic Dakota experiment:

```
>>> d = Dakota()
```

Create a vector parameter study experiment:

```
>>> d = Dakota(method='vector_parameter_study')
```

auxiliary_files
Auxiliary files used by the component.

configuration_file
The configuration file path.

classmethod from_file_like(*file_like*)
Create a Dakota instance from a file-like object.

file_like [file_like] A configuration file or file-like object.

Dakota A new Dakota instance.

run()
Run the Dakota experiment.

Run is executed in the directory specified by run_directory keyword and run log and error log are created.

run_directory
The run directory path.

serialize(*config_file=None*)
Dump settings for experiment to a YAML configuration file.

config_file [str, optional] A path/name for a new configuration file.

Make a configuration file for a vector parameter study experiment:

```
>>> d = Dakota(method='vector_parameter_study')
>>> d.serialize('dakota.yaml')
```

setup()
Write the Dakota configuration and input files.

As a convenience, make a configuration file and an input file for an experiment in one step:

```
>>> d = Dakota(method='vector_parameter_study')
>>> d.setup()
```

template_file
The template file path.

write_input_file(*input_file=None*)
Create the Dakota input file for the experiment.

The input file is written to the directory specified by the *run_directory* attribute.

input_file [str, optional] A path/name for a new Dakota input file.

Make an input file for a vector parameter study experiment:

```
>>> d = Dakota(method='vector_parameter_study')
>>> d.write_input_file('dakota.in')
```

1.2 Experiment class

A Python interface to a Dakota input file.

class dakotathon.experiment.**Experiment**(*component=None*, *plugin=None*, *environment='environment'*, *method='vector_parameter_study'*, *variables='continuous_design'*, *interface='direct'*, *responses='response_functions'*, ***kwargs*)

Bases: object

An aggregate of control blocks that define a Dakota input file.

__init__(*component=None*, *plugin=None*, *environment='environment'*, *method='vector_parameter_study'*, *variables='continuous_design'*, *interface='direct'*, *responses='response_functions'*, ***kwargs*)

Create the set of control blocks for a Dakota experiment.

Called with no parameters, a Dakota experiment with basic defaults (a vector parameter study with the built-in *rosenbrock* example) is created.

component [str, optional] Name of CSDMS component which Dakota is analyzing (default is None). The *component* and *plugin* parameters are exclusive.

plugin [str, optional] Name of a plugin model which Dakota is analyzing (default is None). The *component* and *plugin* parameters are exclusive.

environment [str, optional] Type of environment used in Dakota experiment (default is 'environment').

method [str, optional] Type of method used in Dakota experiment (default is 'vector_parameter_study').

variables [str, optional] Type of variables used in Dakota experiment (default is 'continuous_design').

interface [str, optional] Type of interface used in Dakota experiment (default is 'direct').

responses [str, optional] Type of responses used in Dakota experiment (default is 'response_functions').

****kwargs** Arbitrary keyword arguments.

Create a generic Dakota experiment:

```
>>> x = Experiment()
```

Create a vector parameter study experiment:

```
>>> x = Experiment(method='vector_parameter_study')
```

__str__()

The contents of the Dakota input file represented as a string.

Print the Dakota input file to the console.

```
>>> x = Experiment()
>>> print(x)
# Dakota input file
environment
  tabular_data
    tabular_data_file = 'dakota.dat'
<BLANKLINE>
method
  vector_parameter_study
    final_point = 1.1 1.3
    num_steps = 10
<BLANKLINE>
variables
  continuous_design = 2
    descriptors = 'x1' 'x2'
    initial_point = -0.3 0.2
<BLANKLINE>
interface
  id_interface = 'CSDMS'
  direct
  analysis_driver = 'rosenbrock'
<BLANKLINE>
responses
  response_functions = 1
    response_descriptors = 'y1'
  no_gradients
  no_hessians
<BLANKLINE>
```

blocks = ('environment', 'method', 'variables', 'interface', 'responses')
The named control blocks of a Dakota input file.

environment
The environment control block.

interface
The interface control block.

method
The method control block.

responses
The responses control block.

variables
The variables control block.

1.3 Dakota environments

Environments are top-level settings for Dakota execution.

1.3.1 Environment base class

An abstract base class for top-level Dakota settings.

class dakotathon.environment.base.**EnvironmentBase**(***kwargs*)

Bases: object

Describe features common to all Dakota environments.

__str__()

The header for the environment block of a Dakota input file.

1.3.2 Environment

A class for top-level Dakota settings.

class dakotathon.environment.environment.**Environment**(*data_file='dakota.dat'*, ***kwargs*)

Bases: *dakotathon.environment.base.EnvironmentBase*

Describe Dakota environment.

__init__(*data_file='dakota.dat'*, ***kwargs*)

Define parameters for the Dakota environment.

data_file [str, optional] The Dakota tabular data file (default is 'dakota.dat').

****kwargs** Optional keyword arguments.

__str__()

Define the environment block of a Dakota input file.

1.4 Dakota analysis methods

Python wrappers for Dakota analysis methods.

1.4.1 Methods base classes

Abstract base classes for Dakota analysis methods.

class dakotathon.method.base.**MethodBase**(*method='vector_parameter_study'*, *max_iterations=None*, *convergence_tolerance=None*, ***kwargs*)

Bases: object

Describe common features of Dakota analysis methods.

The *max_iterations* and *convergence_tolerance* keywords are included in Dakota's set of method independent controls.

__init__(*method='vector_parameter_study'*, *max_iterations=None*, *convergence_tolerance=None*, ***kwargs*)

Create default method parameters.

method [str] The name of the analysis method; e.g., 'vector_parameter_study'.

max_iterations [int, optional] Stopping criterion based on number of iterations.

convergence_tolerance [float, optional] Stopping criterion based on convergence of the objective function or statistics. Defined on the open interval (0, 1).

__str__()

Define the preamble of the Dakota input file method block.

convergence_tolerance
Convergence tolerance for the method.

max_iterations
Maximum number of iterations for the method.

method
The name of the analysis method used in the experiment.

class dakotathon.method.base.**UncertaintyQuantificationBase**(*basis_polynomial_family='extended'*, *probability_levels=(0.1, 0.5, 0.9)*, *response_levels=()*, *samples=10*, *sample_type='random'*, *seed=None*, *variance_based_decomp=False*, ***kwargs*)

Bases: *dakotathon.method.base.MethodBase*

Describe features of uncertainty quantification methods.

To supply *probability_levels* or *response_levels* to multiple responses, nest the inputs to these properties.

__init__(*basis_polynomial_family='extended'*, *probability_levels=(0.1, 0.5, 0.9)*, *response_levels=()*, *samples=10*, *sample_type='random'*, *seed=None*, *variance_based_decomp=False*, ***kwargs*)
Create default method parameters.

basis_polynomial_family: str, optional The type of polynomial basis used in the expansion, either 'extended' (the default), 'askey', or 'wiener'.

probability_levels [list or tuple of float, optional] Specify probability levels at which to estimate the corresponding response value. Default is (0.1, 0.5, 0.9).

response_levels [list or tuple of float, optional] Values at which to estimate desired statistics for each response

samples [int] The number of randomly chosen values at which to execute a model.

sample_type [str] Technique for choosing samples, *random* or *lhs*.

seed [int, optional] The seed for the random number generator. If seed is specified, a stochastic study will generate identical results when repeated using the same seed value. If not specified, a seed is randomly generated.

variance_based_decomp [bool, optional] Set to activate global sensitivity analysis based on decomposition of response variance into main, interaction, and total effects.

__str__()
Define the method block for a UQ experiment.

dakotathon.method.base.MethodBase.__str__

basis_polynomial_family
The type of basis polynomials used by the method.

probability_levels
Probabilities at which to estimate response values.

response_levels
Values at which to estimate statistics for responses.

sample_type
Sampling strategy.

samples
Number of samples in experiment.

seed
Seed of the random number generator.

variance_based_decomp
Use variance-based decomposition global sensitivity analysis.

1.4.2 Centered parameter study

Implementation of a Dakota centered parameter study.

class dakotathon.method.centered_parameter_study.**CenteredParameterStudy**(*steps_per_variable=(5, 4)*, *step_vector=(0.4, 0.5)*, ***kwargs*)

Bases: *dakotathon.method.base.MethodBase*

Define parameters for a Dakota centered parameter study.

__init__(*steps_per_variable=(5, 4)*, *step_vector=(0.4, 0.5)*, ***kwargs*)
Create a new Dakota centered parameter study.

steps_per_variable [array_like of int] Number of steps to take in each direction.

steps_vector [array_like of float] Size of steps in each direction.

Create a default centered parameter study experiment:

```
>>> c = CenteredParameterStudy()
```

__str__()
Define a centered parameter study method block.

dakotathon.method.base.MethodBase.__str__

step_vector
Step size in each direction.

steps_per_variable
Number of steps to take in each direction.

1.4.3 Multidim parameter study

Implementation of a Dakota multidim parameter study.

class dakotathon.method.multidim_parameter_study.**MultidimParameterStudy**(*partitions=(10, 8)*, ***kwargs*)

Bases: *dakotathon.method.base.MethodBase*

Define parameters for a Dakota multidim parameter study.

__init__(*partitions=(10, 8)*, ***kwargs*)
Create a new Dakota multidim parameter study.

partitions [array_like of int] Number of intervals between lower and upper bounds for each study parameter.

Create a default multidim parameter study experiment:

```
>>> m = MultidimParameterStudy()
```

__str__()
Define a multidim parameter study method block.

dakotathon.method.base.MethodBase.__str__

partitions
The number of evaluation intervals for each parameter.

1.4.4 Vector parameter study

Implementation of a Dakota vector parameter study.

class dakotathon.method.vector_parameter_study.**VectorParameterStudy**(*final_point=(1.1, 1.3)*, *n_steps=10*, ***kwargs*)

Bases: *dakotathon.method.base.MethodBase*

Define parameters for a Dakota vector parameter study.

__init__(*final_point=(1.1, 1.3)*, *n_steps=10*, ***kwargs*)
Create a new Dakota vector parameter study.

final_point [array_like of float] End point for the parameter study.

n_steps [int] Number of steps along vector.

Create a default vector parameter study experiment:

```
>>> v = VectorParameterStudy()
```

__str__()
Define a vector parameter study method block for a Dakota input file.

dakotathon.method.base.MethodBase.__str__

final_point
End points used by study variables.

n_steps
Number of steps along vector.

1.4.5 Sampling

Implementation of the Dakota sampling method.

class dakotathon.method.sampling.**Sampling**(***kwargs*)

Bases: *dakotathon.method.base.UncertaintyQuantificationBase*

The Dakota sampling method.

__init__(***kwargs*)

Create a new Dakota sampling study.

Create a default sampling experiment:

```
>>> x = Sampling()
```

__str__()

Define the method block for a sampling experiment.

dakotathon.method.base.UncertaintyQuantificationBase.__str__

1.4.6 Polynomial chaos

Implementation of the Dakota polynomial chaos method.

class dakotathon.method.polynomial_chaos.**PolynomialChaos**(*coefficient_estimation_approach='quadrature_or*
quadrature_order=2, *dimension_preference=()*, *nested=False*, ***kwargs*)

Bases: *dakotathon.method.base.UncertaintyQuantificationBase*

The Dakota polynomial chaos uncertainty quantification method.

Designation of a coefficient estimation approach is required, but the only approach currently implemented is *quadrature_order_sequence*, which obtains coefficients of the expansion using multidimensional integration by a tensor-product of Gaussian quadrature rules specified with *quadrature_order*, and, optionally, with *dimension_preference*. If *dimension_preference* is defined, its highest value is set to the *quadrature_order*.

This implementation of the polynomial chaos method is based on the description provided in the Dakota 6.4 documentation.

__init__(*coefficient_estimation_approach='quadrature_order_sequence'*, *quadrature_order=2*, *dimension_preference=()*, *nested=False*, ***kwargs*)

Create a new Dakota polynomial chaos study.

coefficient_estimation_approach [str] Technique to obtain coefficients of expansion.

quadrature_order [int] The highest order of the polynomial basis.

dimension_preference [list or tuple of int, optional] A set of weights specifying the relative importance of each uncertain variable (dimension).

nested [bool, optional] Set to enforce nested quadrature rules, if available (default is False).

Create a default instance of PolynomialChaos with:

```
>>> m = PolynomialChaos()
```

__str__()

Define the method block for a polynomial_chaos experiment.

Display the method block created by a default instance of PolynomialChaos:

```
>>> m = PolynomialChaos()
>>> print(m)
method
  polynomial_chaos
    sample_type = random
```

(continues on next page)

(continued from previous page)

```
    samples = 10
    probability_levels = 0.1 0.5 0.9
    quadrature_order = 2
    non_nested
<BLANKLINE>
<BLANKLINE>
```

dakotathon.method.base.UncertaintyQuantificationBase.__str__

dimension_preference
Weights specifying the relative importance of each dimension.

nested
Enforce use of nested quadrature rules.

quadrature_order
The highest order polynomial used by the method.

1.4.7 Stochastic collocation

Implementation of the Dakota stochastic collocation method.

class dakotathon.method.stoch_collocation.**StochasticCollocation**(*coefficient_estimation_approach='quad*, *quadrature_order=2*, *dimension_preference=()*, *nested=False*, ***kwargs*)

Bases: *dakotathon.method.base.UncertaintyQuantificationBase*

The Dakota stochastic collocation uncertainty quantification method.

Stochastic collocation is a general framework for approximate representation of random response functions in terms of finite-dimensional interpolation bases. Stochastic collocation is very similar to polynomial chaos, with the key difference that the orthogonal polynomial basis functions are replaced with interpolation polynomial bases.

This implementation of the stochastic collocation method is based on the description provided in the Dakota 6.4 documentation.

__init__(*coefficient_estimation_approach='quadrature_order_sequence'*, *quadrature_order=2*, *dimension_preference=()*, *nested=False*, ***kwargs*)

Create a new Dakota stochastic collocation study.

coefficient_estimation_approach [str] Technique to obtain coefficients of expansion.

quadrature_order [int] The highest order of the polynomial basis.

dimension_preference [list or tuple of int, optional] A set of weights specifying the relative importance of each uncertain variable (dimension).

nested [bool, optional] Set to enforce nested quadrature rules, if available (default is False).

Create a default instance of StochasticCollocation with:

```
>>> m = StochasticCollocation()
```

__str__()
Define the method block for a stoch_collocation experiment.

Display the method block created by a default instance of StochasticCollocation:

```
>>> m = StochasticCollocation()
>>> print(m)
method
  stoch_collocation
    sample_type = random
    samples = 10
    probability_levels = 0.1 0.5 0.9
    quadrature_order = 2
    non_nested
<BLANKLINE>
<BLANKLINE>
```

dakotathon.method.base.UncertaintyQuantificationBase.__str__

basis_polynomial_family
The type of basis polynomials used by the method.

dimension_preference
Weights specifying the relative importance of each dimension.

nested
Enforce use of nested quadrature rules.

quadrature_order
The highest order polynomial used by the method.

1.5 Dakota variable types

Dakota variables are the parameter sets to be iterated by a particular analysis method.

1.5.1 Variables base class

An abstract base class for all Dakota variable types.

class dakotathon.variables.base.**VariablesBase**(*variables='continuous_design'*, *descriptors=()*, ***kwargs*)

Bases: object

Describe features common to all Dakota variable types.

__init__(*variables='continuous_design'*, *descriptors=()*, ***kwargs*)
Create default variables parameters.

descriptors [str or tuple or list of str, optional] Labels for the variables.

variables [str, optional] The type of parameter set (default is 'continuous_design').

__str__()
Define the variables block of a Dakota input file.

descriptors
Labels attached to Dakota variables.

1.5.2 Continuous design

Implementation of a Dakota continous design variable.

class dakotathon.variables.continuous_design.**ContinuousDesign**(*descriptors=('x1', 'x2')*, *initial_point=None*, *lower_bounds=None*, *upper_bounds=None*, ***kwargs*)

Bases: *dakotathon.variables.base.VariablesBase*

Define attributes for Dakota continous design variables.

Continuous variables are defined by a real interval and are changed during the search for the optimal design.

__init__(*descriptors=('x1', 'x2')*, *initial_point=None*, *lower_bounds=None*, *upper_bounds=None*, ***kwargs*)

Create the parameter set for a continuous design variable.

descriptors [str or tuple or list of str, optional] Labels for the variables.

initial_point [tuple or list of numbers] Start points used by study variables.

lower_bounds [tuple or list of numbers] Minimum values used by the study variables.

upper_bounds [tuple or list of numbers] Maximum values used by the study variables.

****kwargs** Optional keyword arguments.

Create a default ContinuousDesign instance with:

```
>>> v = ContinuousDesign()
```

__str__()

Define the variables block for continous design variables.

Display the variables block created by a default instance of ContinuousDesign:

```
>>> v = ContinuousDesign()
>>> print(v)
variables
  continuous_design = 2
    descriptors = 'x1' 'x2'
    initial_point = -0.3 0.2
<BLANKLINE>
<BLANKLINE>
```

dakotathon.variables.base.VariablesBase.__str__

initial_point

Start points used by study variables.

lower_bounds

Minimum values of study variables.

upper_bounds

Maximum values of study variables.

1.5.3 Uniform uncertain

Implementation of a Dakota uniform uncertain variable.

class dakotathon.variables.uniform_uncertain.**UniformUncertain**(*descriptors=('x1', 'x2')*, *lower_bounds=(-2.0, -2.0)*, *upper_bounds=(2.0, 2.0)*, *initial_point=None*, ***kwargs*)

Bases: *dakotathon.variables.base.VariablesBase*

Define attributes for Dakota uniform uncertain variables.

The distribution lower and upper bounds are required specifications; the initial point is optional.

__init__(*descriptors=('x1', 'x2')*, *lower_bounds=(-2.0, -2.0)*, *upper_bounds=(2.0, 2.0)*, *initial_point=None*, ***kwargs*)

Create the parameter set for a uniform uncertain variable.

descriptors [str or tuple or list of str, optional] Labels for the variables.

initial_point [tuple or list of numbers, optional] Start points used by study variables.

lower_bounds [tuple or list of numbers] Minimum values used by the study variables.

****kwargs** Optional keyword arguments.

Create a default instance of UniformUncertain with:

```
>>> v = UniformUncertain()
```

__str__()

Define the variables block for a uniform uncertain variable.

Display the variables block created by a default instance of UniformUncertain:

```
>>> v = UniformUncertain()
>>> print(v)
variables
  uniform_uncertain = 2
    descriptors = 'x1' 'x2'
    lower_bounds = -2.0 -2.0
    upper_bounds = 2.0 2.0
<BLANKLINE>
<BLANKLINE>
```

dakotathon.variables.base.VariablesBase.__str__

initial_point

Start points used by study variables.

lower_bounds

Minimum values of study variables.

upper_bounds

Maximum values of study variables.

1.5.4 Normal uncertain

Implementation of a Dakota normal uncertain variable.

class dakotathon.variables.normal_uncertain.**NormalUncertain**(*descriptors=('x1', 'x2')*, *means=(0.0, 0.0)*, *std_deviations=(1.0, 1.0)*, *lower_bounds=None*, *upper_bounds=None*, *initial_point=None*, ***kwargs*)

Bases: *dakotathon.variables.base.VariablesBase*

Define attributes for Dakota normal uncertain variables.

The means and standard deviations are required specifications; the initial point, and the distribution lower and upper bounds are optional.

For vector and centered parameter studies, an inferred initial starting point is needed for uncertain variables. These variables are initialized to their means for these studies.

__init__(*descriptors=('x1', 'x2')*, *means=(0.0, 0.0)*, *std_deviations=(1.0, 1.0)*, *lower_bounds=None*, *upper_bounds=None*, *initial_point=None*, ***kwargs*)

Create the parameter set for a normal uncertain variable.

descriptors [str or tuple or list of str, optional] Labels for the variables.

means [tuple or list of numbers] First parameter of Gaussian distribution.

std_deviations [tuple or list of numbers] Second parameter of Gaussian distribution.

lower_bounds [tuple or list of numbers, optional] Minimum values used by the study variables.

upper_bounds [tuple or list of numbers, optional] Maximum values used by the study variables.

initial_point [tuple or list of numbers, optional] Start points used by study variables.

****kwargs** Optional keyword arguments.

Create a default instance of NormalUncertain with:

```
>>> v = NormalUncertain()
```

__str__()

Define the variables block for a normal uncertain variable.

Display the variables block created by a default instance of NormalUncertain:

```
>>> v = NormalUncertain()
>>> print(v)
variables
  normal_uncertain = 2
    descriptors = 'x1' 'x2'
    means = 0.0 0.0
    std_deviations = 1.0 1.0
<BLANKLINE>
<BLANKLINE>
```

dakotathon.variables.base.VariablesBase.__str__

initial_point
Start points used by study variables.

lower_bounds
Minimum values of study variables.

means
Mean values of study variables.

std_deviations
Standard deviations of study variables.

upper_bounds
Maximum values of study variables.

1.6 Dakota interface types

Dakota interfaces specify how function evaluations will be performed in order to map variables into responses.

1.6.1 Interface base class

An abstract base class for all Dakota interfaces.

class dakotathon.interface.base.**InterfaceBase**(*interface='direct'*, *id_interface='CSDMS'*, *analysis_driver='rosenbrock'*, *asynchronous=False*, *evaluation_concurrency=2*, *work_directory='/home/docs/checkouts/readthedocs.org/user_buil dakota/checkouts/stable/docs/source'*, *work_folder='run'*, *parameters_file='params.in'*, *results_file='results.out'*, ***kwargs*)

Bases: object

Describe features common to all Dakota interfaces.

__init__(*interface='direct'*, *id_interface='CSDMS'*, *analysis_driver='rosenbrock'*, *asynchronous=False*, *evaluation_concurrency=2*, *work_directory='/home/docs/checkouts/readthedocs.org/user_builds/d dakota/checkouts/stable/docs/source'*, *work_folder='run'*, *parameters_file='params.in'*, *results_file='results.out'*, ***kwargs*)

Create a default interface.

interface [str, optional] The Dakota interface type (default is 'direct').

id_interface [str, optional] Interface identifier.

analysis_driver [str, optional] Name of analysis driver for Dakota experiment (default is 'rosenbrock').

asynchronous [bool, optional] Set to perform asynchronous evaluations (default is False).

evaluation_concurrency [int, optional] Number of concurrent evaluations (default is 2).

work_directory [str, optional] The file path to the work directory (default is the run directory)

work_folder [str, optional] The name of the folders Dakota will create for each run (default is **run**).

parameters_file [str, optional] The name of the parameters file (default is **params.in**).

results_file [str, optional] The name of the results file (default is **results.out**).

****kwargs** Optional keyword arguments.

__str__()
Define the interface block of a Dakota input file.

asynchronous
State of Dakota evaluation concurrency.

evaluation_concurrency
Number of concurrent evaluations.

1.6.2 Direct

Implementation of a Dakota direct interface.

class dakotathon.interface.direct.**Direct**(***kwargs*)
Bases: *dakotathon.interface.base.InterfaceBase*

Define attributes for a Dakota direct interface.

__init__(***kwargs*)
Create a direct interface.

****kwargs** Optional keyword arguments.

Create an instance of Direct:

```
>>> f = Direct()
```

__str__()
Define the block for a direct interface.

dakotathon.interface.base.InterfaceBase.__str__

1.6.3 Fork

Implementation of a Dakota fork interface.

class dakotathon.interface.fork.**Fork**(***kwargs*)
Bases: *dakotathon.interface.base.InterfaceBase*

Define attributes for a Dakota fork interface.

__init__(***kwargs*)
Create a fork interface.

****kwargs** Optional keyword arguments.

Create an instance of Fork:

```
>>> f = Fork()
```

__str__()
Define the block for a fork interface.

dakotathon.interface.base.InterfaceBase.__str__

1.7 Dakota response types

Responses are the description of the model output data returned to Dakota upon evaluation of an interface.

1.7.1 Responses base class

An abstract base class for all Dakota responses.

class dakotathon.responses.base.**ResponsesBase**(*responses='response_functions'*, *response_descriptors=()*, *gradients='no_gradients'*, *hessians='no_hessians'*, ***kwargs*)

Bases: object

Describe features common to all Dakota responses.

__init__(*responses='response_functions'*, *response_descriptors=()*, *gradients='no_gradients'*, *hessians='no_hessians'*, ***kwargs*)

Create a default response.

responses [str, optional] The Dakota response type (default is 'response_functions').

response_descriptors [str or tuple or list of str, optional] Labels attached to the responses.

gradients [str, optional] Gradient type (default is 'no_gradients').

hessians [str, optional] Hessian type (default is 'no_hessians').

__str__()

Define the responses block of a Dakota input file.

response_descriptors

Labels attached to Dakota responses.

1.7.2 Response functions

Implementation of the Dakota response_function response type.

class dakotathon.responses.response_functions.**ResponseFunctions**(*response_descriptors=('y1',)*, *response_files=()*, *response_statistics=('mean',)*, ***kwargs*)

Bases: *dakotathon.responses.base.ResponsesBase*

Define attributes for Dakota response functions.

__init__(*response_descriptors=('y1',)*, *response_files=()*, *response_statistics=('mean',)*, ***kwargs*)

Create a response using response functions.

response_descriptors [str or tuple or list of str, optional] Labels attached to the responses.

response_files [str or tuple or list of str, optional] Model output files from which responses are calculated.

response_statistics [str or tuple or list of str, optional] Statistics used to generate responses.

****kwargs** Optional keyword arguments.

Create a ResponseFunctions instance:

```
>>> f = ResponseFunctions()
```

__str__()
Define the responses block of a Dakota input file.

dakotathon.responses.base.ResponsesBase.__str__

response_files
Model output files used in Dakota responses.

response_statistics
Statistics used to calculate Dakota responses.

1.8 Model plugins

Plugin classes for non-componentized models that can be called by Dakota.

1.8.1 Plugins base class

An abstract base class for all Dakota model plugins.

class dakotathon.plugins.base.**PluginBase**(***kwargs*)
Bases: object

Describe features common to all Dakota plugins.

calculate()
Calculate Dakota response functions.

call()
Call the model through the shell.

load(*output_file*)
Read data from a model output file.

output_file [str] The path to a model output file.

array_like A numpy array, or None on an error.

setup(*config*)
Configure model inputs.

Sets attributes using information from the run configuration file. The Dakota parsing utility `dprepro` reads parameters from Dakota to create a new input file from a template.

config [dict] Stores configuration settings for a Dakota experiment.

write(*params_file*, *results_file*)
Write a Dakota results file.

params_file [str] A Dakota parameters file.

results_file [str] A Dakota results file.

dakotathon.plugins.base.**write_dflt_file**(*tmpl_file*, *parameters_file*, *run_duration=1.0*)
Create a model input file populated with default values.

tmpl_file [str] The path to the template file defined for the model.

parameters_file [str] The path to the parameters file defined for the model.

run_duration [str, int, or float] Simulation run length, in undetermined units (default is 1.0).

str or None The path to the new dflt file, or None on an error.

dakotathon.plugins.base.**write_dtmpl_file**(*tmpl_file*, *dflt_input_file*, *parameter_names*)

Create a template input file for use by Dakota.

In the CSDMS framework, the tmpl file is an input file for a model, but with the parameter values replaced by *{parameter_name}*. Dakota uses the same idea. This function creates a Dakota dtmpl file from a CSDMS model tmpl file. Only the parameters used by Dakota are left in the tmpl format; the remainder are populated with default values for the model. The dtmpl file is written to the current directory.

tmpl_file [str] The path to the template file defined for the model.

dflt_input_file [str] An input file that contains the default parameter values for a model.

parameter_names [list of str] A list of parameter names for the model to be evaluated by Dakota.

str or None The path to the new dtmpl file, or None on an error.

1.8.2 HydroTrend

Provides a Dakota interface to the HydroTrend model.

class dakotathon.plugins.hydrotrend.**HydroTrend**(*input_dir='HYDRO_IN'*, *output_dir='HYDRO_OUTPUT'*, *input_file='HYDRO.IN'*, *input_template='HYDRO.IN.dtmpl'*, *hypsometry_file='HYDRO0.HYPS'*, *output_files=None*, *output_statistics=None*, ***kwargs*)

Bases: *dakotathon.plugins.base.PluginBase*

Represent a HydroTrend simulation in a Dakota experiment.

__init__(*input_dir='HYDRO_IN'*, *output_dir='HYDRO_OUTPUT'*, *input_file='HYDRO.IN'*, *input_template='HYDRO.IN.dtmpl'*, *hypsometry_file='HYDRO0.HYPS'*, *output_files=None*, *output_statistics=None*, ***kwargs*)

Configure a default HydroTrend simulation.

input_dir [str, optional] HydroTrend input directory (default is 'HYDRO_IN').

output_dir [str, optional] HydroTrend output directory (default is 'HYDRO_OUTPUT').

input_file [str, optional] HydroTrend input file (default is 'HYDRO.IN').

input_template [str, optional] Dakota template formed from HydroTrend input file (default is 'HYDRO.IN.dtmpl').

hypsometry_file [str, optional] The hypsometry file for the HydroTrend experiment.

output_files [str or list or tuple of str, optional] HydroTrend output files to analyze.

output_statistics [str or list or tuple of str, optional] Statistics to apply to HydroTrend output.

****kwargs** Optional keyword arguments.

Create a HydroTrend instance with:

```
>>> h = HydroTrend()
```

calculate()
Calculate Dakota output functions.

call()
Invoke HydroTrend through the shell.

load(*output_file*)
Read a column of data from a HydroTrend output file.

output_file [str] The path to a text HydroTrend output file.

array_like A numpy array, or None on an error.

setup(*config*)
Configure HydroTrend inputs.

Sets attributes using information from the run configuration file. The Dakota parsing utility `dprepro` reads parameters from Dakota to create a new HydroTrend input file from a template.

config [dict] Stores configuration settings for a Dakota experiment.

setup_directories(*config*)
Configure HydroTrend input and output directories.

config [dict] Configuration settings for a Dakota experiment.

setup_files(*config*)
Configure HydroTrend input and output files.

config [dict] Configuration settings for a Dakota experiment.

write(*params_file*, *results_file*)
Write the Dakota results file.

params_file [str] A Dakota parameters file.

results_file [str] A Dakota results file.

dakotathon.plugins.hydrotrend.**is_installed**()
Check whether HydroTrend is in the execution path.

1.9 Dakota console scripts

These console scripts are called as Dakota's analysis_driver: *dakota_run_component* for a CSDMS component, and *dakota_run_plugin* for a model interfaced by a Dakotathon plugin class.

1.9.1 The *dakota_run_component* script

Defines the *dakota_run_component* console script.

dakotathon.run_component.**main**()
Handle arguments to the *dakota_run_component* console script.

dakotathon.run_component.**run_component**(*params_file*, *results_file*)
Brokers communication between Dakota and a CSDMS component.

params_file [str] The path to the parameters file created by Dakota.

results_file [str] The path the results file returned to Dakota.

This console script provides a generic *analysis driver* for a Dakota experiment. At each evaluation step, Dakota calls this script with two arguments, the names of the parameters and results files:

1. The parameters file provides information on the current Dakota evaluation step, including the names and values of component variables and their responses. It also includes, as the *analysis component*, the name of a configuration file that stores information about the setup of the experiment, including the name of the component to call, input files, output file(s) to examine, and the statistic to apply to the output file(s).
2. The results file contains component output values in a format specified by the Dakota documentation.

Once the component is identified, a worker is created to perform three steps: preprocessing, execution, and postprocessing. In the preprocessing step, information from the configuration file is transferred to the component. In the execution step, the component is called, using the information passed from Dakota. In the postprocessing step, output from the component is read, and a single statistic (e.g., mean, median, max, etc.) is applied to it. This number, one for each response, is returned to Dakota through the results file, ending the Dakota evaluation step.

1.9.2 The *dakota_run_plugin* script

Defines the *dakota_run_plugin* console script.

dakotathon.run_plugin.**main**()
Handle arguments to the *dakota_run_plugin* console script.

dakotathon.run_plugin.**run_plugin**(*params_file*, *results_file*)
Brokers communication between Dakota and a model through files.

params_file [str] The path to the parameters file created by Dakota.

results_file [str] The path the results file returned to Dakota.

This console script provides a generic *analysis driver* for a Dakota experiment. At each evaluation step, Dakota calls this script with two arguments, the names of the parameters and results files:

1. The parameters file provides information on the current Dakota evaluation step, including the names and values of model variables and their responses. It also includes, as the *analysis component*, the name of a configuration file that stores information about the setup of the experiment, including the name of the model to call, input files, output file(s) to examine, and the statistic to apply to the output file(s).
2. The results file contains model output values in a format specified by the Dakota documentation.

Once the model is identified, an interface is created to perform three steps: preprocessing, execution, and postprocessing. In the preprocessing step, information from the configuration file is transferred to the component. In the execution step, the component is called, using the information passed from Dakota. In the postprocessing step, output from the component is read, and a single statistic (e.g., mean, median, max, etc.) is applied to it. This number, one for each response, is returned to Dakota through the results file, ending the Dakota evaluation step.

1.10 Dakota utilities

Helper functions for processing Dakota parameter and results files.

dakotathon.utils.**add_dyld_library_path**()
Add the *DYLD_LIBRARY_PATH* environment variable for Dakota.

dakotathon.utils.**compute_statistic**(*statistic*, *array*)
Compute the statistic used in a Dakota response function.

statistic [str] A string with the name of the statistic to compute ('mean', 'median', etc.).

array [array_like] An array data structure, such as a numpy array.

float The value of the computed statistic.

dakotathon.utils.**configure_parameters**(*params*)
Preprocess Dakota parameters prior to committing to a config file.

params [dict] Configuration parameters for a Dakota experiment that map to the items in the Dakota configuration file, **dakota.yaml**.

(dict, dict) An updated dict of Dakota configuration parameters, and a dict of substitutions used to create the Dakota template ("dtmpl") file.

dakotathon.utils.**deserialize**(*config_file*)
Load settings from a YAML configuration file.

dict Configuration settings in a dict.

dakotathon.utils.**get_attributes**(*obj*)
Get and format the attributes of an object.

section An object that has attributes.

dict The object's attributes.

dakotathon.utils.**get_configuration_file**(*params_file*)
Extract the configuration filepath from a Dakota parameters file.

params_file [str] The path to a Dakota parameters file.

str The path to the configuration file for the Dakota experiment.

dakotathon.utils.**get_response_descriptors**(*params_file*)
Extract response descriptors from a Dakota parameters file.

params_file [str] The path to a Dakota parameters file.

list A list of response descriptors for the Dakota experiment.

dakotathon.utils.**is_dakota_installed**()
Check whether Dakota is installed and in the execution path.

bool True if Dakota is callable.

dakotathon.utils.**to_iterable**(*x*)
Get an iterable version of an input.

x Anything.

If the input isn't iterable, or is a string, then a tuple; else, the input.

Courtesy http://stackoverflow.com/a/6711233/1563298

dakotathon.utils.**which**(*prog*, *env=None*)
Call the OS *which* function.

prog [str] The command name.

env [str, optional] An environment variable.

The path to the command, or None if the command is not found.

dakotathon.utils.**which_dakota**()
Locate the Dakota executable.

The path to the Dakota executable, or None if Dakota is not found.

dakotathon.utils.**write_results**(*results_file*, *values*, *labels*)
Write a Dakota results file from a set of input values.

results_file [str] The path to a Dakota results file.

values [array_like] A list or array of numeric values.

labels [str] A list of labels to attach to the values.

1.11 Dakota BMI classes

The Basic Model Interface (BMI) defines an interface for converting a standalone model into an integrated modeling framework component.

- Background: http://dx.doi.org/10.1016/j.cageo.2012.04.002
- Documentation: http://bmi-forum.readthedocs.io/

Basic Model Interface for the Dakota iterative systems analysis toolkit.

class dakotathon.bmi.**BmiDakota**
Bases: bmipy.bmi.Bmi

The BMI implementation for the CSDMS Dakota interface.

__init__()
Create a BmiDakota instance.

finalize()
Perform tear-down tasks for the model.

Perform all tasks that take place after exiting the model's time loop. This typically includes deallocating memory, closing files and printing reports.

get_component_name()
Name of the component.

str The name of the component.

get_current_time()
Current time of the model.

float The current model time.

get_end_time()
End time of the model.

float The maximum model time.

get_grid_edge_count(*grid*)
Get the number of edges in the grid.

grid [int] A grid identifier.

int The total number of grid edges.

get_grid_edge_nodes(*grid*, *edge_nodes*)
Get the edge-node connectivity.

grid [int] A grid identifier.

edge_nodes [ndarray of int, shape *(2 x nnodes,)*] A numpy array to place the edge-node connectivity. For each edge, connectivity is given as node at edge tail, followed by node at edge head.

ndarray of int The input numpy array that holds the edge-node connectivity.

get_grid_face_count(*grid*)
Get the number of faces in the grid.

grid [int] A grid identifier.

int The total number of grid faces.

get_grid_face_nodes(*grid*, *face_nodes*)
Get the face-node connectivity.

grid [int] A grid identifier.

face_nodes [ndarray of int] A numpy array to place the face-node connectivity. For each face, the nodes (listed in a counter-clockwise direction) that form the boundary of the face.

ndarray of int The input numpy array that holds the face-node connectivity.

get_grid_node_count(*grid*)
Get the number of nodes in the grid.

grid [int] A grid identifier.

int The total number of grid nodes.

get_grid_nodes_per_face(*grid*, *nodes_per_face*)
Get the number of nodes for each face.

grid [int] A grid identifier.

nodes_per_face [ndarray of int, shape *(nfaces,)*] A numpy array to place the number of edges per face.

ndarray of int The input numpy array that holds the number of nodes per edge.

get_grid_origin(*grid*, *origin*)
Get coordinates for the lower-left corner of the computational grid.

grid [int] A grid identifier.

origin [ndarray of float, shape *(ndim,)*] A numpy array to hold the coordinates of the lower-left corner of the grid.

ndarray of float The input numpy array that holds the coordinates of the grid's lower-left corner.

get_grid_rank(*grid*)
Get number of dimensions of the computational grid.

grid [int] A grid identifier.

int Rank of the grid.

get_grid_shape(*grid*, *shape*)
Get dimensions of the computational grid.

grid [int] A grid identifier.

shape [ndarray of int, shape *(ndim,)*] A numpy array into which to place the shape of the grid.

ndarray of int The input numpy array that holds the grid's shape.

get_grid_size(*grid*)
Get the total number of elements in the computational grid.

grid [int] A grid identifier.

int Size of the grid.

get_grid_spacing(*grid*, *spacing*)
Get distance between nodes of the computational grid.

grid [int] A grid identifier.

spacing [ndarray of float, shape *(ndim,)*] A numpy array to hold the spacing between grid rows and columns.

ndarray of float The input numpy array that holds the grid's spacing.

get_grid_type(*grid*)
Get the grid type as a string.

grid [int] A grid identifier.

str Type of grid as a string.

get_grid_x(*grid*, *x*)
Get coordinates of grid nodes in the x direction.

grid [int] A grid identifier.

x [ndarray of float, shape *(nrows,)*] A numpy array to hold the x-coordinates of the grid node columns.

ndarray of float The input numpy array that holds the grid's column x-coordinates.

get_grid_y(*grid*, *y*)
Get coordinates of grid nodes in the y direction.

grid [int] A grid identifier.

y [ndarray of float, shape *(ncols,)*] A numpy array to hold the y-coordinates of the grid node rows.

ndarray of float The input numpy array that holds the grid's row y-coordinates.

get_grid_z(*grid*, *z*)
Get coordinates of grid nodes in the z direction.

grid [int] A grid identifier.

z [ndarray of float, shape *(nlayers,)*] A numpy array to hold the z-coordinates of the grid nodes layers.

ndarray of float The input numpy array that holds the grid's layer z-coordinates.

get_input_var_names()
List of a model's input variables.

Input variable names must be CSDMS Standard Names, also known as *long variable names*.

list of str The input variables for the model.

Standard Names enable the CSDMS framework to determine whether an input variable in one model is equivalent to, or compatible with, an output variable in another model. This allows the framework to automatically connect components.

Standard Names do not have to be used within the model.

get_output_var_names()
List of a model's output variables.

Output variable names must be CSDMS Standard Names, also known as *long variable names*.

list of str The output variables for the model.

get_start_time()
Start time of the model.

Model times should be of type float.

float The model start time.

get_time_step()
Current time step of the model.

The model time step should be of type float.

float The time step used in model.

get_time_units()
Time units of the model.

float The model time unit; e.g., *days* or *s*.

CSDMS uses the UDUNITS standard from Unidata.

get_value(*name*, *dest*)
Get a copy of values of the given variable.

This is a getter for the model, used to access the model's current state. It returns a *copy* of a model variable, with the return type, size and rank dependent on the variable.

name [str] An input or output variable name, a CSDMS Standard Name.

dest [ndarray] A numpy array into which to place the values.

ndarray The same numpy array that was passed as an input buffer.

get_value_at_indices(*name*, *dest*, *inds*)
Get values at particular indices.

name [str] An input or output variable name, a CSDMS Standard Name.

dest [ndarray] A numpy array into which to place the values.

indices [array_like] The indices into the variable array.

array_like Value of the model variable at the given location.

get_value_ptr(*name*)
Get a reference to values of the given variable.

This is a getter for the model, used to access the model's current state. It returns a reference to a model variable, with the return type, size and rank dependent on the variable.

name [str] An input or output variable name, a CSDMS Standard Name.

array_like A reference to a model variable.

get_var_grid(*name*)
Get grid identifier for the given variable.

name [str] An input or output variable name, a CSDMS Standard Name.

int The grid identifier.

get_var_itemsize(*name*)
Get memory use for each array element in bytes.

name [str] An input or output variable name, a CSDMS Standard Name.

int Item size in bytes.

get_var_location(*name*)
Get the grid element type that the a given variable is defined on.

The grid topology can be composed of *nodes*, *edges*, and *faces*.

node A point that has a coordinate pair or triplet: the most basic element of the topology.

edge A line or curve bounded by two *nodes*.

face A plane or surface enclosed by a set of edges. In a 2D horizontal application one may consider the word "polygon", but in the hierarchy of elements the word "face" is most common.

name [str] An input or output variable name, a CSDMS Standard Name.

str The grid location on which the variable is defined. Must be one of *"node"*, *"edge"*, or *"face"*.

CSDMS uses the ugrid conventions to define unstructured grids.

get_var_nbytes(*name*)
Get size, in bytes, of the given variable.

name [str] An input or output variable name, a CSDMS Standard Name.

int The size of the variable, counted in bytes.

get_var_type(*name*)
Get data type of the given variable.

name [str] An input or output variable name, a CSDMS Standard Name.

str The Python variable type; e.g., `str`, `int`, `float`.

get_var_units(*name*)
Get units of the given variable.

Standard unit names, in lower case, should be used, such as `meters` or `seconds`. Standard abbreviations, like `m` for meters, are also supported. For variables with compound units, each unit name is separated by a single space, with exponents other than 1 placed immediately after the name, as in `m s-1` for velocity, `W m-2` for an energy flux, or `km2` for an area.

name [str] An input or output variable name, a CSDMS Standard Name.

str The variable units.

CSDMS uses the UDUNITS standard from Unidata.

initialize(*config_file*)
Perform startup tasks for the model.

Perform all tasks that take place before entering the model's time loop, including opening files and initializing the model state. Model inputs are read from a text-based configuration file, specified by *filename*.

config_file [str, optional] The path to the model configuration file.

Models should be refactored, if necessary, to use a configuration file. CSDMS does not impose any constraint on how configuration files are formatted, although YAML is recommended. A template of a model's configuration file with placeholder values is used by the BMI.

set_value(*name*, *values*)
Specify a new value for a model variable.

This is the setter for the model, used to change the model's current state. It accepts, through *src*, a new value for a model variable, with the type, size and rank of *src* dependent on the variable.

var_name [str] An input or output variable name, a CSDMS Standard Name.

src [array_like] The new value for the specified variable.

set_value_at_indices(*name*, *inds*, *src*)
Specify a new value for a model variable at particular indices.

var_name [str] An input or output variable name, a CSDMS Standard Name.

indices [array_like] The indices into the variable array.

src [array_like] The new value for the specified variable.

update()
Advance model state by one time step.

Perform all tasks that take place within one pass through the model's time loop. This typically includes incrementing all of the model's state variables. If the model's state variables don't change in time, then they can be computed by the *initialize()* method and this method can return with no action.

class dakotathon.bmi.**CenteredParameterStudy**
Bases: *dakotathon.bmi.BmiDakota*

BMI implementation of a Dakota centered parameter study.

initialize(*filename=None*)
Create a Dakota instance and input file.

filename [str, optional] Path to a Dakota configuration file.

class dakotathon.bmi.**MultidimParameterStudy**
Bases: *dakotathon.bmi.BmiDakota*

BMI implementation of a Dakota multidim parameter study.

initialize(*filename=None*)
Create a Dakota instance and input file.

filename [str, optional] Path to a Dakota configuration file.

class dakotathon.bmi.**PolynomialChaos**
Bases: *dakotathon.bmi.BmiDakota*

BMI implementation of a Dakota study with the polynomial chaos method.

initialize(*filename=None*)
Create a Dakota instance and input file.

filename [str, optional] Path to a Dakota configuration file.

class dakotathon.bmi.**PsuadeMoat**
Bases: *dakotathon.bmi.BmiDakota*

BMI implementation of a Dakota study with the PSUADE MOAT method.

initialize(*filename=None*)
Create a Dakota instance and input file.

filename : str, optional Path to a Dakota configuration file.

class dakotathon.bmi.**Sampling**
Bases: *dakotathon.bmi.BmiDakota*

BMI implementation of a Dakota sampling study.

initialize(*filename=None*)
Create a Dakota instance and input file.

filename [str, optional] Path to a Dakota configuration file.

class dakotathon.bmi.**StochasticCollocation**
Bases: *dakotathon.bmi.BmiDakota*

BMI implementation of a Dakota study with the stochastic collocation method.

initialize(*filename=None*)
Create a Dakota instance and input file.

filename [str, optional] Path to a Dakota configuration file.

class dakotathon.bmi.**VectorParameterStudy**
Bases: *dakotathon.bmi.BmiDakota*

BMI implementation of a Dakota vector parameter study.

initialize(*filename=None*)
Create a Dakota instance and input file.

filename [str, optional] Path to a Dakota configuration file.

CHAPTER 2

Indices and tables

- genindex
- modindex
- search

Python Module Index

d

Index

Symbols

F

G

H

I

L

M

N

P

Q

R