Par Sim

Simulation Model Parameterization and Workflow Automation

parsim Documentation

Release 2.1.0.post1

Ola Widlund

Aug 03, 2020

CONTENTS

CHAPTER

ONE

GETTING STARTED

Parsim is a tool for working with parameterized simulation models. The primary objective is to facilitate quality assurance of simulation projects. The tool supports a scripted and automated workflow, where verified and validated simulation models are parameterized, so that they can altered/modified in well-defined ways and reused with minimal user invention. All events are logged on several levels, to support traceability, project documentation and quality control.

Parsim provides basic functionality for generating studies based on common design-of experiments (DOE) methods, for example using factorial designs, response surface methods or random sampling, like Monte Carlo or Latin Hypercube.

Parsim can also be used as an interface to the Dakota library; Dakota is run as a subprocess, generating cases from a Parsim model template.

1.1 How it works

Once a prototype simulation case has been created, a corresponding simulation *model template* is created by collecting all simulation input files, data files and scripts into a *template directory*. The template directory may contain subdirectories, for example to separate files for different phases of the simulation, or for different solvers, pre- and post-processors, etc.

The text files in a model template can then be parameterized by replacing numerical values, or text strings with macro names. The filename extension `.macro` is added to all files containing macros. A model template usually defines default values for all its parameters in a specific parameter file.

When a simulation case is created, the model template directory is recursively replicated to create a *case directory*. Parameterized files with extension `.macro` are processed by a macro processor which replaces parameter names by actual values. The processed file has the same name as the template file, but without the `.macro` extension.

Parsim operations can also be carried out on a *study*, containing multiple cases. A study is a directory containing multiple case directories. The cases of a study are defined in a *caselist* file; the first column contains the case name, while other columns define values of parameters defined in a header row. A study can also be created directly from one of the built-in DOE (Design Of Experiments) sampling schemes, e.g. Monte Carlo, Latin-Hypercube, full or fractional factorial designs, etc. When the DOE functionality is used, statistical distributions are specified in the parameter file for the variable parameters.

When creating a case or a study, custom parameter values can be defined on the command line, in a separate *parameter file*, or in a *caselist* defining multiple cases of a *study*.

Your simulation project lives in a Parsim *project directory*, which holds all cases and studies of the project. The project directory holds Parsim configuration settings and logs project events, like creation of cases and studies, serious errors, change of configuration settings, etc.

The best way to learn more about how you can use Parsim, is to follow the tutorial examples.

1.2 Installation

Parsim is available at both PyPI, the Python Package Index and as a conda package through the conda-forge repository, depending on which Python distribution and package manager you use (`pip` and `conda`, respectively).

The Parsim installation requires and automatically installs the Python library pyexpander, which is used for macro and parameter expansion (parameterization of input files). The DOE (Design of Experiments) functionality is provided by the pyDOE2, numpy and scipy libraries. The pandas library has also been included, so that the Python API can provide results and caselist data as pandas DataFrames. If you want to use the Dakota toolkit, it is installed separately; the `dakota` executable should be in your `PATH`.

Note: If you experience issues with the installation, it is recommended to first make a clean and fully functional installation of the NumPy, SciPy and pandas libraries. The best way to do this depends on which Python distribution you use. The anaconda Python distribution is highly recommended. It works well on both Windows and Linux.

1.2.1 Installation from PyPI

Use the package installer `pip` to install:

```
pip install parsim
```

1.2.2 Installation with conda

Note that you need to select the `conda-forge` channel to find parsim with conda.

To install in your base environment:

```
conda install -c conda-forge parsim
```

Alternatively, create a separate conda environment (here called `psm-env`) for using parsim:

```
conda create -n psm-env -c conda-forge parsim
conda activate psm-env
```

CHAPTER

TWO

BASIC CONCEPTS

2.1 Projects

A Parsim simulation project lives in a project directory. The project directory holds alla cases and studies of the project, as well as custom Parsim configuration settings and event logs for the project.

To start using Parsim with your simulation, you need to initiate Parsim in an existing directory, using the `psm init` command. This directory is now your Parsim project directory.

2.1.1 Configuration settings

There are a number of configuration settings, which control how Parsim works with you project. All settings have sensible defaults, which can be altered when the project is created. For an existing project, configuration settings can be modified with the `psm config` command.

The following table describes the available project configuration settings.

Parameter	Description	Default value(s)
`template_root`	Path to directory holding model templates for the project. Either an absolute path, or relative to the project directory.	'modelTemplates'
`default_template`	Name of default model template directory (inside the `template_root` directory).	'default'
`default_parameter_file`	Name of file defining default values for all parameters of a model template. Path relative to template directory.	'default.parameters'
`default_executable`	Name of default executable for `psm run` command, if not given on the command line.	None
`python_exe`	Path/name of Python executable, for use with Python scripts and the `psm run` command.	'python'
`dakota_exe`	Path/name of Dakota executable, for using Parsim as an interface to the Dakota library.	'dakota'
`psm_ignore_file`	File containing ignore patterns. Files/directories matching a pattern will be ignored when templates are replicated into new cases.	'.psmignore'
`psm_ignore_patterns`	Project-wide definition of patterns to be ignored when processing model templates and creating cases. a pattern will be ignored when templates are replicated into new cases.	'.psm*', '.git*', '.svn*', '*~', 'default.parameters'
`log_level`	Log level to use for parsim loggers (use only 'info' or 'debug' for correct behavior).	'info'
`default_results_file`	Default name of file containing results, to collect with the `psm collect` command.	'results.json'
`case_prefix`	Prefix to use when constructing a directory name for case directories from a case name.	'case_'
`study_prefix`	Prefix to use when constructing a directory name for study directories from a study name.	'study_'

2.2 Model templates

A *model template* is a directory containing all files necessary to run a particular simulation. The input files and scripts may be parameterized by replacing values and pieces of text with parameter names. When a new case is created, the template directory will be replicated into a case directory, and all parameter names replaced by case-specific values.

To create a model template, you usually start by creating a working prototype case, including all scripts used for pre- and post-processing and running the simulation. Then you identify the settings and parameters you want to modify.

2.2.1 Where to store templates

It is practical to place all model templates you want to use in the *template root directory* of your project. When you create projects, the model template to use is then specified as a relative path to this directory.

The template root directory is defined when you initialize your project directory, but can be changed later. By default, the template root directory is a subdirectory named `modelTemplates` inside the project directory. In some situations, the template root directory could be in a central location separate from the project, for example if you share model templates with your colleagues.

When you create new cases, it is also possible to specify the model template as an absolute path.

2.2.2 Parametrization of files and scripts

Parsim uses the Python library pyexpander to process parameterized text files in a model template. The syntax for parameters is a valid Python variable name, enclosed in `$()`. For example, to introduce a parameter `DENSITY` in an input file, you would replace all occurencies of the nominal value by the string `$(DENSITY)`.

You must add the extension `.macro` to the name of all files containing parameters or macros; the extension is removed when a case is created and the file is processed for macro expansion.

The **pyexpander** library allows you to do very advanced operations in your input files, for example working with loops and conditionals; see the pyexpander documentation for details.

> **Warning:** The syntax for the pyexpander library can in principle be used to *redefine* the value of a model parameter inside a parameterized text files. **NEVER DO THIS**, as it breaks the link between the values you define when you created the case and the actual values in your input files!

2.2.3 Default parameters

It is very important that all parameters of a model have well-defined values. Each model template should therefor define default values for all parameters in the model template. The default values usually represent a well documented and validated reference case.

Default parameter values are defined in a *parameter file* named `default.parameters`, located in the root of the model template directory.

2.2.4 Ignoring files in templates

When you define model templates, you may have files in the template directory that you do not want copied or processed when you create cases. You can tell Parsim to ignore these files by specifying a matching ignore pattern in a file named `.psmignore`, placed in the same directory.

For example, you may keep detailed model documentation in a subdirectory `docs` in the model template, and you have som include files with extension `.inc`, which are included by macro expresssions in other files. To avoid having these files copied into every case you create, you could but these ignore patterns into a file `.psmignore` in the template directory:

```
docs
*.inc
```

The following patterns are ignored by default:

```
default.parameters
.psm*
.svn*
.git*
```

These standard patterns prevent copying of the default parameters file and version control system files (in case your model template is under version control).

2.3 Cases and Studies

Your simulations take place in Parsim *cases*. When a simulation case is created, a *model template* directory is recursively replicated to create a case directory. Parsim operations can also be carried out on a *study*, containing multiple cases. A study is a directory containing multiple case directories.

Parsim cases and studies are created using the commands **`psm case`** and **`psm study`**, respectively. With the `psm study` command, multiple cases are defined in a *caselist* file; see Section *Caselist files* below. Studies are also created by the `psm doe` command, which offers support for common *Design of Experiments (DOE)* methods like full factorial and central composite designs, or random sampling schemes like Monte Carlo or Latin Hypercube. The `psm dakota` command allows parsim to be used as a *front-end to the versatile Dakota library*; cases of a study are spawned dynamically by Dakota, based on the methods defined in a Dakota input file.

To run your simulation, you would use the command **`psm run`** to execute a script on the case, or on all cases of a study, see Section *Running scripts and executables*. All events and operations are logged in event logs for the case, study and/or project. This provides tracebaility and helps documentation of your simulation project.

When creating a case or a study, custom parameter values can be defined using several sources, which are listed here, in order of precedence:

1. Parameter definitions on the command-line (see the command-line reference, Sections *psm case* or *psm study*),
2. For studies only: Parameters defined case-by-case in a *caselist file* (see Section *Caselist files* below),
3. In a separate *parameter file*, which is named on the command-line (see Section *Parameter files* below),
4. In a default parameters file located in the *model template* directory (this file has the same format as other parameter files; see Section *Parameter files* below).

Parsim also defines a set of parameters containing parsim-related case information, as defined in the table *Automatically defined parameters with parsim-related case information* below. The names of these parameters all start with `PARSIM_`.

Table 1: Automatically defined parameters with parsim-related case information

Parameter	Description
`PARSIM_PROJECT_NAME`	Name of the parsim project.
`PARSIM_PROJECT_PATH`	Path to directory of the parsim project.
`PARSIM_TEMPLATE_PATH`	Path to template directory used to create the case.
`PARSIM_STUDY_NAME`	Name of Study, if any (otherwise empty string).
`PARSIM_CASE_NAME`	Name of Case.
`PARSIM_CASE_ID`	Case ID, which can be used as "target" with some parsim commands. Uses colon notation, for example `A:1` for a case "1" of a study named "A".
`PARSIM_VERSION`	Version of parsim used to create the case.

When you create cases and studies from model templates, the default values are often used for many of the model parameters, for example solver settings. The command-line option `--define` can be used to set custom values for a small number of parameters. Otherwise it is more practical to prepare a parameter file, especially if this parameter combination will be used several times. When you create a study with multiple cases, a *parameter file* is often used to define parameter values shared by all cases in the study. The *caselist file* is then used to define only the parameter values that vary between cases.

The properties and formats of *parameter files* and *caselist files* are described in the following sections.

2.3.1 Parameter files

Parameter files assign real values for parameters in your model. Paraeter files are used when you create individual cases. When creating studies, a parameter file is often used to define all parameters that have the same values for all cases. The *default parameter file* `default.parameters`, located in the root of every model template, is also written in the same format.

In its simplest form, a parameter file is a text file with two columns separated by white-space. The first column contains parameter names and the second column defines their values. Parameter values can be numbers or text strings. Strings should be enclosed in single or double quotes.

The complete syntax of parameter files are described by the following rules:

- Rows starting with characters "#" or ";" will be treated as comments and skipped.
- Parameter name and value columns can be separated by white-space and/or by a single colon ":".
- Values may be numbers or text strings. Strings should be enclosed in single or double quotes.
- If the value column (column two) is followed by one of the chracters "#" or ";", everything that follows is treated as a description of the parameter. This may be used in later Parsim versions, to provide help to the user of a model template.

The following is an example of a parameter file, where we have used all valid format options:

```
#------------------------------------------------------------------------
#  This is a sample parameter file (these comment lines are skipped...)
#------------------------------------------------------------------------
; This is also a comment line. They can start with both "#" and ";".
; The blank line below is also skipped...

# Geometry parameters
length = 12.0
width: 8
height      24      # [m] Height of our object.
# It's practical to describe parameters this way (see above)!
# Especially in a default parameter file, where all parameters
# of a model template occur.

# Strings are quoted:
color:  'blue'
```

In practice, you would use consistent formatting of your choice, to improve readability.

2.3.2 Caselist files

A caselist is mandatory when you create a study. The caselist has one row for each case in the study and it defines the parameter values that differ between cases in the study.

The first row is a header row, starting with the string `CASENAME` as the first field, followed by names of the parameter to define. Then follows one row for each case of the study. The first field defines the name of case, followed by values for the parameters in the header row. Fields in a row are white-space or comma-delimited. Extra white-space is ignored.

The following example is a valid caselist file:

```
# Comment lines are skipped. (They start with "#" or ";")
CASENAME  length    width    height  color
A1        12.0      3.1      2       'blue'
A2        12.0      3.1      4       'blue'
B1        8.22      3.1      2       'red'
B2        8.22      3.1      4       'red'
```

2.4 Running scripts and executables

The command `psm run` is used to run scripts or executables; for an individual case, or for all cases in a study.

The first positional argument to this command is an identifier of the target case or study. This could be a single case or a study. It could also be a single case within a study; if so, both study and case names are provided, separated by a colon ":". For example, `s2:c1` identifies the case "c1" of study "s2".

The second positional argument is the name of the script or executable to run. The remainder of the command line is forwarded as arguments to the script/executable. Unless the script or executable is given as an absolute path, it is looked for in the following locations (in order of precedence):

1. The current working directory (where `psm run` is executed),
2. The `bin` subdirectory of the project directory (if it exists),
3. The root of the case directory,
4. The `bin` subdirectory of the case directory (if it exists),
5. In a subdirectory of the case directory, as specified by the option `--sub_dir`.

The script or executable executes in the root of the case directory, or in a subdirectory specified by the `--sub_dir` option. The subprocess running the script or executable inherits the environment of the calling process (the terminal in which `psm run` was called), augmented with a set of environment variables with parsim-related case information. These extra variables correspond with the parameters in the table *Automatically defined parameters with parsim-related case information*.

Note: Parsim uses the `subprocess` python module to execute scripts and executables. Sometimes a script or executable will not run, unless the subprocess is started through a shell interpreter (corresponds to using the `shell` option of the `subprocess.Popen` constructor). This behavior can be forced with the `--shell` option of the `psm run` command.

2.5 Collecting results

Results from parsim studies may be collected conveniently in tabular format using the `psm collect` command. This assumes that the execution of a case produces a text file with output scalars in JSON format; see the *tutorial examples*.

CHAPTER

THREE

TUTORIAL

Your parameterized model templates are usually stored in the project directory. The default location is a subdirectory `modelTemplates` in the root of the project directory, but this can be customized by changing the project configuration settings. In some cases it is practical to store all model templates in a centralized location, separate from the project directory.

In this tutorial we will set up a very simple simulation project example, with one model template located inside the project directory.

For this tutorial, we assume that you work in Linux or Cygwin.

Parsim is a command-line tool. The main program is called **`psm`**. Use it with option **`-h`** to get information about available subcommands. You get detailed information about each subcommand using the command **`psm help <command>`**.

3.1 Creating a project

Let's call our project "myProject" and assume we want it in project directory `my_proj` in our home directory. We create the project directory and initialize it using the **`psm init`** command:

```
$ mkdir ~/my_proj
$ cd ~/my_proj
$ psm init myProject
INFO: Parsim project "myProject" successfully created
```

3.2 Creating a simple model template

Assume we have the following Python script, named `boxcalc.py`, which represents a very simple "simulation model":

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

from __future__ import print_function
import json

# Model parameters

#- Geometry
length = 12
width = 4
height = 1.5

#- Material properties
density = 1000 # kg/m3
color = 'black'
```

(continues on next page)

(continued from previous page)

```
# Calculations...

base_area = length * width
volume = base_area * height

mass = volume * density

# Print output (in json format)

output = {
             'base_area': base_area,
             'volume': volume,
             'mass': mass
         }

print('base_area =', base_area)
print('volume =', volume)
print('mass =', mass)

with open('output.json', 'w') as f:
    f.write(json.dumps(output))
    print('Successfully written results to output file "output.json"')
```

You can run this script through the Python interpreter:

```
$ python boxcalc.py
base_area = 48
volume = 72.0
mass = 72000.0
Successfully written results to output file "output.json"
```

If you run Linux, you can give the script executable permissions and run it directly:

```
$ ./boxcalc.py
base_area = 48
volume = 72.0
mass = 72000.0
Successfully written results to output file "output.json"
```

The script uses the `json` library to format the output dictionary in json format. The resulting output file `output.json` has the following content:

```
{"base_area": 48, "volume": 72.0, "mass": 72000.0}
```

This script has hard-coded parameter values. We can say that these values define a working reference case, where the results are known and "validated", in some sense. This is a good starting point for creating a parameterized model template, which we will call `box`.

By default, Parsim assumes that model templates are stored in a subdirectory `modelTemplates` in the project directory. Let us start by creating the model template directory, and copy our existing model files there. For your convenience, the `boxcalc.py` script can be found in the `demo` subdirectory of your Parsim installation; in our case, Parsim is installed in `$HOME/psm`. As we copy, we change the name of the script to `calc.py`. Inside the project directory,

```
$ mkdir modelTemplate
$ mkdir modelTemplate/box
$ copy $HOME/psm/demo/boxcalc.py modelTemplate/box/calc.py
```

Now we go to the template directory, and continue there:

```
cd modelTemplate/box
```

In a real application, the executable for running the simulation would usually read input data from another file, but in our example the input data is hard-coded in the script itself. This means that the script itself will be parameterized. Let us start by adding the extension `.macro` to the file name, so that Parsim will parse it for parameter substitution when you create cases from the template. We also set execute permissions on the script file (permission of the parameterized file will be inherited by the resulting script file, when cases are created):

```
$ mv calc.py calc.py.macro
$ chmod u+x calc.py.macro
```

The file contains the following numerical model parameters: `length`, `width`, `height` and `density`. It also contains the string parameter `color`. We note that the name of the output file is also hard-coded in the script file; while we're at it, we let this file name be a parameter, too.

We now create a default parameter file with the standard name `default.parameters`. We use the hard-coded values as the default values, as this represents our known and presumably validated reference case... The contents of `default.parameters` could then be:

```
#============================================================
# Model template "Box"
#
# Computes base area, volume and mass of a rectangular box
#============================================================
#------------------------
# Geometry
#------------------------
length:     12          # [m]
width:      4           # [m]
height:     1.5         # [m]
#------------------------
# Material properties
#------------------------
density:    1000        # [kg/m3] Density of the solid material
color:      'black'     # Color of the box
#------------------------
# Configuration
#------------------------
output_file: 'results.json'   # Holds scalar results, written as a json-format␣
↪dictionary
```

Note that we this file is in *parameter file* format, and that we have used comments to document the model template.

The next step is to substitute the hard-coded values in the script file with the parameters we defined:

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

from __future__ import print_function
import json

# Model parameters

#- Geometry
length = $(length)
width = $(width)
height = $(height)

#- Material properties
density = $(density) # kg/m3
color = '$(color)'

# Calculations...
```

(continues on next page)

(continued from previous page)

```
base_area = length * width
volume = base_area * height

mass = volume * density

# Print output (in json format)

output = {
            'base_area': base_area,
            'volume': volume,
            'mass': mass
         }

print('base_area =', base_area)
print('volume =', volume)
print('mass =', mass)

with open('$(output_file)', 'w') as f:
    f.write(json.dumps(output))
    print('Successfully written results to output file "%s"' % '$(output_file)')
```

Note that we put quotes around `$(output_file)`, as the parameter substitution returns the string without quotes.

That's it, our first model template is ready!

3.3 Creating a case and running the simulation

To create cases, we use the `psm case` command. See *psm case* for details. The following creates a case in a case directory `case_ref`, with the default parameters defined above:

```
$ psm case --template box ref
INFO: Found template in project template directory: C:\Users\Ola\PycharmProjects\
↪psm\doc\demo\modelTemplates\box
INFO: Parsim case "ref" successfully created
```

In this simple example, we can easily make a custom case with parameters modified on the command-line. For example, let's make a case "bigBox", with higher box and larger density:

```
$ psm case --template box --define height=20,density=1200 bigBox
INFO: Found template in project template directory: C:\Users\Ola\PycharmProjects\
↪psm\doc\demo\modelTemplates\box
INFO: Parsim case "bigBox" successfully created
```

Let's run the "simulation" of this larger box, using the **`psm run`** command:

```
$ psm run bigBox calc.py
INFO: Executing command/script/executable: calc.py
bigBox: Running executable...
INFO: Executable finished successfully (runtime: 0:00:00.059957)
   Executable : C:\Users\Ola\PycharmProjects\psm\doc\demo\case_bigBox\calc.py
   stdout     : C:\Users\Ola\PycharmProjects\psm\doc\demo\case_bigBox\calc.out
   stderr     : C:\Users\Ola\PycharmProjects\psm\doc\demo\case_bigBox\calc.err
```

As indicated by the console output, the standard output of the script is found in the file `calc.out` in the case directory:

3.4 Creating and running a study

It is equally simple to setup and operate on a whole parameter study, containing several cases, using the **psm study** command.

You would usually use a *parameter file* to define parameter values that are common to all cases and a *caselist file* to define the case names and the parameters that differ between cases. In our simple tutorial example, however, it is sufficient to use only a *caselist file*.

Let us assume that we want to create a study named `variants`, with case names and parameters defined in a caselist file named `variants_caselist`, as follows:

We now use the **psm study** command to create the study and all its cases:

```
$ psm study --template box --description "Variants in series A and B" --name
↪"variants" variants_caselist
INFO: Found template in project template directory: C:\Users\Ola\PycharmProjects\
↪psm\doc\demo\modelTemplates\box
INFO: Parsim study "variants" successfully created
INFO: Found template as absolute or relative path: C:\Users\Ola\PycharmProjects\
↪psm\doc\demo\modelTemplates\box
INFO: Parsim case "A1" successfully created
INFO: Found template as absolute or relative path: C:\Users\Ola\PycharmProjects\
↪psm\doc\demo\modelTemplates\box
INFO: Parsim case "A2" successfully created
INFO: Found template as absolute or relative path: C:\Users\Ola\PycharmProjects\
↪psm\doc\demo\modelTemplates\box
INFO: Parsim case "A3" successfully created
INFO: Found template as absolute or relative path: C:\Users\Ola\PycharmProjects\
↪psm\doc\demo\modelTemplates\box
INFO: Parsim case "B1" successfully created
INFO: Found template as absolute or relative path: C:\Users\Ola\PycharmProjects\
↪psm\doc\demo\modelTemplates\box
INFO: Parsim case "B2" successfully created
INFO: Found template as absolute or relative path: C:\Users\Ola\PycharmProjects\
↪psm\doc\demo\modelTemplates\box
INFO: Parsim case "B3" successfully created
```

This creates a study directory `study_variants` in the current directory; the study directory contains all the case directories. Note that we used option `--name` for naming the study and the option `--description` to provide useful information about its content.

We can run the simulation of all cases of the study with one single command:

```
$ psm run variants calc.py
INFO: Starts RUN operation on cases (executable: calc.py)...
INFO: Executing command/script/executable: calc.py
A1: Running executable...
INFO: Executable finished successfully (runtime: 0:00:00.070959)
   Executable : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_A1\
↪calc.py
   stdout     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_A1\
↪calc.out
   stderr     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_A1\
↪calc.err
INFO: Executing command/script/executable: calc.py
A2: Running executable...
INFO: Executable finished successfully (runtime: 0:00:00.058967)
   Executable : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_A2\
↪calc.py
   stdout     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_A2\
↪calc.out
   stderr     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_A2\
↪calc.err
```

(continues on next page)

(continued from previous page)

```
INFO: Executing command/script/executable: calc.py
A3: Running executable...
INFO: Executable finished successfully (runtime: 0:00:00.056948)
   Executable : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_A3\
↪calc.py
   stdout     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_A3\
↪calc.out
   stderr     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_A3\
↪calc.err
INFO: Executing command/script/executable: calc.py
B1: Running executable...
INFO: Executable finished successfully (runtime: 0:00:00.058966)
   Executable : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_B1\
↪calc.py
   stdout     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_B1\
↪calc.out
   stderr     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_B1\
↪calc.err
INFO: Executing command/script/executable: calc.py
B2: Running executable...
INFO: Executable finished successfully (runtime: 0:00:00.057967)
   Executable : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_B2\
↪calc.py
   stdout     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_B2\
↪calc.out
   stderr     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_B2\
↪calc.err
INFO: Executing command/script/executable: calc.py
B3: Running executable...
INFO: Executable finished successfully (runtime: 0:00:00.061950)
   Executable : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_B3\
↪calc.py
   stdout     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_B3\
↪calc.out
   stderr     : C:\Users\Ola\PycharmProjects\psm\doc\demo\study_variants\case_B3\
↪calc.err
INFO: Successfully finished RUN operation!
```

3.5 Collecting results from a Study

We can use the **`psm collect`** command to collect the output data from all cases and present it in one single table.

```
$ psm collect variants
INFO: Start collecting case results...
Case results files (input)   : ['results.json']
Study results file (output) : results.txt
INFO: Successfully finished collecting case results!
```

By default, results are read from a JSON formatted file `results.json` in the case directories. For the example here, a results file in a case would look something like this:

```
{"base_area": 40, "volume": 800, "mass": 720000}
```

The `--input` option can be used to specify a custom file path inside the case directory (a comma-separated list of multiple files is allowed). Unless a delimited format is requested (by defining a delimiter with the `--delim` option), the output is in tabular format, white fixed column spacing. The name of the output file is derived from the name of the (first) input file, unless specified explicitly with the `--output` option.) In this example, the output is written to the file `results.txt`, located in the study directory:

```
CASENAME  base_area  mass     volume  density  height  length
A1        40         576000   480     1200     12      10
A2        40         640000   640     1000     16      10
A3        40         720000   800     900      20      10
B1        60         864000   720     1200     12      15
B2        60         960000   960     1000     16      15
B3        60         1080000  1200    900      20      15
```

If the input file is missing or incomplete for one or more cases, this will be reported. Only the succesfully processed cases will be included in the output file.

Every time the collect command is run (e.g. collecting additional results from another simulation in the same Study), a tabular text file "study.results" inside the Study directory will be updated with the new data. This file will then contain all aggregated results for the study. All cases are reported in this file, even if data is missing. Missing data is reported as "NaN" in the table. Input parameters are not included in "study.results". These can instead be found separately in the file "study.caselist". The files "study.results" and "study.caselist" could be conveniently imported into pandas DataFrame objects for further processing. As an example, consider a study with 16 cases created from the "box" template. One simulation executable outputs results variables "basearea", "mass" and "volume" for all 16 cases. Another simulation outputs the result variable "d_eff" in another results file, but this simulation fails for several of the cases. The `psm collect` command is run separately after each of the simulations, to collect results. The file "study.results" would now look something like this:

```
CASENAME  base_area      volume           mass
       1  69.976002  118.945208  124886.521349
       2  49.999998   84.989997   89235.246931
       3  42.007998   71.405195   74971.884491
       4  30.016002   51.021200   53569.709150
       5  69.976002   90.982798   95527.388551
       6  49.999998   65.009997   68257.246770
       7  42.007998   54.618799   57347.008010
       8  30.016002   39.026806   40976.194750
       9  69.976002  118.945208  113003.895050
      10  49.999998   84.989997   80744.746270
      11  42.007998   71.405195   67838.505510
      12  30.016002   51.021200   48472.691250
      13  69.976002   90.982798   86438.207050
      14  49.999998   65.009997   61762.748029
      15  42.007998   54.618799   51890.589990
      16  30.016002   39.026806   37077.416851
```

Note that all output variables of both simulations are included in the file, but there is missing data, "NaN", in the "d_eff" column for some of the cases.

3.6 Object information and event logs

We can use the **`psm info`** to get information about the properties of a case, study or about the project as a whole. For example, let's look at the properties of the `bigBox` case:

```
$ psm info bigBox
Case ID              : bigBox
Creation date        : 2020-07-30 12:03:22
Description          :
Project name         : myProject
Study name           :
Template path        : C:\Users\Ola\PycharmProjects\psm\doc\demo\modelTemplates\box
Creation log file    : C:\Users\Ola\PycharmProjects\psm\doc\demo\case_bigBox\.psm\
↪create.log
Parsim version       : 2.0.0
Project path         : C:\Users\Ola\PycharmProjects\psm\doc\demo
```

(continues on next page)

(continued from previous page)

```
Study path          :
------------------------------------------------------
User parameters (command-line or parameter file)
------------------------------------------------------
density               : 1200
height                : 20
------------------------------------------------------
Default parameters (defined in template)
------------------------------------------------------
color                 : black
length                : 12
output_file           : results.json
width                 : 4
```

All Parsim objects (cases, studies and project) have an event log, where all events and operations are logged. The **`psm log`** command prints the event log to the console. For example, we look at the event log of the `bigBox` case:

```
$ psm log bigBox
2020-07-30 12:03:22 - INFO: Parsim case "bigBox" successfully created
2020-07-30 12:03:23 - INFO: Executing command/script/executable: calc.py
2020-07-30 12:03:23 - INFO: Executable finished successfully (runtime: 0:00:00.
↪059957)
   Executable : C:\Users\Ola\PycharmProjects\psm\doc\demo\case_bigBox\calc.py
   stdout     : C:\Users\Ola\PycharmProjects\psm\doc\demo\case_bigBox\calc.out
   stderr     : C:\Users\Ola\PycharmProjects\psm\doc\demo\case_bigBox\calc.err
```

CHAPTER

FOUR

PARAMETRIZATION OF FILES AND SCRIPTS

Parsim uses the Python library pyexpander to process parameterized text files in a model template. You must add the extension `.macro` to the name of all files containing parameters or macros; the extension is removed when a case is created and the file is processed for macro expansion.

Each template file is processed, "expanded", individually. The parsing of a file takes place in a Python session, where all the model parameters exist as variables in the global namespace. Template files usually consist mostly of pure text, which is simply echoed as-is to the corresponding case file.

The **pyexpander** library allows you to do very advanced operations in your input files, for example inserting Python code blocks for calculating complex constructs, working with loops and conditionals, include other files, etc. Some of these possibilities are explained below; see the pyexpander documentation for additional details. Note that some of the text below is copied from the official pyexpander documentation.

4.1 The meaning of the dollar sign

Almost all elements of the pyexpander language start with a dollar "$" sign. If a dollar is preceded by a backslash "\" it is escaped. The "\$" is then replaced with a simple dollar character "$" and the rules described further down do not apply.

Here is an example:

```
an escaped dollar: \$
```

This would produce this output:

```
an escaped dollar: $
```

4.2 Parameter substitution

The syntax for parameter substitution is a valid parameter name enclosed in `$()`. For example, to introduce a parameter `DENSITY` in an input file, you would replace all occurencies of the nominal value by the string `$(DENSITY)`. What is inside the brackets must be a valid Python expression. Note that a python expression, in opposition to a python statement, always has a value. This value is converted to a string and this string is inserted in the text in place of the substitution command.

The use of python expressions for parameter substitutions allows you to do more advanced substitutions. For example, you could compute and insert the mass, by multiplying parameters for density and volume: `$(DENSITY*VOLUME)`.

4.3 Comments

A comment is started by a sequence "$#" where the dollar sign is not preceded by a backslash (see above). All characters until and including the end of line character(s) are ignored. Here is an example:

```
This is ordinary text, $# from here it is a comment
here the text continues.
```

4.4 Commands

If the dollar sign, which is not preceded by a backslash, is followed by a letter or an underline "_" and one or more alphanumeric characters, including the underline "_", it is interpreted to be an expander command.

The *name* of the command consists of all alphanumeric characters including "_" that follow. In order to be able to embed commands into a sequence of letters, as a variant of this, the *name* may be enclosed in curly brackets. This variant is only allowed for commands that do not expect parameters.

If the command expects parameters, an opening round bracket "(" must immediately (without spaces) follow the characters of the command name. The parameters end with a closing round bracket ")".

Here are some examples:

```
this is not a command due to escaping rules: \$mycommand
a command: $begin
a command within a sequence of letters abc${begin}def
a command with parameters: $for(x in range(0,3))
```

Note that in the last line, since the parameter of the "for" command must be a valid python expression, all opening brackets in that expression must match a closing bracket. By this rule pyexpander is able to find the closing bracket that belongs to the opening bracket of the parameter list.

4.5 Executing python statements and code blocks

A statement may be any valid python code. Statements usually do not return values. All expressions are statements, but not all statements are expressions.

In order to execute python statements, there is the "py" command. "py" is an abbreviation of python. This command expects that valid python code follows enclosed in brackets. Note that the closing bracket for "py" *must not* be in the same line with a python comment, since a python comment would include the bracket and all characters until the end of the line, leading to a pyexpander parser error.

The "py" command leads to the execution of the python code but produces no output. It is usually used to define variables, but it can also be used to execute python code of more complexity. Here are some examples:

```
Here we define the variable "x" to be 1: $py(x=1)
Here we define two variables at a time: $py(x=1;y=2)
Here we define a function, note that we have to keep
the indentation that python requires intact:
$py(
def multiply(x,y):
    return x*y
    # here is a python comment
    # note that the closing bracket below
    # *MUST NOT* be in such a comment line
   )
```

Warning: The syntax for the pyexpander library can in principle be used to *redefine* the value of a model parameter inside a parameterized text files. **NEVER DO THIS**, as it breaks the link between the values you define when you created the case and the actual values in your input files!

4.5.1 Importing modules in code blocks

You can import modules inside code blocks. Any imported modules and packages will then be available for use also in substitutions and other code blocks later in the same template file.

In order to allow import of your own modules, parsim prepends some additional directories to the Python search path (PYTHONPATH):

- `bin` subdirectory of the project directory,
- the template directory,
- the `bin` subdirectory of the template directory.

4.5.2 Working with files inside code blocks

Before a template file is processed, the current directory is set to the corresponding destination directory in the case. This makes it possible to write files during template processing, and have these available in the case directory. This can be very useful, for example when debugging or generating data files for post-processing and reporting.

See the following example, where we have used the standard `math` module and `matplotlib` to generate a plot to illustrate the effect of the relevant case parameters, while processing an inputfile for a solver.

```
ghfgf
fghfghf
fgh
fg
fghf
g
```

4.6 Line continuation

Since the end of line character is never part of a command, commands placed on a single line would produce an empty line in the output. Since this is sometimes not wanted, the generation of an empty line can be suppressed by ending the line with a single backslash "\". Here is an example:

```
$py(x=1;y=2)\
The value of x is $(x), the value of y is $(y).
Note that no leading empty line is generated in this example.
```

4.7 Conditionals

A conditional part consists at least of an "if" and an "endif" command. Between these two there may be an arbitrary number of "elif" commands. Before "endif" and after the last "elif" (if present) there may be an "else" command. "if" and "elif" are followed by a condition expression, enclosed in round brackets. "else" and "endif" do not have parameters. If the condition after "if" is true, this part is evaluated. If it is false, the next "elif" part is tested. If it is true, this part is evaluated, if not, the next "elif" part is tested and so on. If no matching condition was found, the "else" part is evaluated.

All of this is oriented on the python language which also has "if","elif" and "else". "endif" has no counterpart in python since there the indentation shows where the block ends.

Here is an example:

```
We set x to 1; $py(x=1)
$if(x>2)
x is bigger than 2
$elif(x>1)
x is bigger than 1
$elif(x==1)
x is equal to 1
$else
x is smaller than 1
$endif
here is a classical if-else-endif:
$if(x>0)
x is bigger than 0
$else
x is not bigger than 0
$endif
here is a simple if-endif:
$if(x==0)
x is zero
$endif
```

4.8 While loops

While loops are used to generate text that contains almost identical repetitions of text fragments. The loop continues while the given loop condition is true. A while loop starts with a "while" command followed by a boolean expression enclosed in brackets. The end of the loop is marked by a "endwhile" statement.

Here is an example:

```
$py(a=3)
$while(a>0)
a is now: $(a)
$py(a-=1)
$endwhile
```

In this example the loop runs 3 times with values of a ranging from 3 to 1.

The command "while_begin" combines a while loop with a scope; see the pyexpander documentation.

4.9 For loops

For loops are a powerful tool to generate text that contains almost identical repetitions of text fragments. A "for" command expects a parameter that is a python expression in the form "variable(s) in iterable". For each run the variable is set to another value from the iterable and the following text is evaluated until "endfor" is found. At "endfor", pyexpander jumps back to the "for" statement and assigns the next value to the variable.

Here is an example:

```
$for(x in range(0,5))
x is now: $(x)
$endfor
```

The range function in python generates a list of integers starting with 0 and ending with 4 in this example.

You can also have more than one loop variable:

```
$for( (x,y) in [(x,x*x) for x in range(0,3)])
x:$(x) y:$(y)
$endfor
```

or you can iterate over keys and values of a python dictionary:

```
$py(d={"A":1, "B":2, "C":3})
$for( (k,v) in d.items())
key: $(k) value: $(v)
$endfor
```

The command "for_begin" combines a for loop with a scope; see the pyexpander documentation.

4.10 Include files

The "include" command is used to include a file at the current position. It must be followed by a string expression enclosed in brackets. The given file is then interpreted until the end of the file is reached, then the interpretation of the text continues after the "include" command in the original text.

Unless the file is given with an absolute path, it will be looked for first in the current directory of the template (same location as the template file being processed), then in the root of the template directory.

Here is an example:

```
$include("additional_defines.inc")
```

The command "include_begin" combines an include with a scope. It is equivalent to the case when the include file starts with a "begin" command and ends with an "end" command. See the pyexpander documentation for a discussion on scopes.

CHAPTER

FIVE

DESIGN OF EXPERIMENTS (DOE)

Parsim integrates functionality for common Design Of Experiments (DOE) methods, both random sampling schemes and well-known factorial designs.

The implementation of all methods (except Monte Carlo sampling) rely on the pyDOE Python library; for details, see the pyDOE documentation.

The psm doe command is used to create a study, based on the specified DOE scheme and parameter definitions in a parameter file. Some parameters are typically given fixed values in the parameter file, as for an ordinary study. The difference from an ordinary study is that the parameter file also defines a statistical distribution for each "uncertain", or varying, parameter. The distribution specifications must follow the syntax of distribution in the scipy.stats module. The cases created inside the study will then be sampled or mapped from these distributions according to the chosen DOE scheme.

All schemes implemented in the pyDOE2 package (and possibly others) will eventually be made accessible, but currently only the following schemes can be used:

- Monte Carlo random sampling (MC)
- Latin Hypercube Sampling (LHS)
- Plackett-Burman (fraction factorial designs)
- Two-level full factorial design
- General full factorial sampling (for more than two levels)
- Two-level fractional factorial sampling
- Central Composite Design (CCD)
- Generalized Subset Design (GSD)

To get help on using the `psm doe` command, you use the `-h` option, as usual:

```
psm doe -h
```

This also gives a list of the currently implemented DOE schemes. You can get detailed help on syntax and capabilities for each of these by adding the name of the scheme as argument to the `-h` option. For example, to get help on using the Plackett-Burman scheme,

```
psm doe -h pb
```

5.1 Example: Tutorial "box" model with full factorial design

We use the simplistic "box" model from the tutorial, assuming we want to investigate a full factorial design in the parameters `length`, `width`, `height` and `density`.

The lower and upper levels to use will be obtained by sampling the statistical distributions defined for the variability of the parameters. These distributions are defined in a parsim parameter file, and must match distribution classes defined in the `scipy.stats` library. In this example we assume uniform distributions centred around the nominal values in the default parameters file. The `scipy.stats.uniform` class takes the lower bound and the width as arguments. The parameter file `box_uniform.par` looks like this:

```
length:     uniform(10, 4)     # 12 [m]
width:      uniform(3, 2)       # 4 [m]
height:     uniform(1.3, 0.4)   # 1.5 [m]
density:    uniform(950, 100) # 1000 [kg/m3]
```

Running `psm doe -h` tells you that the two-level full factorial scheme is called `ff2n`. Let's check for additional options:

```
$ psm doe -h ff2n

Two-level full factorial sampling.

Keyword arguments:
    mapping (str): Selection of method for mapping factor levels to actual values␣
↪in the distribution.

                'int': (default) Use confidence interval with equal areas around␣
↪the mean. Width of interval is given by parameter 'beta' (default: 0.9545)
    beta (float): Width of confidence interval for 'int' method (see above).␣
↪Default: 0.9545 (+/- 2*sigma)
```

Considering all our distributions are uniform, it would be natural to use the lower and upper bound as our two test levels. This, however, would not work for a normal distribution, or any other unbounded distribution. The default method (and the only one currently implemented) for mapping levels to a given distribution is `int`, which means that we use a confidence interval of the distribution to define the lower and upper levels. The `beta` argument defines the width of this interval. If we set `beta` to 0.999, for example, then 0.01 % of the PDF is outside of this interval; half of it below the lower level, half above the upper level.

We create a study named "box_ff2n" for a two-level full factorial design:

This design creates 16 cases in the study. The actual simulations can now be run as in the tutorial:

Collect is also done as before:

CHAPTER

SIX

DAKOTA INTERFACE

Parsim can be used as an interface to the Dakota toolkit, developed by Sandia National Laboratories. For projects where you already have developed parameterized models with Parsim, this gives easy access to the most complete collection of methods and tools for optimization, uncertainty quantification and parameter estimation available.

What you need is a Dakota input file, which specifies which method to use and which variables to vary. You also need to add a couple of lines to your simulation script, so that it outputs the response variables that Dakota wants back. You then use the `psm dakota` command to have Parsim create an empty Study and start Dakota. Dakota uses a special analysis driver to create the cases needed for the analysis, based on your existing Parsim model template.

Dakota must be installed according to instructions. The Dakota executable should be in the executable path of your OS environment, so that the single command `dakota` will start the program. To check your Dakota installation, you can run Dakota to output version information:

```
> dakota -v
Dakota version 6.5 released Nov 11 2016.
Repository revision f928f89 (2016-11-10) built Nov 11 2016 05:09:45.
```

6.1 The Dakota input file

The Dakota input file contains a specification of the method to use, the model parameters to modify, how to execute the simulation and what output to expect. You need to study the Dakota user documentation and tutorials to learn how to use the functionality provided.

In the "variables" section of the input file, you need to make sure the variable "descriptors" match parameter names in your Parsim model template.

In the "interface" section of the Dakota input file, you need to tell Dakota how to create a new case and execute the simulation. This section must have exactly this content:

```
interface
  analysis_driver = 'psm_dakota_driver'
    fork
    parameters_file = 'params.in'
    results_file = 'results.out'
```

Here `psm_dakota_driver` is a special script executable installed with Parsim.

6.2 Output from the simulation

Your simulation script needs to be modified so that it writes the response variables in the proper format to an output file in the case directory, usually `results.out`. The name of the output file is specified by the "results_file" entry in the interface section of the Dakota input file (see above).

6.3 Restarts

The Parsim interface supports the Dakota restart functionality. Dakota writes a binary restart file with results from the function evaluations. Parsim saves restart files in the Study directory, for successive Dakota restarts. The initial Dakota run will have a run index of 0, and restarts will be numbered from 1, 2, etc. Parsim also saves copies of the Dakota inputfile used at each run, tagged by the same run index.

The `psm dakota` command has an option `--restart` to request restart and specify the index of the restart file to use. The index number itself is optional; if no index is given, the last restart file will be used for the restart. You will still need to specify Dakota input file and simulation executable, as these may have changed.

Dakota has an option `-stop_restart` to specify how many saves records to read from the restart file. The `psm dakota` command has a corresponding option `--stop_restart`.

To understand how the Dakota restart functionality works, please consult the Dakota documentation.

6.4 Dakota execution phases and pre-run

Dakota has three execution phases: pre-run, run and post-run. Some Dakota methods are implemented so that the pre-run phase can be run separately, which means that a table of case specifications will be generated without actually launching a simulation executable. This functionality is typically supported for sampling, parameter study and DACE methods.

The `psm dakota` command has an option `--pre_run` to use Dakota in pre-run mode. If supported by the selected method, this will create the corresponding Parsim cases of the study. The user can then use this as any other study, running simulation scripts and other activities using the `psm run` command, collecting results with the `psm collect` command, etc.

6.5 Example: Rosenbrock, gradient optimization

As an example, let us look at the Rosenbrock problem of the Dakota user documenation, where the gradient optimization method is used to find the minimum of the Rosenbrock function.

Assume we have the following Python script, which computes the Rosenbrock function for a fixed point:

```
"""
Compute Rosenbrock function.
"""
from __future__ import print_function

def rb(x1, x2):
    return 100*(x2 - x1**2)**2 + (1 - x1)**2

# ----------------------------------
# Input data
# ----------------------------------

x1 = 1.5
x2 = 0.5
```

(continues on next page)

(continued from previous page)

```
# ----------------------------------
# Calculation/simulation
# ----------------------------------

f = rb(x1, x2)

# ----------------------------------
# Results output
# ----------------------------------

print('x1', 'x2', 'f')
print(x1, x2, f)
```

We assume you already have Parsim project to work with, otherwise create one as explained in the tutorial.

6.5.1 Creating the model template

In the `modelTemplates` directory of your project, create a new model template called "rosenbrock":

```
> cd modelTemplates
> mkdir rosenbrock
```

Inside the rosenbrock directory, create a parameterized version of the Python script above, and name it `rb.py.macro`:

```
"""
Compute Rosenbrock function.
"""
from __future__ import print_function

import json

def rb(x1, x2):
    return 100*(x2 - x1**2)**2 + (1 - x1)**2

#----------------------------------
# Input data
#----------------------------------

x1 = $(x1)
x2 = $(x2)

#----------------------------------
# Calculation/simulation
#----------------------------------

f = rb(x1, x2)

#----------------------------------
# Results output
#----------------------------------

#print('x1', 'x2', 'f')
#print(x1, x2, f)

with open('results.out', 'w') as fout:
    fout.write('%s\n' % str(f))

with open('results.json', 'w') as fjson:
    fjson.write(json.dumps({'f': f}))
```

The only changes we have made is to introduce parameters `x1` and `x2` for the input data, and to write the computed function value to the output file `results.out`. We also chose to output the results in json format, as discussed in the Parsim tutorial, in case we would want to collect the results also in tabular format with the `psm collect` command.

The model template must also have a `default.parameters` file, which defines default values for the parameters:

```
#============================================================
# Model template "rosenbrock"
#
# Computes Rosenbrock problem.
# Outputs function value to 'results.out'
#============================================================
x1  :   1.0
x2  :   0.5
```

6.5.2 Modifying the Dakota input file

Compared to the original example in the Dakota manual, the only change needed in the input file `rosen_grad_opt.in` is the specification of the analysis driver in the interface section:

```
# Dakota Input File: rosen_grad_opt.in
# Usage:
#   dakota -i rosen_grad_opt.in -o rosen_grad_opt.out > rosen_grad_opt.stdout

environment
  # graphics
  tabular_data
    tabular_data_file = 'rosen_grad_opt.dat'

method
  max_iterations = 100
  convergence_tolerance = 1e-4
  conmin_frcg

model
  single

variables
  continuous_design = 2
    initial_point    -1.2      1.0
    lower_bounds     -2.0     -2.0
    upper_bounds      2.0      2.0
    descriptors       'x1'     "x2"

interface
  analysis_driver = 'psm_dakota_driver'
    fork
    parameters_file = 'params.in'
    results_file = 'results.out'

responses
  objective_functions = 1
#  analytic_gradients
  numerical_gradients
    method_source dakota
    interval_type forward
    fd_gradient_step_size = 1.e-5
  no_hessians
```

We here need to use the special executable `psm_dakota_driver`.

6.5.3 Running Dakota with Parsim

To run the Dakota optimization, with our new Parsim model template, use the `psm dakota` command:

```
psm dakota --template rosenbrock --name rb1 rosen_grad_opt.in rb.py
```

The first positional argument is the Dakota input file, the second is the name of the simulation executable, in this case the simple parameterized Python script above.

The output from Dakota is found inside the study directory `study_rb1`. The standard output from Dakota (the execution history) is found in `dakota.out`. The Dakota input file also instructed Dakota to write tabular data to the file `rosen_grad_opt.dat`.

6.5.4 Restarting a failed Dakota run

The example above generates 134 cases in the study. Assume that the process stops and crashes after, say, 20 succesful cases (for example because of a full disk, or something else). We would then want to restart the Dakota run, but making use of the existing 20 succesful function evaluations. The Dakota restart functionality makes this possible. In this example, you restart the Dakota execution with the command

```
psm dakota -t rosenbrock --name rb1 --restart 0 --stop_restart 20 rosen_grad_opt.
↪in rb.py
```

We here explicitly selected the initial restart file (0), although this is the same as the last one generated in this example. Parsim stores succesive restart files in the study directory, numbered by an integer run index, 0 corresponding to the original run. We also explicitly told Dakota to only use the first 20 function evaluations of the restart file; by default it would use as many as it would find.

Note that we define the name of the Dakota input file and the simulation executable for the restart. This is because one may want to modify these, to avoid the problems experienced in the previous run.

6.6 Example: Generating cases with the Dakota pre-run functionality

For the optimization problem above, Dakota must select parameter values for each new case based on the result of the previous cases. For other methods, for example random sampling methods or traditional response surface designs, parameter values for all cases can be produced before starting any simulations. This is possible with the Dakota pre-run functionality.

Let us assume we want to create a complete study with 200 cases, based on the Dakota random sampling method, using the Dakota input file `rosen_sampling.in`:

```
# Dakota Input File: rosen_sampling.in
# Usage:
#   dakota -i rosen_sampling.in -o rosen_sampling.out > rosen_sampling.stdout

environment
  # graphics
  tabular_data
    tabular_data_file = 'rosen_sampling.dat'

method
  sampling
    sample_type random
```

(continues on next page)

(continued from previous page)

```
    samples = 200
    seed = 17
    response_levels = 100.0

model
  single

variables
  uniform_uncertain = 2
    lower_bounds       -2.0  -2.0
    upper_bounds        2.0   2.0
    descriptors         'x1'  'x2'

interface
  analysis_driver = 'psm_dakota_driver'
    fork
    parameters_file = 'params.in'
    results_file = 'results.out'

responses
  response_functions = 1
  no_gradients
  no_hessians
```

This file is essentially the same as in the Dakota documentation. For consistency, we have modified the interface section to use the Parsim simulation driver, although the driver will never be executed in pre-run mode.

The Parsim study `rb2` can now be created in Dakota pre-run mode,

```
psm dakota -t rosenbrock --name rb2 --pre_run rosen_sampling.in rb.py
```

With the command-line syntax currently implemented, we have to provide a name of a simulation executable, although it is not used.

Once the study and its cases are created, you interact with it as with any other Parsim study. For exemple, you would run the actual Rosenbrock "simulation" for all cases with the `psm run` command,

```
psm run rb2 rb.py
```

Since the `rb.py` script above also outputs the response in the json file `results.json`, we can use the `psm collect` command to collect all results into a table:

```
psm collect -i results.json rb2
```

The default, the results table was written in space-separated format to the file `results.txt` in the study direcory.

6.7 Example: Polynomial Chaos Expansion on the Rosenbrock problem

As an additional example, we apply the Polynomial Chaos Expansion method (PCE) on the Rosenbrock function. This example is taken from Section 5.4.1.1 in the Dakota User's Manual; the interested reader should read about these methods there, to fully appreciate what is going on.

Again, we modify the interface section of the input file found in the Dakota documentation:

```
# Dakota Input File: rosen_uq_pce.in

environment
```

(continues on next page)

(continued from previous page)

```
  #graphics

method
  polynomial_chaos
    quadrature_order   = 5
    dimension_preference = 5 3
    samples_on_emulator = 10000
    seed = 12347 rng rnum2
    response_levels = .1 1. 50. 100. 500. 1000.
    variance_based_decomp #interaction_order = 1

variables
  uniform_uncertain = 2
    lower_bounds      = -2.  -2.
    upper_bounds      =  2.   2.
    descriptors       = 'x1' 'x2'

interface
  analysis_driver = 'psm_dakota_driver'
    fork
    parameters_file = 'params.in'
    results_file = 'results.out'

responses
  response_functions = 1
  no_gradients
  no_hessians
```

We then run Dakota through Parsim, as before,

```
psm dakota --template rosenbrock --name pce rosen_uq_pce.in rb.py
```

The results output by Dakota are found in the file `dakota.out` in the study directory `study_pce`.

CHAPTER

SEVEN

COMMAND-LINE REFERENCE

Parsim is a command-line tool. The command-line client program is called **psm**.

The general syntax is

```
psm [-h] SUBCOMMAND [<arguments>]
```

The available subcommands are described in the following sections.

Use option *-h* to get information about available subcommands. You get detailed information about each subcommand using the command **psm help <subcommand>**, or **psm <subcommand> -h**.

-h, --help
Show help message, including available subcommands, and exit.

SUBCOMMAND [<arguments>]
Parsim subcommand (see below), with arguments.

7.1 Project initialization and configuration

Before Parsim can be used with your project, your project directory must be *initialized* using the *psm init* command. Most of the configurations settings may be changed later using the *psm config* command.

7.1.1 psm init

Create a new project called NAME in the current directory. This stores some metadata and configuration settings for the project. There are sensible defaults for all settings, and most of them can be changed later using the psm config command.

```
usage: psm init [-h] [--description DESCRIPTION]
                [--default_executable EXECUTABLE] [--template_root DIRECTORY]
                [--default_template DIRECTORY]
                [--default_parameter_file FILENAME]
                [--log_level {debug,info,warning,error}]
                [--config PARAMETER=VALUE]
                NAME
```

Positional Arguments

NAME a short name for the project; should not contain spaces.

Named Arguments

--description, -m Project description (text in quotes)

--default_executable Name/path of default executable for the "run" command

--template_root Path to directory where model templates are stored

--default_template Name of default model template to replicate (a directory). A relative path is searched for in the current and in the template root directory. Could also be an absolute path.

--default_parameter_file Name of parameter file holding default parameter definitions. The default name is "default.parameters", located in the model template directory.

--log_level Possible choices: debug, info, warning, error

Set log-level for logging. The default is "info". This value is stored in the project settings.

--config, -c Definition of project configuration name=value pairs. Could be a comma-separated list of name/value pairs. (Explicit config options (above) have precedence.)

Note: When configuration settings containing strings are defined using the `--config` option, the whole option argument string must be enclosed in quotes. The string value itself must also be quoted. For example, the following option will correctly set the value of the `default_executable` setting:

```
psm init --config 'default_executable="myScript.sh"' myProject
```

7.1.2 psm config

Modify the settings for the current project.

```
usage: psm config [-h] [--description DESCRIPTION]
                  [--default_executable EXECUTABLE]
                  [--template_root DIRECTORY] [--default_template DIRECTORY]
                  [--default_parameter_file FILENAME]
                  [--log_level {debug,info,warning,error}]
                  [--config PARAMETER=VALUE]
```

Named Arguments

--description, -m Project description (text in quotes)

--default_executable Name/path of default executable for the "run" command

--template_root Path to directory where model templates are stored

--default_template Name of default model template to replicate (a directory). A relative path is searched for in the current and in the template root directory. Could also be an absolute path.

--default_parameter_file Name of parameter file holding default parameter definitions. The default name is "default.parameters", located in the model template directory.

--log_level Possible choices: debug, info, warning, error

Set log-level for logging. The default is "info". This value is stored in the project settings.

--config, -c Definition of project configuration name=value pairs. Could be a comma-separated list of name/value pairs. (Explicit config options (above) have precedence.)

7.2 Creation of cases and studies

Individual cases can be created using the *psm case* command, while entire case studies are created using the *psm study* command.

Case studies can also be created with the *psm doe* command. The DOE (Design Of Experiments) functionality uses built-in sampling algorithms to generate the parameter matrix, rather than reading it from a caselist.

7.2.1 psm case

Create a new case, named from CASE_ID.

A case is created from a model template. A template is a directory, whose content will be replicated recursively to form a new case. During replication, all files with extension ".macro" are assumed to be parameterized. These files are processed for macro expansion, in which parameter names are replaced by actual values. The processed files have the same name, but without the ".macro" extension. Parameter values can be specified on the command line directly (with the –define option), or in a PARAMETER_FILE. A model template usually defines default variables for all its parameters.

```
usage: psm case [-h] [--description DESCRIPTION] [--template TEMPLATE]
                [--parameters PARAMETER_FILE] [--define PARAMETER=VALUE]
                CASE_ID
```

Positional Arguments

CASE_ID Name used to form the case directory name.

Named Arguments

--description, -m Description of case or study (text in quotes)

--template, -t Name of the model template. This is either an absolute directory path, a path relative to the current directory, or a directory inside the template root directory of the parsim project.

--parameters, -p Name of a parameter file (absolute path, or relative to the current directory)

--define, -D Definition of parameter name=value pairs. Could be a comma-separated list of name/value pairs. Overrides values in a parameter file.

Note: When string values are defined for parameters using the `--define` option, the whole option argument string must be enclosed in quotes. The string value itself must also be quoted. For example, when creating a case `A13`, the following option will correctly set the value of the `output_file` parameter to `out.txt`:

```
psm case --template box --define 'output_file="out.txt"' A13
```

7.2.2 psm study

Create cases in a case study, as defined by the records in the CASELIST file.

A case is created from a model template. A template is a directory, whose content will be replicated recursively to form a new case. During replication, all files with extension ".macro" are assumed to be parameterized. These files are processed for macro expansion, in which parameter names are replaced by actual values. The processed files have the same name, but without the ".macro" extension. Parameter values can be specified on the command line directly (with the –define option), or in a PARAMETER_FILE. A model template usually defines default variables for all its parameters.

When a case study is created, case-specific values for the parameters are given in a CASELIST_FILE. Values in the caselist override values given on the command line.

```
usage: psm study [-h] [--description DESCRIPTION] [--template TEMPLATE]
                 [--parameters PARAMETER_FILE] [--define PARAMETER=VALUE]
                 [--name STUDY_NAME]
                 CASELIST
```

Positional Arguments

CASELIST Name of the caselist file, which defines case-specific parameter values.

Named Arguments

--description, -m Description of case or study (text in quotes)

--template, -t Name of the model template. This is either an absolute directory path, a path relative to the current directory, or a directory inside the template root directory of the parsim project.

--parameters, -p Name of a parameter file (absolute path, or relative to the current directory)

--define, -D Definition of parameter name=value pairs. Could be a comma-separated list of name/value pairs. Overrides values in a parameter file.

--name, -n Name used to form the study directory name. By default, the study directory is named from the CASELIST file name.

7.2.3 psm doe

Create cases in a case study, based on a sampling scheme and parameter definitions.

A case is created from a model template. A template is a directory, whose content will be replicated recursively to form a new case. During replication, all files with extension ".macro" are assumed to be parameterized. These files are processed for macro expansion, in which parameter names are replaced by actual values. The processed files have the same name, but without the ".macro" extension. Parameter values can be specified on the command line directly (with the –define option), or in a PARAMETER_FILE. A model template usually defines default variables for all its parameters.

When a DOE (Design Of Experiments) study is created, a sampling scheme is used to generate the caselist for the study. Parameters are defined in a parameter file, where statistical distributions are specified for the uncertain variables. Other parameters are given constant values.

```
usage: psm doe [-h] [--description DESCRIPTION] [--template TEMPLATE]
               [--define PARAMETER=VALUE] [--name STUDY_NAME]
               PARAMETER_FILE DOE_SCHEME [DOE_ARGS [DOE_ARGS ...]]
```

Positional Arguments

PARAMETER_FILE Name of the parameter file (absolute path, or relative to the current directory) defining constant parameter values, as well as statistical distributions for uncertain parameters.

DOE_SCHEME Valid DOE sampling scheme, for example "mc" (Monte Carlo) or "lhs" (Latin Hypercube Sampling.

DOE_ARGS Valid arguments for the chosen DOE sampling scheme

Named Arguments

--description, -m Description of case or study (text in quotes)

--template, -t Name of the model template. This is either an absolute directory path, a path relative to the current directory, or a directory inside the template root directory of the parsim project.

--define, -D Definition of parameter name=value pairs. Could be a comma-separated list of name/value pairs. Overrides values in a parameter file.

--name, -n Name used to form the study directory name. By default, the study directory is named from the PARAMETER_FILE file name.

To get help on an individual sampling scheme, search help for the doe command and add name of scheme. For example:

```
psm help doe <scheme>
```

or

```
psm doe -h <scheme>
```

The following table shows the currently implemented DOE schemes.

Table 1: Available DOE schemes.

mc	Monte Carlo random sampling.
ff2n	Two-level full factorial sampling.
fullfact	General full factorial sampling (for more than two levels).
fracfact	Two-level fractional factorial sampling.
ccdesign	Central Composite Design (CCD).
lhs	Latin Hypercube sampling.
pb	Plackett-Burman (pbdesign).
gsd	Generalized Subset Design (GSD)

7.2.4 psm dakota

Create a study and start a Dakota process in it.

A case is created from a model template. A template is a directory, whose content will be replicated recursively to form a new case. During replication, all files with extension ".macro" are assumed to be parameterized. These files are processed for macro expansion, in which parameter names are replaced by actual values. The processed files have the same name, but without the ".macro" extension. Parameter values can be specified on the command line directly (with the –define option), or in a PARAMETER_FILE. A model template usually defines default variables for all its parameters.

When a Dakota study is created, an empty study is created and control is then handed over to Dakota. Dakota uses a built-in Parsim analysis driver interface to create, run and evaluate successive cases needed for the Dakota simulation task. The Dakota simulation task is configured in the Dakota input file supplied on the command line.

```
usage: psm dakota [-h] [--description DESCRIPTION] [--template TEMPLATE]
                  [--parameters PARAMETER_FILE] [--define PARAMETER=VALUE]
                  [--name STUDY_NAME] [--restart [INDEX]] [--stop_restart N]
                  [--pre_run]
                  DAKOTA_INPUT EXECUTABLE
```

Positional Arguments

DAKOTA_INPUT Path to Dakota input file.

EXECUTABLE Name of executable, or path to it (named executable will be searched for)

Named Arguments

--description, -m Description of case or study (text in quotes)

--template, -t Name of the model template. This is either an absolute directory path, a path relative to the current directory, or a directory inside the template root directory of the parsim project.

--parameters, -p Name of a parameter file (absolute path, or relative to the current directory)

--define, -D Definition of parameter name=value pairs. Could be a comma-separated list of name/value pairs. Overrides values in a parameter file.

--name, -n Name used to form the study directory name. By default, the study directory is named from the DAKOTA_INPUT file name.

--restart Perform a restart of failed Dakota run. An extra integer index argument specifies which of successive restart files to use. "0" selects the orginal run; the default is to use the latest restart file created.

--stop_restart Integer -stop_restart option sent to Dakota if restart requested.

Default: 0

--pre_run Only create study with cases according to Dakota method, using the Dakota -pre_run option. No script execution.

Default: False

7.3 Operations on cases and studies

The following subcommands perform specific operations on a specified *target*, which may be a either a case or a study. In particular, the *psm run* command is used to execute scripts (or other executables) on a specific case, or on all cases of a study.

The target can be either a case or a study. Based on the name provided, the current directory is searched for a matching case or study directory. In case both a case and study is found, an error is issued. If this happens, please specify the full name of the directory instead (starting with either `case_` or `study_`).

It is possible to specify an individual case belonging to a study, by specifying both study and case names, separated by a colon. For example, the following will print detailed information about the case `A1`, which is part of the study `test_matrix`:

```
psm info test_matrix:A1
```

Some commands, like *psm info* and *psm comment*, can operate also on the project as a whole. If no `TARGET` argument is specified, the current project is used as the target.

7.3.1 psm run

Execute script or executable in specified case, or in each case of a study.

```
usage: psm run [-h] [--sub_dir SUBDIR] [-o OUTFILE] [-e ERRFILE] [--shell]
               TARGET EXECUTABLE ...
```

Positional Arguments

TARGET	Name of target (case or a study)
EXECUTABLE	Name of executable, or path to it (named executable will be searched for)
...	Additional arguments forwarded to the executable command-line

Named Arguments

--sub_dir, -d	Case subdirectory to run in (otherwise the case root)
-o	Name of file for redirection of stdout
-e	Name of file for redirection of stderr
--shell	Run executable through a shell (see Python subprocess module) Default: False

7.3.2 psm collect

Collect results (json format) from all cases of a study and create text table.

```
usage: psm collect [-h] [--input INFILE] [--output OUTFILE] [--delim CHAR]
                   STUDY
```

Positional Arguments

STUDY	Name of study

Named Arguments

--input, -i	Name of case file(s) containing results in json format. Multiple comma-delimited file names allowed. (Default: "results.json")
--output, -o	Name of output file (default derived from name of input file)
--delim	Field delimeter (default is white-space with fixed column width)

7.3.3 psm info

Print information about the specified target (case or study), or current project (no target).

```
usage: psm info [-h] [TARGET]
```

Positional Arguments

TARGET Name of target (case or a study). If missing, the current project is assumed.

7.3.4 psm log

Print event log of specified target (case or study), or current project (no target).

```
usage: psm log [-h] [TARGET]
```

Positional Arguments

TARGET Name of target (case or a study). If missing, the current project is assumed.

7.3.5 psm comment

Write user comment to the event log of the specified target (case or study), or current project (no target).

```
usage: psm comment [-h] [TARGET] comment
```

Positional Arguments

TARGET Name of target (case or a study). If missing, the current project is assumed.

comment user comment to add to the event log

CHAPTER

EIGHT

SOURCE CODE DOCUMENTATION

8.1 parsim.core module

Main parsim classes and utility functions.

class parsim.core.**Case**(*name=None*, *path=None*, ***kwargs*)

Bases: *parsim.core.ParsimObject*

Parsim Case class.

A *Case* instance holds information about a case, created from a model template.

A *Case* is usually identified by its name, and the `path` to its storage on disk will then be constructed from the `name` argument.

The constructor extends the baseclass constructor.

Parameters

- **project** (`Project`) – Reference to parent *Project* instance.
- **name** (*str*) – Name of the case.
- **study** (`Study`) – Reference to parent *Study* instance, if any (otherwise None).

project

Reference to parent *Project* instance.

Type *Project*

study

Reference to parent *Study* instance (if any).

Type *Study*

collect(*input=None*, *parameters=None*)

Collect results from the case.

Parameters

- **input** (*list*) – List of names of results files to process (could also be string of single file name).
- **parameters** (*list*) – List of input parameter columns to include in output table.

Returns Tuple containing dict of name and value of result variables (output_dict) and dict of name and value of input parameters to include in output table (param_dict).

Return type (dict, dict)

create(*description=None*, *template=None*, *user_parameters=None*, *caselist_parameters=None*, *default_parameters=None*)

Create a new *Case* object.

Parameters

- **description** (*str*) – Optional description of case.
- **template** (*str*) – Name of (or path to) model template used for Cases of the Study.
- **user_parameters** (*dict*) – Dict of parameters common to all cases.
- **caselist_parameters** (*dict*) – Dict of parameters read from Study caselist.
- **default_parameters** (*dict*) – Dict of default parameters (constructed here, if not provided by parent *Study*).

get_env()
Create and return dict with parsim-related information about the case.

These key-value pairs can be sent along as extra parameters on case creation, or injected into "run" subprocess as environment variables.

Returns parsim-related case information

Return type dict

info(*create_log=False*)
Show some basic information about the case.

load()
Loads object data from an object data file on disk.

The object must, of course, already exist on disk.

property parameters
Getter returning pandas DataFrame containing value and source of all parameters.

property results
Getter returning pandas Series containing all results collected for the case

run(*executable*, *args=''*, *sub_dir=None*, *out=None*, *err=None*, *shell=False*)
Run `executable` on the case.

Parameters

- **executable** (*str*) – Name or path of executable to run.
- **args** (*list*) – List of string arguments to executable.
- **sub_dir** (*str*) – Relative path to Case subdirectory in which to run the executable (default is to run the in the root of the Case directory).
- **out** (*str*) – Optional custom name of output file.
- **err** (*str*) – Optional custom name of error file.
- **shell** (*bool*) – Flag used by the subprocess call (whether to run executable in OS shell, or directly).

parsim.core.**DEFAULT_PROJECT_CONFIG = {'case_prefix': 'case_', 'dakota_exe': 'dakota', 'c**
Default config settings for Parsim projects.

exception parsim.core.**ParsimCaseError**(**args*, ***kwargs*)
Bases: *parsim.core.ParsimError*

Error in operation on case, which may affect only this particular case.

Raising this exception, rather than *ParsimError*, would allow an outer exception handler to continue. For example, the *Study.collect()* method would like to process all cases of the study, even if the result file is missing for a particular case (raising *ParsimCaseError*). Other cases may still be ok!

exception parsim.core.**ParsimError**(**args*, ***kwargs*)
Bases: `Exception`

Baseclass for Parsim exceptions.

The keyword argument *handled* may be set, if the the error has already been handled, in the sense that information has been given to the user. This is to avoid duplicate output if the exception is captured and handled by outer handler.

Parameters **handled** (*bool*) – Set to true if exception already reported to user. Defaults to False.

handled
Flag to show if exception already reported to user.

Type bool

exception parsim.core.**ParsimExpanderError**(**args*, ***kwargs*)
Bases: *parsim.core.ParsimError*

Exception raised inside pyExpander library.

class parsim.core.**ParsimObject**(*name=None*, *path=None*, ***kwargs*)
Bases: object

Baseclass for all Parsim objects (projects, cases and studies).

A *ParsimObject* is uniquely identified by its path on disk. Unless the path is given with the *path* argument, a path can be constructed from an object name.

If the object exists on disk, its data will be automatically loaded from the object data file (with the *load* method). Otherwise an empty object is created by the constructor, but its data will have to be initialized by a separate call to the *create* method.

Parameters

- **path** (*str*) – Path to object directory. Mandatory, unless the *path* attribute has already been set by the subclass constructor before calling the baseclass constructor.
- **name** (*str*) – Name of object. Used for name value in *data* dict attribute, if given.
- **registry** (*list*) – List of names of attributes to store on disk (by default, only the *data* and *registry* attributes are stored.

path
Path to object directory.

Type str

_parent_object_path
Path to parent object (*Study* or *Project*).

Type str

_parent_object_type
Type of parent object

Type str

_psm_path
Path to object storage subdirectory (inside object directory).

Type str

_data_file
Name of data file for object data storage.

Type str

_logger
Logger instance of this object.

Type logging.Logger

_logger_name
Name of object logger instance.

Type str

_logger_file
File used by logger file handler.

Type str

exists
Flag is True is object exists on disk.

Type bool

open
Flag is True if logger file is open and/or (?) object data not saved.

Type bool

registry
List of object attributes to save to and load from disk storage.

Type list

data
Dict for storage of object data.

Type dict

add_comment(*msg*)
Add user comment to object event log.

Parameters **msg** (*str*) – Comment text string for event log.

close()
Closes the object.

Closing the object means its data is written to the object data file on disk and the object logger is closed.

create(*name=None*, *description=None*, *noSave=False*, *onlySave=False*)
Create new object.

ParsimObject creation is made separate from the object instantiation. The object is complete only when this creation step is done, and this is when the object is written to disk.

Parameters

- **name** (*str*) – Sets `name` value in dict attribute *data*, if provided.
- **description** (*str*) – Optional description of the object.

delete(*force=False*, *logging=True*)
Delete the object's directory from disk, including all its contents.

Parameters

- **force** (*bool*) – Forces deletion without interactive query, also if the object actually exists and seems to correctly created. Default is to ask for confirmation if the the object's `exist` attribute is True.
- **logging** (*bool*) – Argument currently not used...

Returns Returns True if delete is successful. Returns False if the object's directory path does not exist, or if an error occurred.

Return type bool

find_parent_object_path(*p=None*, *project=False*)
Find path to an object's parent object, if any.

Most `ParsimObjects` live in the directory of a parent object. For example, a Study lives in a Project directory, and a Case lives either in a Project directory, or it is part of a Study.

This function will move back up through the directory tree, until a parent *ParsimObject* is found. The `project` argument can be set if we want to find the Parsim Project.

Parameters

- **p** (*str*) – Optional path to the child object, whose parent we seek. By default, the present object is the child.
- **project** – If True, the function will look for the Project to which the object belongs.

Returns If a parent is found, returns tuple of *path* of parent object and its `type` attribute. If not no parent is found, returns `(None, None)`.

Return type (str, str)

get(*key*)
Get a value from the object data attribute dictionary.

Parameters **key** (*str*) – Dictionary key.

Returns Dictionary value.

info()
Create text with basic object information.

Returns Line-wrapped text with object information.

Return type str

load()
Loads object data from an object data file on disk.

The object must, of course, already exist on disk.

log()
Return contents of event log.

Returns Contents of logger file.

Return type str

save(*silent=False*)
Save the object to its object data file on disk.

The object attributes to save are those named in the object's *registry* attribute.

set(*key*, *value*)
Set a key-value pair in the object *data* attribute dictionary.

Parameters

- **key** (*str*) – Dictionary key.
- **value** – Dictionary value.

class parsim.core.**Project**(*name=None*, *path=None*, ***kwargs*)
Bases: *parsim.core.ParsimObject*

Parsim Project class.

A *Project* instance holds information about the Project and the project directory. The project directory is the root directory for all cases and studies of the project.

The constructor extends the baseclass constructor.

Parameters **path** (*str*) – Path to project directory. If not provided, the current directory is used.

config
Dictionary containing configuration settings for the project.

Type dict

create (*name=None*, *description=None*, ***config_dict*)
Create new *Project* object.

Extends baseclass *ParsimObject* method. Sets description, if given as keyword argument. The *config* dict attribute is initialized with values from *DEFAULT_PROJECT_CONFIG*, and then modified by remaining keyword arguments (in `config_dict`). Creates directories for template root and default template, if missing.

Parameters

- **name** (*str*) – Mandatory name of project.
- **description** (*str*) – Description of project
- ****config_dict** – Dict of keyword arguments.

delete (*force=False*, *logging=True*)
Delete the object's directory from disk, including all its contents.

Projects can only be deleted manually – ABORT...

Parameters

- **force** (*bool*) – Forces deletion without interactive query, also if the object actually exists and seems to correctly created. Default is to ask for confirmation if the the object's `exist` attribute is True.
- **logging** (*bool*) – Argument currently not used...

Returns Returns True if delete is successful. Returns False if the object's directory path does not exist, if the object is a `ParsimProject`, or if an error occurred.

Return type bool

find_target (*target_arg*)
Process a command-line target argument and generate *ParsimError* if invalid.

Parameters **target_arg** (*str*) – A string identifying a "target". This could be a single case or a study. It could also be a single case within a study; if so, both study and case names are provided, separated by a colon ":". For example, `s2:c1` identifies the case "c1" of study "s2".

Returns Reference to target *Study* or *Case*. Returns reference to *Project*, if `target_arg` is empty.

Return type *ParsimObject*

find_target_path_and_type (*name*)
Find an existing case or study directory path, based on given `name`.

The target could be either a case, or a study. Returns both path and type, if it finds a unique target.

Parameters **name** (*str*) – Name of target to look for, which could be a case or a study.

Returns If a unique target (case or study) is found, returns tuple (`path`, `type`) with both path and object type string ('case' or 'study'). If both a case and study exists with the given name, return `None` path and type 'both'. Returns `(None, None)` if no target is found.

Return type (str, str)

get_case_path (*case_id*, *study=None*, *only_existing=False*)
Construct path to case directory, based on a `case_id` (the name of the case).

Parameters

- **case_id** (*str*) – Case ID, i.e. the name of the case.
- **study** (*ParsimStudy*) – Reference to parent *Study* instance.
- **only_existing** (*bool*) – If set, return path only if the case exists.

Returns Path to case directory. Return None if `only_existing` argument is set and the case does not exist.

Return type str

get_study_path(*name*, *only_existing=False*)
Construct path to study directory, based on `name` of study.

Parameters

- **name** (*str*) – Name of the study.
- **only_existing** (*bool*) – If set, return path only if the study exists.

Returns Path to study directory. Return None if `only_existing` argument is set and the study does not exist.

Return type str

get_template_path(*template*)
Search several locations for a valid model template.

Several locations are searched, in the following order:

1. Current directory (only if `template` name is provided)
2. Project template directory
3. Default template, inside project template directory

Parameters **template** (*str*) – Name of model template

Returns Path to model template directory.

Return type str

info()
Show some basic information about the project.

load()
Loads object data from object data file on disk.

Extends baseclass *ParsimObject* method by setting a log level according to Project config settings.

modify(***config_dict*)
Modify project config settings.

Keyword arguments will update project *config* attribute.

Parameters ****config_dict** – Dict of keyword arguments.

class parsim.core.**Study**(*name=None*, *path=None*, ***kwargs*)
Bases: *parsim.core.ParsimObject*

Parsim Study class.

A *Study* instance holds information about a parameter study and contains one or more Cases. The *Study* has a dict of parameters common to all cases (can be represented as a Parsim parameter file). It will also have a caselist defining the parameter values that vary between cases of the Study.

A *Study* is usually identified by its name, and the `path` to its storage on disk will then be constructed from the `name` argument.

The constructor extends the baseclass constructor.

Parameters

- **project** (`Project`) – Reference to parent *Project* instance.
- **name** (*str*) – Name of the parameter study.

- **path** (*str*) – Path to the *Study* on disk. If absent (common), the path is constructed from the `name` argument.

project
Reference to parent *Project* instance.

Type *Project*

property caselist
Getter returning pandas DataFrame with the Study caselist (all varying parameters).

collect(***kwargs*)
Collect results from all cases of the study.

Iterates over all cases of the study and calls the *Case.collect* method of each.

Parameters ****kwargs** (*dict*) – Dict of keyword arguments to forward.

create(*description=None*, *template=None*, *user_parameters=None*, *caselist_file=None*, *doe_scheme=None*, *doe_args=None*, *distr_dict=None*)
Create a new *Study* object.

Extends the baseclass method. This method provides two ways to generate Cases for Study; either the path of a caselist file is provided with the `caselist_file` option, or a DOE scheme is defined with the `doe_scheme` option. These two options are mutually exclusive. If none of them are provided, an empty Study (without cases) will be created, and the information about model template and user parameters is stored. (An empty Study is used when Parsim is run as an interface to Dakota.)

If a DOE scheme is given, a dict with additional DOE options can be provided with the `doe_args` argument. The `distr_dict` argument should then be a dict defining statistical distributions for all uncertain parameters.

Parameters

- **description** (*str*) – Optional description of parameter study.
- **template** (*str*) – Name of (or path to) model template used for Cases of the Study.
- **user_parameters** (*dict*) – Dict of parameters common to all cases.
- **caselist_file** (*str*) – Path to caselist file with case and parameter definitions.
- **doe_scheme** (*str*) – Name of DOE scheme to use for case creation.
- **doe_args** (*dict*) – Dict of arguments to the DOE scheme.
- **distr_dict** (*dict*) – Dict of distributions for uncertain parameters.

info(*create_log=False*)
Show some basic information about the study.

next()
Iterator function (proxy for `__next__`, for Python 3 compatibility).

property parameters
Getter returning pandas DataFrame containing value and source of all parameters.

property results
Getter returning pandas DataFrame with the last collected results of the Study.

run(*executable*, ***kwargs*)
Run `executable` on all cases of the study.

Iterates over all cases of the study and calls the *Case.run* method of each.

Parameters

- **executable** (*str*) – Name or path of executable to run.
- ****kwargs** (*dict*) – Dict of keyword arguments to forward.

run_dakota(*dakota_file=None*, *executable=None*, *pre_run=False*, *restart=False*, *restart_index=-1*, *stop_restart=0*, *shell=False*)

Run Dakota in this Study.

Runs Dakota inside the study (already created with the `create` method). The Dakota input file...

Parameters

- **dakota_file** (*str*) –
- **executable** (*str*) –
- **pre_run** (*bool*) –
- **restart** –
- **restart_index** (*int*) –
- **stop_restart** (*int*) –
- **shell** (*bool*) –

parsim.core.**create_caselist_file**(*output_path*, *case_names*, *column_dict=None*, *case_dicts=None*, *parameter_names=None*, *csv=None*)

Create and write a case list to a specified file (`output_path`).

Case names are supplied in a separate list (`case_names`). Parameter names may optionally be supplied in a list (`parameter_names`), to control the order of the parameter columns, or to output only a subset of the available parameters; default is to output all available parameters, in arbitrary order.

Case parameter data are specified in one of two ways: Either `column_dict` is a dict from parameter name to list (vector) of values (one value per case), or `case_dicts` is a list of dicts from parameter name to value (one dict per case).

Parameters

- **output_path** (*str*) – Path to output file.
- **case_names** (*list*) – List of case names (output in first column, "CASENAME")
- **column_dict** (*dict*) – Dict from parameter name to list of values (one value per case).
- **case_dicts** (*list*) – List of dicts from parameter name to value (one dict per case). Arguments column_dict and case_dicts are mutually exclusive.
- **parameter_names** (`list`, optional) – List of parameter names to output.
- **csv** (`bool`, optional) – Flag to request output in CSV format.

  Default (False) is to output in fixed-width whitespace-separated columns. If `csv` is one of the characters ",;%|#", this will be used as separator. If `csv` is any other boolean "True" value, ";" is used as separator.

parsim.core.**create_parameter_file**(*output_path*, *parameters*, *parameter_names=None*)

Create and write a parameter file to a specified file (`output_path`).

Parameter definitions are specified as dictionary (`parameters`).

Parameter names may optionally be supplied in a list (`parameter_names`), to control the order of the parameters in the output, or to output only a subset of the available parameters; default is to output all available parameters, in arbitrary order.

Parameters

- **output_path** (*str*) – Path to output file.
- **parameters** (*dict*) – Dictionary with parameter/value pairs.
- **parameter_names** (*list*) – List of parameter names, to control order of parameters in output.

parsim.core.**find_file_path**(*filename*, **places*)
Search for filename in list of possible paths and return full path to the file.

Parameters

- **filename** (*str*) – Name of file to search for.
- ***places** (*str*) – Variable length argument list of paths to search.

Returns Full path to file.

Return type str

parsim.core.**parse_caselist_file**(*file_name*)
Parse a caselist file (or Dakota annotated file) and return case definitions.

Returns case definitions as a list of tuples. The first tuple element is the case name, and the second is a dictionary of parameters.

Parameters **file_name** (*str*) – Path to input file.

Returns Case definitions as a list of tuples. First tuple element is case name and the second a dict of parameters.

Return type list((str, dict))

parsim.core.**parse_parameter_file**(*file_path*, *get_comments=False*, *doe=False*)
Parse parameter file and return dict with parameters (and perhaps comments).

If called with the doe flag set, tries to interpret un-resolved parameter definitions as distributions.

Parameters

- **file_path** (*str*) – Path to parameter file.
- **get_comments** (*bool*) – If True, return also comments at end of lines.
- **doe** (*bool*) – If True, try to interpret un-resolved parameter definitions as distributions of uncertain parameters.

Returns: Returns dict of parameter values. If doe is True, return tuple with dict of parameter distributions as second item. If get_comments is True, returns tuple with dict of comments as last item.

8.2 parsim.dakota module

class parsim.dakota.**DakotaVariables**(*filepath*)
Bases: object

Class of Dakota variables file.

Parameters **filepath** (*str*) – Path to input file to read.

parsim.dakota.**key_value**(*line*, *type=None*)
Parse key-value pair from line read from Dakota input file.

Parameters

- **line** (*str*) – Text line to parse.
- **type** (*str*) – Optional string {'int', 'float', 'str'} to indicate expected value type.
- **attempts to do the right thing anyway.** (*Otherwise*) –

Returns Tuple of key and value (key is string, but value could be int, float or str).

Return type (str, value)

parsim.dakota.**psm_dakota_driver**()
Analysis driver for Dakota fork interface under parsim.

8.3 parsim.doe module

class parsim.doe.**FactorLevelScheme**(*distr_dict*, ***kwargs*)
Bases: *parsim.doe.SamplingScheme*

Subclass for sampling factors at certain levels.

log_message()
Return log message for the sampling operation.

sampling_method = 'levels'

class parsim.doe.**RandomSamplingScheme**(*distr_dict*, ***kwargs*)
Bases: *parsim.doe.SamplingScheme*

Subclass for random sampling of distributions through their Cumulative Distribution Function (CDF).

log_message()
Return log message for the sampling operation.

sampling_method = 'cdf'

class parsim.doe.**SamplingScheme**(*distr_dict*, ***kwargs*)
Bases: object

Baseclass for sampling schemes, used for creating parsim Study cbjects.

Parameters

- **distr_dict** (*dict*) – Dict of distribution definitions for varying parameters.
- **kwargs** (*dict*) – Dict of keyword arguments for sampling schemes.

add_required_args(*args*)
Add variable name to list of required arguments for the scheme.

Parameters **args** (*str*) – Name of argument.

add_valid_args(*args*)
Add variable name to list of valid arguments for the scheme.

Parameters **args** (*str*) – Name of argument.

get(**key*)
Get value from object dictionary.

Parameters ***key** (*list*) – List of arguments; first argument is key name, second holds optional default value.

Returns Value of dictionary value.

Return type value

get_case_definitions()
Output list of tuples (case_id, case_dict)

classmethod help_message()
Return help message for this sampler class, based on subclass docstring.

Docstring should describe sampling method and all options.

log_message()
Return log message for the sampling operation.

sampling_method = None

set(*key*, *value*)
Sets value in object dictionary.

Parameters

- **key** (*str*) – Name of variable.
- **value** – Value of variable.

write_caselist(*filename*)
Construct and write caselist to file. Not yet implemented.

Parameters **filename** (*str*) – Name of caselist to write.

write_norm_matrix(*filename*)
Write norm_matrix to file.

Parameters **filename** (*str*) – Name of output file.

write_value_matrix(*filename*)
Write value_matrix to file.

Parameters **filename** (*str*) – Name of output file.

class parsim.doe.**ccdesign**(*distr_dict*, ***kwargs*)
Bases: *parsim.doe.FactorLevelScheme*

Central Composite Design (CCD).

The "face" keyword can be used to select one of three combinations of facial and corner points: 'ccc' for circumscribed (corner points are at nominal high/low levels, while facial points are outside (!) of these); 'cci' for inscribed (facial points are at nominal high/low levels, while corner points are inside of these); 'ccf' for faced (both corner and facial points are at nominal high/low levels). Contrary to pyDOE standard behavior, the parsim default is inscribed, 'cci', such that parameter values are guaranteed not to exceed the nominal high/low levels. If a circumscribed ('ccc') is defined, then beware that the facial points may lie far outside of the nominal high/low levels – make sure these extreme values are realistic and physically meaningful! This effect can be controlled to some extend by choosing a smaller value of the beta parameter, so that the nominal high/low levels are taken away from the tails/bounds of the parameter distributions.

The "alpha" keyword is used to define designs that are either orthogonal (value 'o'), or rotatable (value 'r'). Default is orthogonal, 'o'. Note that both circumscribed and inscribed designs may be rotatable, but the faced design ('ccf') cannot.

The CCD method generates only one center point, since simulations are deterministic in nature. For regression analysis of the results, however, the center point should usually be replicated or weighted higher than the other points.

Keyword Arguments

- **mapping** (*str*) – Selection of method for mapping high/low factor levels to actual values in the distribution.

  'int': (default) Use confidence interval with equal areas around the mean. Width of interval is given by parameter 'beta' (default: 0.9545)
- **beta** (*float*) – Width of confidence interval for 'int' method (see above). Default: 0.9545 (+/- 2*sigma)
- **face** (*str*) – Specifies combination of corner and facial points: 'cci' for inscribed (default), 'ccc' for circumscribed or 'ccf' for faced design.
- **alpha** (*str*) – Selects whether design should be orthogonal ('o') or rotatable ('r'). Default is orthogonal.

parsim.doe.**docstring_first_line**(*obj*)
Extract text from first line of object's docstring.

Parameters **obj** (*object*) – Reference to object.

Returns Docstring text (first line).

Return type str

class parsim.doe.**ff2n**(*distr_dict*, ***kwargs*)
Bases: *parsim.doe.FactorLevelScheme*

Two-level full factorial sampling.

Keyword Arguments

- **mapping** (*str*) – Selection of method for mapping factor levels to actual values in the distribution.

  'int': (default) Use confidence interval with equal areas around the mean. Width of interval is given by parameter 'beta' (default: 0.9545)

- **beta** (*float*) – Width of confidence interval for 'int' method (see above). Default: 0.9545 (+/- 2*sigma)

class parsim.doe.**fracfact**(*distr_dict*, ***kwargs*)
Bases: *parsim.doe.FactorLevelScheme*

Two-level fractional factorial sampling.

A fractional factorial design reduces the number of runs, compared to a full factorial design, by confounding certain main factor effects with interaction effects. The confounding must be explicitly defined by the user, either by supplying a so-called *generator* expression with the "gen" keyword, or defining the *resolution* of the design with the "res" keyword (only one of these keywords should be used).

In addition, the resulting design may be folded (replicated with levels switched). The keyword "fold" specifies if folding should be applied, either an all columns (value "all"), or by listing which parameters to fold (comma-delimited list of parameter4 columns; first parameter is "1"). For example, "fold=1,3" to fold parameters one and three, or "fold=all" to fold the whole design.

Keyword Arguments

- **mapping** (*str*) – Selection of method for mapping high/low factor levels to actual values in the distribution.

  'int': (default) Use confidence interval with equal areas around the mean. Width of interval is given by parameter 'beta' (default: 0.9545)

- **beta** (*float*) – Width of confidence interval for 'int' method (see above). Default: 0.9545 (+/- 2*sigma)

- **gen** (*str*) – Generator pattern, for example gen='a,b,ab' for three parameters. Note that the generator expression must be given as a quoted string, with columns delimited by commas rather than white-space.

- **res** (*int*) – Resolution of the design, defined as an integer. Common values are 3, 4 or 5, depending on the number of parameters in the design.

- **fold** (*str*) – Specify optional folding of design. "all" to fold all columns, or a comma-delimited list of columns to fold (starting with 1 for first column). For example, fold='all', or fold=1,3,5.

class parsim.doe.**fullfact**(*distr_dict*, ***kwargs*)
Bases: *parsim.doe.FactorLevelScheme*

General full factorial sampling (for more than two levels).

The number of levels can be given individually for each parameters using the "levels" keyword argument. If a single integer is given, this is the number of levels used for all parameters.

Keyword Arguments

- **mapping** (*str*) – Selection of method for mapping high/low factor levels to actual values in the distribution.

  'int': (default) Use confidence interval with equal areas around the mean. Width of interval is given by parameter 'beta' (default: 0.9545)

- **beta** (*float*) – Width of confidence interval for 'int' method (see above). Default: 0.9545 (+/- 2*sigma)
- **levels** (*str*) – Comma-delimited list of number of levels for each parameter. If only one integer is given,
- **number of levels will be used for all parameters. Defaults to two levels, same as ff2n.** (*this*) –
- **Example** – "levels=2,2,3,3"

class parsim.doe.**gsd**(*distr_dict*, ***kwargs*)

Bases: *parsim.doe.fullfact*

Generalized Subset Design (GSD)

GSD is a generalization of traditional fractional factorial designs to problems, where factors can have more than two levels.

In many application problems, factors can have categorical or quantitative factors on more than two levels. Conventional reduced designs have not been able to deal with such types of problems. Full multi-level factorial designs can handle such problems, but the number of samples quickly grows too large to be useful.

The GSD method provides balanced designs in multi-level experiments, with the number of experiments reduced by a user-specified reduction factor. Complementary reduced designs are also provided, analogous to fold-over in traditional fractional factorial designs.

The number of levels can be given individually for each parameter using the "levels" keyword argument. If a single integer is given, this is the number of levels used for all parameters. The "reduction" keyword is used to specify the reduction factor of the design; the reduction factor must be an integer larger than one (default is 2). The number of complementary designs is controlled by the "ncomp" keyword (default is 1, meaning only the original reduced design is used).

Keyword Arguments

- **mapping** (*str*) – Selection of method for mapping high/low factor levels to actual values in the distribution.

  'int': (default) Use confidence interval with equal areas around the mean. Width of interval is given by parameter 'beta' (default: 0.9545)
- **beta** (*float*) – Width of confidence interval for 'int' method (see above). Default: 0.9545 (+/- 2*sigma)
- **levels** (*str*) – Comma-delimited list of number of levels for each parameter. If only one integer is given,
- **number of levels will be used for all parameters. Defaults to two levels, same as ff2n.** (*this*) –
- **Example** – "levels=2,2,3,3"
- **reduction** (*int*) – Size reduction factor, which must be an integer larger than one (default: 2)
- **ncomp** (*int*) – Number of complementary designs (default: 1)

parsim.doe.**help_message**(*sphinx=False*)

Create help message for DOE sampling, listing all available samplers in this module.

Documentation is available from sampler classes docstrings.

Parameters **sphinx** (*bool*) – Set to True if called from Sphinx run.

Returns Help message description created from docstring.

Return type str

class parsim.doe.**lhs**(*distr_dict*, ***kwargs*)
Bases: *parsim.doe.RandomSamplingScheme*

Latin Hypercube sampling.

Keyword Arguments

- **n** (*int*) – Number of sample points (default: one per parameter)
- **mode** (*str*) – String that tells lhs how to sample the points,

  'rand': Random within sampling interval (default),

  'center', 'c': Center within interval,

  'maximin', 'm': Maximize the minimum distance between points, but place the point in a randomized location within its interval,

  'centermaximin', 'cm': Same as 'maximin', but centered within the intervals,

  'correlation', 'corr': Minimize the maximum correlation coefficient.
- **iter** (*int*) – Number of iterations (used by some modes; see pyDOE docs and code).

valid_modes = ['rand', 'center', 'c', 'maximin', 'm', 'centermaximin', 'cm', 'correl

class parsim.doe.**mc**(*distr_dict*, ***kwargs*)
Bases: *parsim.doe.RandomSamplingScheme*

Monte Carlo random sampling.

Keyword Arguments **n** (*int*) – Number of samples (default: 10)

class parsim.doe.**pb**(*distr_dict*, ***kwargs*)
Bases: *parsim.doe.FactorLevelScheme*

Plackett-Burman (pbdesign).

Another way to generate fractional-factorial designs is through the use of Plackett-Burman designs. These designs are unique in that the number of trial conditions (rows) expands by multiples of four (e.g. 4, 8, 12, etc.). The max number of factors allowed before a design increases the number of rows is always one less than the next higher multiple of four.

Keyword Arguments

- **mapping** (*str*) – Selection of method for mapping factor levels to actual values in the distribution.

  'int': (default) Use confidence interval with equal areas around the mean. Width of interval is given by parameter 'beta' (default: 0.9545)
- **beta** (*float*) – Width of confidence interval for 'int' method (see above). Default: 0.9545 (+/- 2*sigma)

CHAPTER

NINE

ADVANCED INSTALLATION OPTIONS

See Section *Installation* for standard installation instructions.

9.1 Repository installation

The code base for Parsim is hosted at gitlab.com. First clone the git repository in an appropriate location:

```
git clone https://olwi@gitlab.com/olwi/psm.git psm
```

Even if you use `conda` as your package manager, it is easiest to use `pip` for installing an editable development version of parsim. However, do this in a special conda enviroment! Once you have used `pip` in the conda installation, it may be difficult for conda to update other packages without dependency conflicts.

In your development python environment, enter the working copy of the source repository and use `pip` to install parsim in editable development mode:

```
cd psm
pip install -e .
```

9.2 Updating an existing installation

If you upgrade and existing installation with `pip install --upgrade`, it is recommended to do it without upgrading dependencies, unless it is required. This is to avoid triggering an upgrade of NumPy and SciPy, which could break your installation.

Do the upgrade in two steps. First upgrade parsim, but not dependencies:

```
pip install --upgrade --no-deps parsim
```

Then update whatever requirements are not already fulfilled:

```
pip install parsim
```

The two-step process for upgrading a repository installation is analogous.

CHAPTER

TEN

GLOSSARY

Case

Cases

Case directory A case (or rather a case directory) is a replica of a model template, where model parameters have been replaced with actual values. The case is where model simulation, pre- and post-processing takes place.

Caselist

Caselist file

Caselist files A caselist is used to create multiple cases in a study. The caselist has one row defining each case of the study. The first column (with heading "CASENAME") of the caselist contains the name of case. Subsequent columns contain parameter values.

Default parameter values

Default parameter file It is very important that all parameters of a model have well-defined values. Each model template should therefor define default values for all parameters in the model template. This is done in the default parameter file. The default values usually represent a well documented and validated reference case.

Model template

Template directory A model template is a directory containing all files necessary to define a model and run a simulation of it. The files may be organized in subdirectories. A model template should define *default parameter values* for all parameters in the model.

Parameter file

Parameter files A file defining values for parameters. Each row defines one parameter. The parameter name is in the first column and the value is in the second. The columns can be separated by white-space, a colon ":", or a en equality sign "=". String values should be enclosed in quotes. A comment may follow the value, after a separator "#" or ";".

Project

Project directory When you use Parsim, your simulation project is represented by a project directory structure. The project directory must be initialized to use Parsim. This creates some basic configuration settings which apply to the whole project. Inside the project (directory), you create and operate on cases and studies.

Study

Studies

Study directory A study contains multiple cases, defined in a caselist. Operations, like running scripts, can be carried out on all cases of a study with one single command. Results from all cases of a study can be collected in a table for analysis and post-processing.

Template root directory A directory for storing all model templates used in your simulation project. By default, this is a subdirectory `modelTemplates` in the root of the project directory. It could also be a central location, for example if you share models with your colleagues. The template root directory is defined when you initialize your project directory, but can be changed later.

CHAPTER

ELEVEN

USING THE PARSIM API

parsim is intended to be used as a command-line tool. For some particular tasks, however, it can be practical to use the Python API in your own Python script, or interactivly. Post-processing and analysis of results is one such task. Here we show a few examples.

We assume that there is a parsim project in which we have have worked with the simplistic sample models discussed in the tutorial section. We make that project directory, here named `demo`, our current directory, before starting a Python interpreter.

```
$ cd demo
```

Before you continue, please have look at the tutorial, so that you are familiar with the simple model template `box` used there. We recall that the Python simulation script is called `calc.py`. For our examples, we will create and execute two small experimental designs (the output of the parsim command is not shown here). In both cases, the stochastic parameters are defined in a parameter file `box_uniform.par`:

```
length:     uniform(10, 4)      # 12 [m]
width:      uniform(3, 2)       # 4 [m]
height:     uniform(1.3, 0.4)   # 1.5 [m]
density:    uniform(950, 100)   # 1000 [kg/m3]
```

First, we create a two-level full factorial design called `box_ff2n`. With four varying parameters, this yields 16 cases.

```
$ psm doe -t box --name box_ff2n box_uniform.par ff2n beta=0.999
```

We execute the simulation script and collect results:

```
$ psm run box_ff2n calc.py
$ psm collect box_ff2n
```

Second, we try the Generalized Subset Design (GSD) available in the `pyDOE2` package. This scheme makes it possible to create reduced designs with more than two levels. Here we try three levels for `length` and `width`, and two levels for the other parameters. The size of the design is reduced by a factor two, which gives us 18 cases.

```
$ psm doe -t box --name box_gsd box_uniform.par gsd beta=0.999 levels=3,3,2,2␣
↪reduction=2
$ psm run box_gsd calc.py
$ psm collect box_gsd
```

Now start a Python interpreter of your choice, e.g. `ipython` or a Jupyter Notebook (this API tutorial is itself created as a Jupyter Notebook).

[1]:
```
%cd demo
```

```
C:\Users\olawid\PycharmProjects\psm\doc\demo
```

11.1 Loading parsim Case and Study objects

In the previous section, we created parameters studies `box_ff2n` and `box_gsd`. The parsim APi can be used to load the corresponding parsim objects into a Python session.

We start by importing the parsim API:

[2]:
```
import parsim.core as ps
```

Remeber that we started the Python interpreter in the project directory, which is therefor our current directory. Let's load the two studies from the example above. To do this we uspply the study name as an argument to the `Study` class constructor:

[3]:
```
s_ff2n = ps.Study('box_ff2n')
s_gsd = ps.Study('box_gsd')
```

We can also load individual cases with the `Case` class. To open an individual case, which is part of Study, we use the names of both Study and Case, separated by a colon:

[4]:
```
c_gsd_4 = ps.Case('box_gsd:4')
```

On the command-line, the `psm info` command can be used to output information about studies and cases. In a Python console, you can have the same information by printing the output of the object `info` method:

[5]:
```
print(s_ff2n.info())
```

```
Study name              : box_ff2n
Creation date           : 2019-08-29 16:55:58
Description             :
Project name            : myProject
Template path           : C:\Users\olawid\PycharmProjects\psm\doc\demo\
↪modelTemplates\box
Parsim version          : 1.0.dev
Project path            : C:\Users\olawid\PycharmProjects\psm\doc\demo
Caselist/DOE params     : ['length', 'width', 'height', 'density']
DOE scheme              : ff2n
DOE arguments           : {'beta': 0.999}
------------------------------------------------------
Variable parameter distributions (DOE)
------------------------------------------------------
length                  : uniform(10, 4)
width                   : uniform(3, 2)
height                  : uniform(1.3, 0.4)
density                 : uniform(950, 100)
------------------------------------------------------
Default parameters (defined in template)
------------------------------------------------------
output_file             : results.json
color                   : black
------------------------------------------------------------------------------
 Case#  Case_ID           Description
------------------------------------------------------------------------------
 1      1
 2      2
 3      3
 4      4
 5      5
 6      6
 7      7
 8      8
 9      9
 10     10
```

(continues on next page)

(continued from previous page)

```
 11      11
 12      12
 13      13
 14      14
 15      15
 16      16
-----------------------------------------------------------------------
```

In a similar manner you can show the log for a case or study. To look at the logged history of the 4th GSD case, for example,

[6]:
```
print(c_gsd_4.log())
```

```
2019-08-29 16:56:49 - INFO: Parsim case "4" successfully created
2019-08-29 16:56:52 - INFO: Executing command/script/executable: calc.py
2019-08-29 16:56:52 - INFO: Executable finished successfully (runtime: 0:00:00.
↪155078)
   Executable : C:\Users\olawid\PycharmProjects\psm\doc\demo\study_box_gsd\case_4\
↪calc.py
   stdout     : C:\Users\olawid\PycharmProjects\psm\doc\demo\study_box_gsd\case_4\
↪calc.out
   stderr     : C:\Users\olawid\PycharmProjects\psm\doc\demo\study_box_gsd\case_4\
↪calc.err
```

11.2 Working with input parameters and results

In the following example, we will show how we can use the parsim API and the functionality of the `pandas` library. We will also show how we can make a simple linear regression model using the `statsmodels` library

We start by importing the popular packages `pandas` and `statsmodels`. We also import the parsim API from `parsim.core`.

[7]:
```
import pandas as pd
import statsmodels.api as sm

import parsim.core as ps
```

We load the full-factorial study, as before:

[8]:
```
s_ff2n = ps.Study('box_ff2n')
```

11.2.1 Parameters and results as pandas DataFrame and Series objects

Both `Study` and `Case` classes provide data in the form of pandas objects. For a study, the `caselist` attribute is a `DataFrame` with the values of all varying parameters. Similarily, all collected results of the study are aggregated in `results`:

[9]:
```
s_ff2n.caselist
```

[9]:

CASENAME	length	width	height	density
1	13.998	4.999	1.6998	1049.95
2	10.002	4.999	1.6998	1049.95
3	13.998	3.001	1.6998	1049.95
4	10.002	3.001	1.6998	1049.95
5	13.998	4.999	1.3002	1049.95

(continues on next page)

(continued from previous page)

```
6          10.002  4.999  1.3002  1049.95
7          13.998  3.001  1.3002  1049.95
8          10.002  3.001  1.3002  1049.95
9          13.998  4.999  1.6998   950.05
10         10.002  4.999  1.6998   950.05
11         13.998  3.001  1.6998   950.05
12         10.002  3.001  1.6998   950.05
13         13.998  4.999  1.3002   950.05
14         10.002  4.999  1.3002   950.05
15         13.998  3.001  1.3002   950.05
16         10.002  3.001  1.3002   950.05
```

[10]:
```
s_ff2n.results
```

[10]:
```
           base_area      volume           mass
CASENAME
1          69.976002  118.945208  124886.521349
2          49.999998   84.989997   89235.246931
3          42.007998   71.405195   74971.884491
4          30.016002   51.021200   53569.709150
5          69.976002   90.982798   95527.388551
6          49.999998   65.009997   68257.246770
7          42.007998   54.618799   57347.008010
8          30.016002   39.026806   40976.194750
9          69.976002  118.945208  113003.895050
10         49.999998   84.989997   80744.746270
11         42.007998   71.405195   67838.505510
12         30.016002   51.021200   48472.691250
13         69.976002   90.982798   86438.207050
14         49.999998   65.009997   61762.748029
15         42.007998   54.618799   51890.589990
16         30.016002   39.026806   37077.416851
```

The `results` DataFrame may contain missing values, `NaN`, if the case simulation crashed or failed to produce a result.

The `parameter` attribute shows the values and sources of the parameters that all cases of the study have in common. In the present example, all constant parameters have their default values.

[11]:
```
s_ff2n.parameters
```

[11]:
```
                    value   source
color               black  default
output_file  results.json  default
```

Case objects have the attributes `parameters` and `results`, but for obvious reasons not the `caselist`.

[12]:
```
c_ff2n_4 = ps.Case('box_ff2n:4')

c_ff2n_4.parameters
```

[12]:
```
                    value    source
length             10.002  caselist
width               3.001  caselist
height             1.6998  caselist
density           1049.95  caselist
color               black   default
output_file  results.json   default
```

Note that it's easy to see if a parameter is set by the caselist, on the user commandline, or if it's a default value.

`Case.parameters` is a pandas DataFrame, while `Case.results` is pandas Series object,

[13]:
```
c_ff2n_4.results
```

[13]:
```
base_area       30.016002
volume          51.021200
mass         53569.709150
Name: results, dtype: float64
```

11.2.2 Merging datasets from several Studies

Sooner or later, you will be in a situation where you will want to make a joint analysis of the results from more than one Study. Maybe you started out with a small study, which only allows for a simple linear model, for example a factorial design in two levels, such as the `box_ff2n` study avobe. Then you realize that a one or two parameters should rather be varied on at least three levels, in order to capture also quadratic terms. For example the `box_gsd` study se saw earlier, where both `length` and `width` parameters were varied on three levels. From a statistical point of view, this is a rather poor example, but that's another story... Let's go ahead and load the GSD study as well, and merge the two for analysis...

[14]:
```
s_gsd = ps.Study('box_gsd')
```

The results and the caselist from both studies can now be concatenated, but we need to decide what to do with the fact that the DataFrames from both studies have the same index entries. We could either ignore the original indices altogether, creating a new index in the process, or try to keep track of the identities of the cases also in the concatenated datasets. Here we chose the latter option, by creating a hierachical index indicating the names of the original studies. (The option `sort=False` is used to avoid sorting the columns in lexical order.)

[15]:
```
results = pd.concat([s_ff2n.results, s_gsd.results], keys=['ff2n', 'gsd'],␣
↪sort=False)
caselist = pd.concat([s_ff2n.caselist, s_gsd.caselist], keys=['ff2n', 'gsd'],␣
↪sort=False)
```

[16]:
```
caselist
```

[16]:

		length	width	height	density
	CASENAME				
ff2n	1	13.998	4.999	1.6998	1049.95
	2	10.002	4.999	1.6998	1049.95
	3	13.998	3.001	1.6998	1049.95
	4	10.002	3.001	1.6998	1049.95
	5	13.998	4.999	1.3002	1049.95
	6	10.002	4.999	1.3002	1049.95
	7	13.998	3.001	1.3002	1049.95
	8	10.002	3.001	1.3002	1049.95
	9	13.998	4.999	1.6998	950.05
	10	10.002	4.999	1.6998	950.05
	11	13.998	3.001	1.6998	950.05
	12	10.002	3.001	1.6998	950.05
	13	13.998	4.999	1.3002	950.05
	14	10.002	4.999	1.3002	950.05
	15	13.998	3.001	1.3002	950.05
	16	10.002	3.001	1.3002	950.05
gsd	1	13.998	4.999	1.6998	1049.95
	2	13.998	3.001	1.6998	1049.95
	3	10.002	4.999	1.6998	1049.95
	4	10.002	3.001	1.6998	1049.95
	5	13.998	4.999	1.3002	950.05
	6	13.998	3.001	1.3002	950.05
	7	10.002	4.999	1.3002	950.05
	8	10.002	3.001	1.3002	950.05
	9	13.998	4.000	1.6998	950.05
	10	10.002	4.000	1.6998	950.05

(continues on next page)

(continued from previous page)

```
     11         13.998  4.000  1.3002  1049.95
     12         10.002  4.000  1.3002  1049.95
     13         12.000  4.999  1.6998   950.05
     14         12.000  3.001  1.6998   950.05
     15         12.000  4.999  1.3002  1049.95
     16         12.000  3.001  1.3002  1049.95
     17         12.000  4.000  1.6998  1049.95
     18         12.000  4.000  1.3002   950.05
```

11.2.3 Linear regression analysis

We will use the `statsmodels` package to fit the combined data to a simple linear model.

For numerical robustness, you are strongly adviced to normalize or standardize your data! We name our standardized datasets `y` and `x`, for dependent and independent variables, respectively.

[17]:
```
y = (results - results.mean())/results.std()
x = (caselist - caselist.mean())/caselist.std()
```

Fitting a model `statsmodels`typically involves three steps:

1. Use the model class to describe the model,
2. Fit the model
3. Inspect the results

Here we use the `OLS` class, for ordinary least squares.

First we define and fit the model to our data; we use the output variable `volume` for this example:

[20]:
```
mod = sm.OLS(y['volume'], x)   # define model
res = mod.fit()                # fit model to data
```

Now let's look at the results:

[21]:
```
res.summary()
```

[21]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                 volume   R-squared:                       0.973
Model:                            OLS   Adj. R-squared:                  0.969
Method:                 Least Squares   F-statistic:                     269.5
Date:                Thu, 26 Sep 2019   Prob (F-statistic):           4.81e-23
Time:                        16:53:39   Log-Likelihood:                 13.618
No. Observations:                  34   AIC:                            -19.24
Df Residuals:                      30   BIC:                            -13.13
Df Model:                           4
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
length         0.4915      0.030     16.361      0.000       0.430       0.553
width          0.7373      0.030     24.541      0.000       0.676       0.799
height         0.4333      0.030     14.398      0.000       0.372       0.495
density     -1.11e-16      0.030  -3.69e-15      1.000      -0.061       0.061
==============================================================================
Omnibus:                        6.096   Durbin-Watson:                   1.613
Prob(Omnibus):                  0.047   Jarque-Bera (JB):                5.828
Skew:                           1.004   Prob(JB):                       0.0543
```

(continues on next page)

(continued from previous page)

```
Kurtosis:                       2.713   Cond. No.                         1.06
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly␣
↪specified.
"""
```

The middle section of this summary output shows the regression coefficients, their standrad error and some statistics (t and F values) to help us judge how important the parameters are for predicting the output quantity. Since we standardized both parameters and result quantities, the standard error is the same for all independent variables. It should come as no surprise that `density` does not contribute to the predicted `volume` (the coefficient is extremly low).

CHAPTER

TWELVE

CHANGES TO PARSIM

12.1 Changelog

All notable changes to this project will be documented in this file.

The format is based on Keep a Changelog, and this project adheres to Semantic Versioning.

12.1.1 [2.1.0] - 2020-08-03

Added

- Special parameters with parsim-related case information are available for macro expansion on case creation. They are also available as environent variables of the subprocesses started with the `psm run` command.

12.1.2 [2.0.0] - 2019-09-27

Added

- Implemented DOE scheme *`gsd`* (Generalized Subset Design), as available in pyDOE2 package. Allows reduced factorial designs with more than two levels.
- With DOE scheme *`fracfact`*, the user can now define the reduced design either by a generator expression (option "gen"), or by the design resolution (option "res").
- Property-based getter functions in *`Case`* and *`Study`* classes now provide caselist, results and parameter info as pandas `DataFrame` or `Series` objects.
- *`Study.collect()`* method now aggregates all collected results in a file "study.results" in the study directory. This file includes also cases with missing data (marked as "NaN", pandas-style).
- `pyDOE2`, `numpy`, `scipy` and pandas are now mandatory dependencies.
- Constructor of *`Case`* class now handles colon-separated format <study>:<case> for cases inside studies. This simplifies use of the parsim Python API for working with results.
- Examples of how to use the Python API for post-processing and data analysis.

Removed

- Support for Python 2 has been removed!
- Config option `paramlist_upper_names` has been removed (controlled optional automatic conversion of all parameter names to uppercase).

Changed

- API changes in ParsimObjects (incl. subclasses *`Project`*, *`Study`* and *`Case`*), especially in constructors (`__init__`) and *`create()`* method. These changes makes the CLI command implementations shorter and easier to understand. Better checking of sane `name` and *`path`* arguments to constructor.

12.1.3 [1.0.0] - 2018-12-19

Changed

- Use pyDOE2 package, instead of pyDOE.

Fixed

- Now works with Python 3 (and Python 2.7, as before).
- Fixed problems with parsing of DOE command-line arguments.

12.1.4 [0.7.0] - 2018-07-26

CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

C

D

E

F

G

H