pounce Documentation

JD

Oct 11, 2019

Contents:

CHAPTER 1

helpers package

1.1 Submodules

1.2 helpers.baseclass module

class helpers.baseclass.**BaseClass**(*class_dict*, **further_classes*)
Bases: object

Skeleton for most classes to inherit from. Provides methods for user input and to choose subclasses from a user input string

classmethod create(*class_dict*, **args*)
Choose subclass via a input string and init. Further user input for this class is passed to init as a dict

classmethod defaults(**args*, *with_type=True*)
get defaults for a class, first from its own class and parents, then from further input classes. _type (i.e. the subclass name) is added optionally

classmethod defaults_class(*key=None*)
get defaults for a class from its own class and parents, via the multi resolution order list.

classmethod name()
translate class name from camel case (MyClass) to underscore (my_class) for consistent yml input

read_prms(*input_prm_dict*, **further_classes*)
Gets user input for own class as a dictionary. Compares user input against defaults for parent class and subclass. Throws errors for invali input, else converts input dict to class attributes

classmethod subclass(*string*)
choose subclass of a class by string

exception helpers.baseclass.**InputPrmError**
Bases: Exception

1.3 helpers.config module

class helpers.config.**GeneralConfig**(**args*)
Bases: *helpers.baseclass.BaseClass*

Provides a container for some general config options and their default values.

copy_to_globels()

defaults_ = {'archive_level': 0, 'project_name': 'NODEFAULT'}

helpers.config.**config**(*prmfile*)
Reads all user input and sets up (sub-)classes according to this input. Partially, setup is called from uq-method-specific routines. The sim object is copied to globels to be available for pickle.

helpers.config.**config_list**(*name*, *prms*, *class_init*, **args*, *sub_list_name=None*)
Checks for correct input format for list type input and initializes (sub-) class for given input

helpers.config.**restart**(*prmfile=None*)
restart simulation from a pickle file.

1.4 helpers.default_yml module

class helpers.default_yml.**DefaultFile**
Bases: object

container for the defaults dictionary and its manipulating functions

clean()
remove input prms with value "dummy_unused"; wrapper for recursive function **clean_**

clean_(*dict_in*)
recursively remove input prms with value "dummy_unused"

expand_to_several(*sub*, *list_name*, *keys=None*, *exclude=[]*)
Some classes will have several instances. In the ini file, it is desirable to specify some defaults for all instances, and some for each instance seperately. In this routine, the defaults of a class are split into a dict (if keys are given) or a list with one example entry (else) each including all defaults of the class but the excluded ones. This sub-dict/list is then placed inside the default dict for the class. The excluded items are specified once and are used for all instances of the class.

get_list_defaults(*parent*, *args='default'*)
output a list of all implemented types of list-input items (e.g. stoch_var, qoi)

get_machine()

inquire(*msg*)

inquire_subclass(*parent_class*, *description=None*)
Asks for user input to choose one of the available subclasses

print_()
print dict of defaults either to stdout or to specified file

process_subclass(*parent*)
inquires subclass, and adds its default prms to dict. Returns the class as an object

helpers.default_yml.**print_default_yml_file**()
Asks for user input to choose one of the available subclasses, builds up dictionary of defaults for all variables for this sub class combinatiion, then prints default YML file using yaml.dump.

1.5 helpers.globels module

helpers.globels.**archive**()

helpers.globels.**archive_loc**()

helpers.globels.**iteration**(*wrapped_function*)

helpers.globels.**run_step**(*description*, *func*, **args*, ***kwargs*)

helpers.globels.**update_step**(*string=None*)

1.6 helpers.printtools module

class helpers.printtools.**Bcolors**
Bases: object

color and font style definitions for changing output appearance

BLACK = '\x1b[0;30m'

BLUE = '\x1b[0;34m'

BOLD = '\x1b[1m'

CYAN = '\x1b[0;36m'

ENDC = '\x1b[0m'

GREEN = '\x1b[0;32m'

PURPLE = '\x1b[0;35m'

RED = '\x1b[0;31m'

UNDERLINE = '\x1b[4m'

WHITE = '\x1b[0;37m'

YELLOW = '\x1b[0;33m'

class helpers.printtools.**StdOutTable**(**args*)
Bases: object

Buffers several values for each batch for stdout in ordered table. Called in three steps: - before loop over batches: init class and set_descriptions - during loop over batches: update (for each batch) - after loop over batches: print

p_print()

print_row_by_name(*attr*)

set_descriptions(**args*)

update(*level*)

class helpers.printtools.**TableRow**(*attr*)
Bases: object

add_string(*string*)

p_print(*l*, *lm2*)

helpers.printtools.**blue**(*text*)

helpers.printtools.**cyan**(*text*)

helpers.printtools.**green**(*text*)

helpers.printtools.**indent_in**()

helpers.printtools.**indent_out**()

helpers.printtools.**p_print**(*msg*)
wrapper for normal stdout print commands

helpers.printtools.**print_header**()

helpers.printtools.**print_major_section**(*msg*, *color='stdcolor'*)
such as at the beginning of iterations

helpers.printtools.**print_step**(*msg*)

helpers.printtools.**red**(*text*)

helpers.printtools.**time_to_str2**(*sec*)

helpers.printtools.**yellow**(*text*)

1.7 helpers.tools module

class helpers.tools.**Empty**
Bases: object

exception helpers.tools.**InputPrmError**
Bases: Exception

helpers.tools.**isvalidlist**(*arg*)
time lists have three entries h,m,s

helpers.tools.**parse_time_to_seconds**(*arg*)
parse different formats to give time in the yml parameter file.

helpers.tools.**safe_sqrt**(*arg*)
for sqrt of negative values, print a warning instead of crashing.

helpers.tools.**sec_to_list**(*sec*)
helper for time_sto_str

helpers.tools.**time_to_str**(*sec*)

helpers.tools.**time_to_str2**(*sec*)

1.8 Module contents

CHAPTER 2

machine package

2.1 Submodules

2.2 machine.cray module

class machine.cray.**Cray**(*class_dict*)
Bases: *machine.machine.Machine*

Definition of Cray Hazelhen machine.

allocate_resources()
Takes the properties of the batch (number of current samples, walltime and cores of current sample) as well as the machine properties or machine input (number of cores per node, max nodes etc.) and outputs number of cores and number of parallel runs for this batch.

check_errorfile(*batch*)
open error file and parse errrors. Well, parse is a strong word here.

defaults_ = {'max_total_work': 3600000.0, 'max_walltime': 86400, 'n_max_cores': 100

get_best_option(*batch*)
Loop over all possible combinations of n_parallel_runs and n_sequential_runs. Get Rating for all of them. Pick the best one.

get_package_properties(*batch*)
define a "package" of runs to fill a node, e.g. 4 parallel runs with cores_per_sample=6. trivial if cores_per_sample >= 24.

read_qstat()
run 'qstat' on cray and read output

run_batches()
Runs batches by generating the necessary jobfiles, submitting them, and supervising the queuing status.

submit_job(*batch*)
Generates the necessary jobfile and submits job for a batch

to_ssh(*args*)
converts a command into the same command passed via ssh (each argument is an item of a list; in the ssh command, the original command appears as one argument and thus one string)

wait_finished()
Monitors all jobs on Cray Hazelhen HPC queue. Checks if jobfile finished.

class machine.cray.**Option**(*batch*, *n_sequential_runs=None*, *n_parallel_runs=None*)
Bases: object

One combination of n_parallel_runs and n_sequential_runs. Has a Rating based on efficiency (few idling cores) and expcted queuing time. Invalid if does not match criteria of selected queue.

check_valid(*batch*)
Invalid if does not match criteria of selected queue.

rating(*batch*)
Rating based on efficiency (few idling cores) and expcted queuing time.

machine.cray.**get_queue**(*batch*)
check which queue the job is eligible for: if possible, run on multi. If too small, run on small. If too large, run on long (>4h)

machine.cray.**long_queue**(*batch*)
with max_cores, walltime exceeds 4h

machine.cray.**multi_queue**(*batch*)
preferred queue: n_nodes>=48, walltime < 4h

machine.cray.**small_queue**(*batch*)
multi queue cannot be filled with walltime > 5 min

2.3 machine.local module

class machine.local.**Local**(**args*)
Bases: *machine.machine.Machine*

Class: Defines local machine. Since no queuing is required, this all reduces to very basic routines.

allocate_resources()

defaults_ = {'mpi': 'NODEFAULT'}

defaults_add = {'Batch': {'avg_walltime': 'dummy_unused', 'cores_per_sample': 1}}

run_batches()
Runs a job by calling a subprocess.

2.4 machine.machine module

class machine.machine.**Machine**(**args*)
Bases: *simulation.simulation.Stage*, *helpers.baseclass.BaseClass*

defines the machine that an external job is run on. We call the processing of an external job (i.e. allocating resouces, preparation, and running) as a stage. Each stage can (theoretically) be run on a different machine. E.g., post-processing can be doen locally. Therefore, the different stages are instances of machine subclasses.

2.5 Module contents

CHAPTER 3

sampling package

3.1 Submodules

3.2 sampling.sampling module

class sampling.sampling.**Collocation**(*class_dict*, **further_classes*)
Bases: *sampling.sampling.Sampling*

Sampling at collocation nodes based on ChaosPy routines Smolyak sparse grid is possible

defaults_ = {'poly_deg': 'NODEFAULT', 'sparse_grid': 'NODEFAULT'}

get()
get samples

sampling_prms()
For mean and variance, only weights would be used. The rest is for response surface creation.

class sampling.sampling.**MonteCarlo**(*class_dict*, **further_classes*)
Bases: *sampling.sampling.Sampling*

Vanilla Monte Carlo sampling

get()
get samples

sampling_prms()
parameters specific to the sampling strategy which are needed by the solver or by the post-processing routines

class sampling.sampling.**Sampling**(*class_dict*, **further_classes*)
Bases: *helpers.baseclass.BaseClass*

parent class with placeholders for the sampling strategies

get()
get samples

sampling_prms()
parameters specific to the sampling strategy which are needed by the solver or by the post-processing routines

3.3 Module contents

CHAPTER 4

simulation package

4.1 Submodules

4.2 simulation.simulation module

class simulation.simulation.**Iteration**(*n=None*, *name=None*)
Bases: object

Helper class for iterations. Could be extended in the future to store all information about samples etc. which is currently overwritten. Note that the "iteration" decorator is located in globels.

start()
StdOut of A) section and B) skipped steps in case of a restart.

class simulation.simulation.**Simulation**(*class_dict*)
Bases: *helpers.baseclass.BaseClass*

Organizes how the simulation is run, i.e. how iterations follow each other in a loop and how an iteration looks. Simulation is a parent class to UqMethod, where routines can be overwritten.

process_simulation_postproc()

run()
Default main simulation loop: Iterate until finished or maximum number of iterations is reached, the npost-process if necessary.

run_iteration()

class simulation.simulation.**Stage**(**args*)
Bases: object

A Stage is a set of batches that is run externally. This can be simulations or post-processing. Processing this stage includes determining the required resources on the system, preparation (writing input) and running the jobs. Each stage can be run on a different system. The according subclass of Machine therefore inherits from Stage.

active_batches
if no samples are computed in this iteration, the batch is not active.

check_all_finished()

fill(*name*, *multi_sample*)

prepare_set()
Wrapper for preparation of all batches

process()
core function of the class: process the stage

unfinished_batches

4.3 Module contents

CHAPTER 5

solver package

5.1 Submodules

5.2 solver.flexi module

class solver.flexi.**FieldSolution**(**args*)
Bases: solver.flexi.QoI

Takes the whole field solution as quantity of interest.

Caution: routines starting with **prepare_**... will be renamed to "prepare" as part of the create_by_stage routine.

prepare_iteration_postproc()

prepare_simulation_postproc()

class solver.flexi.**Flexi**(**args*)
Bases: *solver.solver.Solver*

Runs with the POUNCE-adaptation of FLEXI, i.e. with the executable flexibatch and the according post-processing tools.

class QoI(**args*)
Bases: *solver.solver.QoI*

Parent class for all FLEXI QoI's

defaults_ = {'prmfile': ''}

get_derived_quantity(*quantity_name*)
Readin sigma_sq or avg_walltime for MLMC.

get_work_mean()
For Flexi, avg work is already read from HDF5 file during check_all_finished

check_finished()
Check last lines of logfiles (stdout) for confirmation that the batch is finished. Also retrieve average work, which is written to the log file as well (as part of flexibatch)

defaults_ = {'prmfile': 'parameter_flexi.ini', 'solver_prms': {'MeshFile': 'NODEFAU

defaults_add = {'StochVar': {'i_occurrence': {}, 'i_pos': {}, 'name': 'NODEFAULT'}

h5write(*h5f*, *name*, *prm*)
helper function for correct data formatting in Fortran readable HDF5 files.

prepare()
Prepares the simulation by generating the run_command and writing the HDF5 file containing all samples of the current iteration and the current samples.

write_hdf5(*file_name*, *solver_prms*, *further_prms*)
Writes the HDF5 file containing all necessary data for flexi run to run.

class solver.flexi.**RecordPoints**(**args*)
Bases: solver.flexi.QoI

Takes a solution time sereis evaluated at record points as QoI.

Caution: routines starting with **prepare_**... will be renamed to "prepare" as part of the create_by_stage routine.

defaults_ = {'time_span': [0.0, 10000000000.0]}

prepare_iteration_postproc()

prepare_simulation_postproc()

5.3 solver.internal module

class solver.internal.**Integral**(**args*)
Bases: solver.internal.QoI

Takes the integral of the solution as quantity of interest.

Caution: routines starting with **prepare_**... will be renamed to "prepare" as part of the create_by_stage routine.

prepare_iteration_postproc()

prepare_simulation_postproc()

class solver.internal.**Internal**(**args*)
Bases: *solver.solver.Solver*

Dummy python solver for testing. python source files are located in the externals directory I/O via HDF5.

class QoI(**args*)
Bases: *solver.solver.QoI*

Parent class for the dummy solver's QoI(s)

get_derived_quantity(*quantity_name*)
Readin sigma_sq for MLMC.

get_work_mean()

defaults_ = {'solver_prms': {'nPoints': 'NODEFAULT'}}

h5write(*h5f*, *name*, *prm*)

prepare()
Prepares the simulation by generating the run_command and writing the HDF5 file containing all samples of the current iteration and the current level.

write_hdf5(*file_name*, *prms*)
Writes the HDF5 file containing all necessary data for the internal to run.

5.4 solver.solver module

class solver.solver.**Batch**(*class_dict*, **further_classes*)
Bases: *helpers.baseclass.BaseClass*

A batch consists of a set of computations. This can be either simulations or post-processing. It is therefore the parent class to Solver and QoI

check_finished()
default: do not carry out a check after a batch is run simply assume all are finished.

defaults_ = {'avg_walltime': 300.0, 'cores_per_sample': 1, 'exe_path': 'NODEFAULT'}

errfile_names

logfile_names

n_runs

prepare(*simulation*)
placeholder; should be overwritten by each subclass.

run_id(*i*)
needed to distinguish input and output files in the case of several runs per batch (i.e. if one solver is run several times instead of a loop over all samples as part of the external solver)

class solver.solver.**QoI**(**args*)
Bases: *solver.solver.Batch*

QoIs are always chosen automatically according to the chosen Solver.

classmethod create_by_stage(*name*, *prms*, **args*)
Some QoI's contain prepare functions for different stages. Here, the functions are renamed to the general "prepare" according to the respective stage string given in "name". QoI parameters are joined: some are given for all stages (prms_other), others are stage-specific (prms_loc).

class solver.solver.**Solver**(**args*)
Bases: *solver.solver.Batch*

Solver is the parent class to subclasses which include routines specidifc to the used solver. Here only the main simulation is considered as opposed to the according QoI's, which are defined separately.

defaults_ = {'avg_walltime': 'NODEFAULT', 'cores_per_sample': 'NODEFAULT', 'solver_p

5.5 Module contents

CHAPTER 6

stochvar package

6.1 Submodules

6.2 stochvar.stochvar module

class stochvar.stochvar.**Normal**(*input_prm_dict*, **args*)
Bases: *stochvar.stochvar.StochVar*

normal distribution uses numpy and chaospy routines

defaults_ = {'mean': 'NODEFAULT', 'standard_deviation': 'NODEFAULT'}

draw_samples(*n_samples*)

class stochvar.stochvar.**StochVar**(*class_dict*, **further_classes*)
Bases: *helpers.baseclass.BaseClass*

parent class for stochastic variables

class stochvar.stochvar.**Uniform**(*input_prm_dict*, **args*)
Bases: *stochvar.stochvar.StochVar*

uniform distribution uses numpy and chaospy routines

defaults_ = {'bounds': 'NODEFAULT'}

draw_samples(*n_samples*)

6.3 Module contents

CHAPTER 7

uqmethod package

7.1 Submodules

7.2 uqmethod.mlmc module

class uqmethod.mlmc.**Mlmc**(*input_prm_dict*)
Bases: *uqmethod.uqmethod.UqMethod*

Multilevel Monte Carlo The number of levels is prescribed, the number of samples is adapted iteratively in a prescribed number of iterations (convergence rate and work per sample are obtained empirically).

SamplingMethod
alias of *sampling.sampling.MonteCarlo*

classmethod default_yml(*d*)
MLMC specific layout of the default yml file.

defaults_ = {'n_max_iter': 'NODEFAULT', 'reset_seed': False, 'tolerance': None, 'to

defaults_add = {'QoI': {'optimize': False}, 'Solver': {'n_warmup_samples': 'NODEFAU

prepare_next_iteration()
Compute number of samples for next iteration. - evaluate sigma^2 and avg work. - get optimal number of samples on every level

(given prescribed tolerance or total work)

- approach this numbr carefully and iteratively

setup(*prms*)
Set up data structure for an MLMC simulation Includes levels and sublevels, quantities of interest, each initiallized according to chosen solver, and stages (main simulation and post proc) according to chosen machine.

setup_level(*i*, *sub_fine*, *sub_coarse*)
set up a level, connect to its sublevels, and add the samples container

setup_qoi(*subdict*, *level*)
set up quantity of interest for a level and make the sublevels its participants

7.3 uqmethod.sc module

class uqmethod.sc.**Sc**(*input_prm_dict*)
Bases: *uqmethod.uqmethod.UqMethod*

Stochastic Collocation (non-adaptive)

SamplingMethod
alias of *sampling.sampling.Collocation*

classmethod default_yml(*d*)

prepare_next_iteration()
There is only one "iteration", so no next one needs to be prepared.

setup(*prms*)
Only one batch is needed (called solver)

7.4 uqmethod.uqmethod module

class uqmethod.uqmethod.**UqMethod**(*class_dict*)
Bases: *simulation.simulation.Simulation*, *helpers.baseclass.BaseClass*

Parent class for different uq methods Inherits from Simulation, since it is also the driver class for the whole simulation.

classmethod default_yml(*d*)

get_samples(*batches*)
The sampling method is determined during setup, so this is just a simple wrapper.

7.5 Module contents

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

m

s

u

Index

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

W

Y