sim_db Documentation

Håkon Austlid Taskén

May 07, 2020

CONTENTS

sim_db is a command line tool and a set of functions for conveniently running a large number of simulations with different parameter values, while keeping track of these all simulation parameters and results along with metadata in a database for you.

It should be easy enough to use that, even small project for simple simulations will benefit from using **sim_db**. However, if the users of your project are going to run simulations with many different parameters and/or running them on a cluster with a SLURM or PBE job scheduler, **sim_db** is a particularly good fit and highly recommended.

Check out **sim_db's** *features*, a *minimal example*, how to *install* and *use*.

CHAPTER

ONE

PURPOSE AND FEATURES

If you have already decided to use **sim_db**, then skip right ahead to *Install or Include*.

1.1 Purpose

Simulations are usually run with a large number of different sets of parameters. It may not happen at once, it may not even be the intent, but over time it probably will accumulate anyway. It is hard to keep track of all of these simulations including their parameters, without deleting any that one might want to checkout later. Especially, weeks, months or even years after the simulations were run.

When doing simulations, one will usually run a great number of simulations with different parameters. After a while it will often become a difficult to keep track of all the simulations; the parameters used to run them, the results and metadate such at the time used to perform the simulation. sim_db aim towards providing a flexible and convenient way of keeping track of all the simulation and does this by storing the parameters in a SQLite3 database.

sim_db aims to fix this problem. It will conveniently let you run a large number of simulations with different parameter values, while keep track of these all simulation parameters and results along with metadata in a database for you.

1.2 Features

1.2.1 Easy to Use

Have a look at a *minimal example* and decide for yourself.

1.2.2 Well Documented

Important for use of any code and software and believed to be a modest claim in this case. However, you are currently reading the site documenting **sim_db**, so you can again just decide for yourself.

1.2.3 Keep Track of Your Results

It obviously stores all the parameters used to run the simulation, but it also provides two mechanicams to organise your results as well. Results can can easily be written to the database, but for large results and for files that are read by other software for visualization or postprocessing it should be written to file. **sim_db** will create a unique subdirectory for you to store your results in, keep track of this subdirectory and easily jump into it.

1.2.4 Stores a lot of Metadata Automatically

sim_db stores a lot of metadata that might be useful down the line or even right away. A full list of what is stored is given in the list explaining the *default columns* in the database. In total what parameters was used, what was the result / where was it stored, what code was used to produce the result (for git projects) including what binary (was it compiled with this code), why this simulation was run, how long did it take to run on how many logical cpus on which hardware should be stored to the database — all while being less work to use that too not use **sim_db** (that is at least the idea).

1.2.5 Few Dependencies

Few dependencies make your project easier to install and to get running, and **sim_db** keeps it to *a minimum* requiring essentially only a Python interpreter.

1.2.6 Python, C, C++ and Fortran

sim_db exists for both Python, C, C++ and Fortran, and wrappers for languages that can call C functions are quite easy to add. It is also very useful that multiple programs of different languages can read the same parameters from the database. This does for example allows the plotting, visualization and after work can be seperated in a in a python program and the actual computational intensive simulation in a C++ program.

1.2.7 Thread Safe

sim_db is thread safe and intended to be used in programs running on hundreds of CPUs. Read more about use in multithreading and multiprocessing applications *here.*

1.2.8 Built to Run on both Local Machine and Super Computers/Clusters

Can easily both run your simulations on your local machine and on a super computer/cluster with a job scheduler, where it will generate the job script and submit it for you.

1.2.9 Many Print Options

With many parameters and lots of simulations it becomes important to be able to view only the simulations and parameters you want to see. **sim_db** has lots of print options to do that.

Default Columns - Metadata Stored

The default columns in the database contain the metadata of the simulations run and are the columns not containing the parameters to the simulations or results saved from the simulations. The purpose of each one is explained below and is essentially a full list of the metadata stored.

- `id` - To uniquely refer to a set of simulation parameters.
- `status` - Status of simulation: 'submitted', 'running' or 'finished'.
- `name` - To easily distinguage the different simulations.
- `describtion` - To further explain the intent of the simulation.
- `run_command` - Command to run the simulation.
- `comment` - Comment about the simulation and how it ran. Standard error may be included.
- `add_to_job_script` - Additional flags, import or load statement added to the job script for the job scheduler.
- `result_dir` - The path to where the results are stored.
- `time_submitted` - To tell how long a submition have been in queque.
- `time_started` - To tell how long a simulation used in queque and how long it have been running.
- `used_walltime` - To tell the total run time of the simulation.
- `max_walltime` - Useful if the simulation is stopped for exceeding this limit. (Also in the context of understanding the time between `time_submitted` and `time_started`.)
- `job_id` - To check the simulation when submitted to a job scheduler.
- `n_tasks` - Number of threads/cores. Needed to understand 'used_walltime'.
- `cpu_info` - Needed to compare `used_walltime` across different machines.
- `git_hash` - To be sure of which commit the simulation is run from.
- `commit_message` - A easier way to distinguage the commits than the hash.
- `git_diff_stat` - Show summary of difference between the working directory and the current commit (HEAD) at the time the simulation is run.
- `git_diff` - Show the explicit difference between the working directory and the current commit at the time when the simulation is run.
- `sha1_executables` - To tell exacetly which executable that was used to run the simulation. Useful to check that it have been compiled after any changes. Is the sha1 of any files in the `run_command`.
- `initial_parameters` - To distinguage between parameters used to run the simulation and results produced by the simulation.

CHAPTER

TWO

INSTALL OR INCLUDE

2.1 Install

(If you wish to include **sim_db** instead of installing it, jump to the next subsection.)

2.1.1 Command line tool and python package

sim_db can be can most easily be installed with `pip`:

```
$ pip install sim_db
```

The command line tool and the python package should now be available. If one are only going to use the python version of **sim_db**, it should now be ready for use. For listing all the available command of the command line tool run:

```
$ sim_db list_commands
```

(This should also work on Windows as long as your Python directory and its *Scripts/* subdirectory is added to the *%PATH%*.)

How to use any of them can be found either by running the with the `--help` or `-h` flag or reading the documentation of the *commands*. Most of the commands need to have some sets of simulation parameters added to the database to work, so read the examples below to see how to do that. (The command line tool can also be invoked by running `$ python -m sim_db`.)

Testing the import of the python package can be done with:

```
$ python -c "import sim_db"
```

And should just return without producing any errors.

Change directory to your projects root directory and initiate **sim_db** with the command:

```
$ sim_db init
```

The command will add a *.sim_db/* directory.

2.1.2 C, C++ or Fortran versions

If one are going to use in the C, C++ or Fortran version of **sim_db**, one also have to download the source code from github. It is recommended to add **sim_db** as a git submodule of your project by (inside your project) running:

```
$ git submodule add https://github.com/task123/sim_db
```

The C, C++ and Fortran libraries now needs to by complied and this can be done either with CMake or just with Make. To compile and install the libraries with CMake run these commands:

```
$ cd sim_db
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release
$ cmake --build . --target install
```

(For Fortran `$ cmake .. -DCMAKE_BUILD_TYPE=Release` must be replaced with `$ cmake .. -DCMAKE_BUILD_TYPE=Release -DFortran=ON`.)

To compile the libraries using just Make run these commands:

```
$ cd sim_db
$ make
```

(If **sim_db** haven't already been install with `pip` and you are running just make, it will be installed now.)

The libraries should now be available in *sim_db/build/* as *libsimdb.a*, *libsimdbcpp.a* and *libsimdbf.a* (+ *sim_db_mod.mod*) with headers *sim_db/include/sim_db.h* and *sim_db/include/sim_db.hpp* respectfully.

2.2 Include in Your Project

(Skip to this section of one have choosen to install **sim_db**.)

sim_db is designed to not add any additional dependencies for your project, except a absolute minimum. It therefore does not itself **need** to be installed, just included. (The command_line_tool is just python scripts, so it can be called with `$ python path_to_sim_db_dir/sim_db/__main__.py`. It is however much more convenient to just add the command line tool to your *PATH*.)

It is recommended to add **sim_db** as a git submodule in your project by (inside your project) running:

```
$ git submodule add https://github.com/task123/sim_db
```

(Otherwise it can taken from github and just copied into your project in a directory called '*sim_db*'.)

If Make is available run the following commands:

```
$ cd sim_db
$ make include
```

Answer yes when asked to add *sim_db/sim_db* to your *PATH* in *~/.bashrc* or *~/.bash_profile* and remember to source it.

If Make is not available, include *sim_db/sim_db* to your *PATH* and if the C, C++ or Fortran libraries are needed compile them with CMake by running these commands:

```
$ cd sim_db
$ mkdir build
$ cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Release
$ cmake --build .
```

(For Fortran `$ cmake .. -DCMAKE_BUILD_TYPE=Release` must be replaced with `$ cmake .. -DCMAKE_BUILD_TYPE=Release -DFortran=ON`.)

All **sim_db** commands should now be available and the C, C++ and Fortran libraries should be compiled and found in the *build/* directory with the headers in *include/*. Test the following command:

```
$ sim_db list_commands
```

It should list all the **sim_db** commands. How to use any of them can be found either by running the with the `--help` or `-h` flag or reading the documentation of the *commands*. Most of the commands need to have some sets of simulation parameters added to the database to work, so read the examples below to see how to do that.

(The full set of tests can be run with `$ pytest` or `$ python -m pytest` provided *pytest* is installed.)

Change directory to your projects root directory and initiate **sim_db** with the command:

```
$ sim_db init
```

The command will add a *.sim_db/* directory.

Since **sim_db** is just included, it will manually need to be added to the *PYTHONPATH* before using the python package. This can be done in your *~/.bashrc* or *~/.bash_profile*, but it can also be done from within your python code. For a python script in the same directory as *sim_db/* it can be done like this:

```
import sys, os.path
sys.path.append(os.path.join(os.path.dirname(os.path.abspath(__file__)), "sim_db"))
import sim_db
```

The python package should now behave as if it was installed. For files in subdirectories, just add more `os.path.dirname` calls round the path.

2.3 Dependencies

The dependencies for **sim_db** is tried to keep at a absolute minimum and it is overwhelming likely that everything is available if on a Linux machine or a Mac. The reason for the minimal dependencies and the detailed list of actual dependencies, is that the it is expected to use in project using clusers and super computers. On these clusters and super computers one typically don't have root access and only limited ability to install the dependencies.

- **Python 2.6 or greater** - A Python interpreter of version 2.6 or greater (that means that is also does work with Python 3) is needed as all the commands are written in Python. Pre-installed on almost all Linux distros and on MacOS.
- **C compiler** - A C99 compiler are needed for using **sim_db** with C, C++ or Fortran, but in that case a C compiler are usually need anyways. For C code it is of couse strictly necessary, and for C++ and Fortran its preprocessor are often used and almost always present if a C++ or Fortran compiler is present.
- **C++ compiler** - A C++98 compilers are needed for using **sim_db** with C++ code, but in that case these the compiler is of couse needed anyways. Only the examples need a C++11 compiler.
- **Fortran compiler** - A Fortran 2008 compiler are needed for using **sim_db** with Fortran code, but in that case a Fortran compiler is of course needed anyways.

Recommended:

- **Git** - Your project must use Git to get the full range of metadata. If Git is not used, metadata from Git (and the executable's SHA1 hash) is not collected. (So, there is no dramatic difference if it not used. It might, however, be useful.)
- **CMake** or just **Make** - Makes the build process much easier.
- **pytest** - Python framework used to run the tests and nothing else. Installed with `$ pip install -U pytest`.

For Windows:

- **Linux Subsystem/Cygwin/MinGW** - The the C, C++ and Fortran libraries relie on Unix (POSIX) style paths, which Cygwin/MinGW/powershell mimicks and Linux subsystem for Windows (obviousely) gives you.

(Not propery tested on windows yet.)

SQLite: **sim_db** uses a SQLite database, so a few word to explain why it is NOT listed as a dependency is probably not out of place. The sqlite3 Python module used in **sim_db's** command line tool and Python module is part of the Python Standard Library, and therefor included with Python. For the C and C++ libraries the SQLite Amalgamation (source code of SQLite in C) is included to remove it as a dependence and too provide a painfree compilation of the libraries.

2.4 License

The project is licensed under the MIT license. A copy of the license is provided here.

CHAPTER

THREE

USE

3.1 An Brief Overview

sim_db is used as follows:

- Run `$ sim_db init` in project's root directoy.
- All simulation parameters is placed in a text file with formatting described in *here*.
- The parameters are added to **sim_db's** database and the simulation is run with the `$ sim_db add_and_run` command, or with some of the other *commands*.
- In the simulation code the parameters are read from the database with the functions/methods documented *here for Python*, *here for C++*, *here for C* and *here for Fortran*.

That is the brief overview. Reading the examples below and the links above will fill in the details.

3.2 Minimal Example using Python

A parameter file called *params_mininal_python_example.txt* is located in the *sim_db/examples/* directory in the source code. The file contains the following:

```
name (string): minimal_python_example

run_command (string): python root/examples/minimal_example.py

param1 (string): "Minimal Python example is running."

param2 (int): 42
```

A python script called *minimal_example.py* and is found in the same directory:

```
import sim_db # 'sim_db/src/' have been include in the path.

# Open database and write some initial metadata to database.
sim_database = sim_db.SimDB()

# Read parameters from database.
param1 = sim_database.read("param1") # String
param2 = sim_database.read("param2") # Integer

# Print param1 just to show that the example is running.
print(param1)
```

(continues on next page)

(continued from previous page)

```
# Write final metadata to database and close connection.
sim_database.close()
```

Add the those simulations parameters to the **sim_db** database and run the simulation with:

```
$ sim_db add_and_run --filename sim_db/examples/params_minimal_python_example.txt
```

Which can also be done from within the *sim_db/examples/* directory with:

```
$ sdb add_and_run -f params_minimal_python_example.txt
```

where `sdb` is just a shorter name for `sim_db` and `-f` a shorter version of the `--filename` flag.

Minimal examples for C++ and C can also be found in the same directory.

3.3 Extensive Example using C++

This example is as the name suggerst much more extensive. It is not as straightforward as the minimal example, but it will demostrate a lot more and will also include explainations of more details.

A parameter file called params_extensive_cpp_example.txt is found in the *sim_db/examples/* directory in the source code. This parameter file contains all the possible types available in addition to some comments:

```
This is a comment, as any line without a colon is a comment.
# Adding a hashtag to the start of a comment line, make the comment easier to␣
↪recognize.

# The name parameter is highly recommended to include.
name (string): extensive_c++_example

# It is also recommended to include a description to further explain the intention of
# the simulation.
description (string): Extensive C++ example to demonstrate most features in sim_db.

# Aliases for cmake commands for compiling the example.
{cmake_config} (alias): cmake -Hroot/ -Broot/examples/build
{cmake_build} (alias): {cmake_config}; cmake --build root/examples/build --target

# This 'run_command' starts with an alias that is replaced with the above two cmake
# commands that compile the extensitve example if needed. The last part of the
# 'run_command' then run the compiled example. Each command is seperated by a
# semicolon, but they all need to be on the same line.
run_command (string): {cmake_build} extensive_cpp_example; root/examples/build/
↪extensive_cpp_example

# A parameter is added for each of the avaiable types.
param1_extensive (int): 3
param2_extensive (float): -0.5e10
param3_extensive (string): "Extensive C++ example is running."
param4_extensive (bool): True
param5_extensive (int array): [1, 2, 3]
param6_extensive (float array): [1.5, 2.5, 3.5]
param7_extensive (string array): ["a", "b", "c"]
param8_extensive (bool array): [True, False, True]
```

(continues on next page)

(continued from previous page)

```
# Include parameters from another parameter file.
include_parameter_file: root/examples/extra_params_example.txt

# Change a parameter value from the included parameter file to demonstrate that
# it is the last parameter value that count for a given parameter name.
extra_param1 (int): 9
```

Notice that the parameters names are different from the *minimal example*. This is because *param1* and *param2* are differnt types in this example and the type of a parameter can not change in the database. (In practice this is a very good thing. However, if one add the wrong type to the database the first time, the `delete_sim` and `delete_empty_columns` commands must be used before making a new column with correct type.)

The line in the parameter file starting with *include_parameter_file:* will be substituted with the contain of the specified *extra_params_example.txt* file, found in the same directory:

```
# Extra parameters included in the extensive examples.

extra_param1 (int): 7
extra_param2 (string): "Extra params added."
extra_param3 (bool): False
```

This syntax for can be used to simplify the parameter files for projects with many parameters. One can for instance have different parameter files for different kindes of parameters, such as printing parameters. The same parameter name, with the same type, can be added to multiple lines in the parameter files, but all the previous parameter values will be overwritting by the last one. This way one can have a default paramter file, include that in any other parameter file and just change the necesarry parameters. Consider including the other parameter file before the parameters to the sure that they are not modified in the other parameter files, and be careful with the order of included parameter files.

extensive_example.cpp is also found in the same directory:

```cpp
#include "sim_db.hpp"  // Parts from the standard library is also included.

int main(int argc, char** argv) {
    // Open database and write some initial metadata to database.
    sim_db::Connection sim_db(argc, argv);

    // Read parameters from database.
    auto param1 = sim_db.read<int>("param1_extensive");
    auto param2 = sim_db.read<double>("param2_extensive");
    auto param3 = sim_db.read<std::string>("param3_extensive");
    auto param4 = sim_db.read<bool>("param4_extensive");
    auto param5 = sim_db.read<std::vector<int> >("param5_extensive");
    auto param6 = sim_db.read<std::vector<double> >("param6_extensive");
    auto param7 = sim_db.read<std::vector<std::string> >("param7_extensive");
    auto param8 = sim_db.read<std::vector<bool> >("param8_extensive");

    // Demonstrate that the simulation is running.
    std::cout << param3 << std::endl;

    // Write all the possible types to database.
    // Only these types are can be written to the database.
    sim_db.write("example_result_1", param1);
    sim_db.write("example_result_2", param2);
    sim_db.write("example_result_3", param3);
    sim_db.write("example_result_4", param4);
```

(continues on next page)

(continued from previous page)

```
    sim_db.write("example_result_5", param5);
    sim_db.write("example_result_6", param6);
    sim_db.write("example_result_7", param7);
    sim_db.write("example_result_8", param8);

    // Make unique subdirectory for storing results and write its name to
    // database. Large results are recommended to be saved in this subdirectory.
    std::string name_results_dir =
            sim_db.unique_results_dir("root/examples/results");

    // Write some results to a file in the newly create subdirectory.
    std::ofstream results_file;
    results_file.open(name_results_dir + "/results.txt");
    for (auto i : param6) {
        results_file << i << std::endl;
    }

    // Check if column exists in database.
    bool is_column_in_database = sim_db.column_exists("column_not_in_database");

    // Check if column is empty and then set it to empty.
    bool is_empty = sim_db.is_empty("example_result_1");
    sim_db.set_empty("example_result_1");

    // Get the 'ID' of the connected simulation an the path to the project's
    // root directory.
    int id = sim_db.get_id();
    std::string path_proj_root = sim_db.get_path_proj_root();

    // Add an empty simulation to the database, open connection and write to it.
    sim_db::Connection sim_db_2 = sim_db::add_empty_sim(path_proj_root, false);
    sim_db_2.write<int>("param1_extensive", 7);

    // Delete simulation from database.
    sim_db_2.delete_from_database();
}
```

Adding the simulation parameters to the **sim_db** database and running the simulation can be just as in the minimal example:

```
$ sim_db add_and_run -f sim_db/examples/params_extensive_cpp_example.txt
```

If the filename passed to either the `add_sim` or `add_and_run` commands starts with *root/* that part will be substituted with the full path to the projects root directory (where *.sim_db/* is located). This way the same path to a parameter file can be passed from anywhere within the project.

It is, as the name suggest, the *run_command* parameter that is used to run the simulation. And it need to included in the parameter file for the `run_sim`, `add_and_run` and `submit_sim` commands to work. (The *name* parameter is needed for the *unique_results_dir* function to work, but is always recommended to included reguardless of whether that function is used or not.)

Notice that when it is run, it first call two `cmake` commands to compile the code if needed. What `cmake` does is equvalient to the following command called from *sim_db/examples/* (given that the static C++ library are compiled and located in *sim_db/build/*):

```
$ c++ -std=c++11 -o build/extensive_cpp_example extensive_example.cpp -I../include -L.
↪./build -lsimdbcpp -lpthread -ldl -m
```

If the add_and_run command is run without any flags, it will look for any files in the current directory matching the ones *Parameter filenames* in *.sim_db/settings.txt* and add and run the first match. The command is often divided into adding the simulations parameters to the database with:

```
$ sdb add
```

and running the simulation:

```
$ sdb run
```

When passed without any flags run will run the last simulation added, that have not yet been started. To run a spesific simulation different from the last one, add the --id flag:

```
$ sdb run --id 'ID'
```

where '*ID*' is the a unique number given to each set of simulation parameters added to the database. The '*ID*' is printed when using add, but to check the '*ID*' of the last couple of siulations added one can run:

```
$ sdb print -n 2 -c id name
```

print have lots of flags to control and limit what is printed. The -n 2 flag prints the last two entries. -c id name limit the output to just the column named *id* and *name*. -v -i 'ID' are two other useful flags that prints the columns in the database as rows for the set of parameters that have id 'ID'. To avoid typing out lots of flags and column names/parameter names for each time one would like to print something, one can set *Personlized print configurations* in *settings.txt*. *Personlized print configurations* are a set of print_sim flags that are given a name and can be set as default or called as:

```
$ sdb print -p 'name_of_personalized_config'
```

When running $ sdb run --id 'ID', the flags --id 'ID' --path_proj_root 'PATH_TO_PROJECT_ROOT_DIR is added to the *run_command* before it is run, so that the program know where the database is and which 'ID' to read from. So, the executable prodused by make or the compile command stated above can be run in the *sim_db/examples/* directoy as:

```
$ ./extensive_cpp_example --id 'ID' --path_proj_root ".."
```

The *sim_db/* directory is there the project root directory, and where *.sim_db/* is located.

The example stored some results in a unique subdirectory, which is the recommended way to store large results. To change the directory to that subdirectory, so one can check out the results, just run:

```
$ sdb cd_results_dir --id 'ID'
```

To run this example or any other simulation on a cluster or a super computer with a job scheduler, just fill out the *Settings for job scheduler* in *settings.txt* and run:

```
$ sdb submit --id 'ID' --max_walltime 00:00:10 --n_tasks 1
```

The command will create a job script and submit it to the job scheduler. **sim_db** supports job scheduler SLURM and PBS, but it should be quite easy to add more. *n_tasks* is here the number of logical CPUs you want to run on, and can together with *max_walltime* also be set in the parameter file.

It does not make any sense to run such a small single threaded example on a super computer. If one uses a super computer, one are much more likely to want to run a large simulation on two entire nodes:

```
$ sdb submit --id 'ID' --max_walltime 10:30:00 --n_nodes 2
```

If a number of simulations are added all including the parameters *max_walltime* and *n_tasks*, one can simply run:

```
$ sdb submit
```

, which will run all simulations that have not been run yet after a confimation question.

Extensive examples for Python and C can also be found in the same directory, *sim_db/examples/*, on github.

3.4 Multithreading and Multiprocessing

sim_db is thread safe and can be used in both multithreading and multiprocessing appications (and is intended for such use). **sim_db** utilies SQLite as its database engine and is thread safe in the same way that SQLite is thread safe. This means that connections to the database should not be shared across threads. Instead each thread/process should have its own connection (instance of a SimDB class).

One should also be aware of that writing to the database is blocking - other threads/processes have to wait before they can read from or write to the database and could potentially time out. Extensive concurrent writing to the database, must therefor be avoided (or dealt with). A 'only_if_empty' option for writing is however provided as a convenient way for many thead/processes to write to the same column without additional syncronisation.

3.4.1 In a nutshell:

- **sim_db** is thread safe.
- Each thread/process MUST have its own connection.
- Avoid extensive concurrent writing. (Can be done with the 'only_if_empty' option.)

CHAPTER

FOUR

PARAMETER FILES

A text file with a particular format is used to pass parameters to the simulations. The parameters are easily edited in the text files, and added to the database and used to run the simulation with the command:

```
$ sim_db add_and_run -f sim_param_file.txt
```

4.1 The Format

The format of the parameter file is for each parameter as following:

parameter_name (*type*): *parameter_value*

type can be *int*, *float*, *string*, *bool* or *int/float/string/bool array*. Lines without any colon is ignored. This means that the parameter name, type, colon and value MUST all be on the same line and colons can ONLY be used on lines with parameters (except when including other parameter files).

The '*run_command* (*string*): *command*' parameter need to be one of the parameters in the parameter file for the `run_sim`, `add_and_run` and `submit_sim` commands to work. The '*name* (*string*): *name_of_simulation*' stricly only needed if the *unique_results_dir* function is used, but it is always recommeneded to include.

The parameters from other parameter files can be included with a line like this:

include_parameter_file: *name_parameter_file*

The line is simply substituted with the contain of the file, and the included files are also allowed to include other parameter files. (Including any files in which the file is itself included, will cause an infinite loop). `include parameter file:` can also be used instead of `include parameter file:`.

It is perfectly fine to have the same parameter name, with the same type, in multiple plasses in the parameter file. The previous parameter values will just be overwritten by the last one.

It is also possible create an alias taking the follow format:

{string_to_replace} (alias): *replacement_string*

Any occurrence of *{string_to_replace}* on any line after a colon and below this alias definition, will be replaced by *replacement_string*. This also includes any parameter files included after the alias. It is required that the alias name starts and ends with curly brackets as in the above example. Aliases is powerful and can be very useful if a parameter or part of a parameter appear in multiple parameters. It also allows for a layer of abstraction. But be careful as this can cause unintended replacements and can easily make the parameter file harder to read. Avoid excessive use and use unique alias names.

The format is very flexible, as the parameters can be in any order and lines without colons can be used freely to comment, describe and organise (with blank lines and indents) the parameters. This makes it easy to make the parameters of the simulation well understood. It is also very fast to change any number of parameters as it is only a text file that need to be edited. The parameters can also be organised in different files using `include_parameter_file:`.

4.2 run_command

A couple words about how the 'run_command' is treated by **sim_db** can be helpful. The `run_sim`, `add_and_run` and `submit_sim` commands uses the 'run_command' parameter (given as `run_command (string): command`) to run the simultion. Before the 'run_command' is run a couple things are done to it. It is split at all semicolons into multiple commands, all occurrences of 'root/' will be replaced with the path to the root directory of the project and ' # ' will be replaced with the ' *n_tasks* ', where *n_tasks* is a parameter passed to one of the commands that uses the run_command. As an example let's say a project with root directory, `/path/root/dir`, have a run_command defined as:

`run_command (string):  make -C root/example; mpirun # root/example/program`

in `sim_params.txt`. Running `$ sim_db add_and_run -f sim_params.txt -n 4` will then in turn run the commands:

```
$ make -C /path/root/dir/example
$ mpirun 4 /path/root/dir/example/program
```

(If your run_command contain a 'root/' that your don't want to be replaced, you should substitute it with an alias.)

4.3 Example

Example of a parameter file that uses all the different parameter types:

```
This is a comment, as any line without a colon is a comment.
# Adding a hashtag to the start of a comment line, make the comment easier to␣
↪recognize.

# The name parameter is highly recommended to include.
name (string): extensive_c_example

# It is also recommended to include a description to further explain the intention of
# the simulation.
description (string): Extensive C example to demonstrate most features in sim_db.

# Aliases for cmake commands for compiling the example.
{cmake_config} (alias): cmake -Hroot/ -Broot/examples/build
{cmake_build} (alias): {cmake_config}; cmake --build root/examples/build --target

# This 'run_command' starts with an alias that is replaced with the above two cmake
# commands that compile the extensitve example if needed. The last part of the
# 'run_command' then run the compiled example. Each command is seperated by a
# semicolon, but they all need to be on the same line.
run_command (string): {cmake_build} extensive_c_example; root/examples/build/
↪extensive_c_example

# A parameter is added for each of the avaiable types.
param1_extensive (int): 3
param2_extensive (float): -0.5e10
param3_extensive (string): "Extensive C example is running."
param4_extensive (bool): True
param5_extensive (int array): [1, 2, 3]
param6_extensive (float array): [1.5, 2.5, 3.5]
param7_extensive (string array): ["a", "b", "c"]
param8_extensive (bool array): [True, False, True]
```

(continues on next page)

(continued from previous page)

```
# Include parameters from another parameter file.
include_parameter_file: root/examples/extra_params_example.txt

# Change a parameter value from the included parameter file to demonstrate that
# it is the last parameter value that count for a given parameter name.
extra_param1 (int): 9
```

The line in the parameter file starting with *include_parameter_file:* will be substituted with the contain of the specified *extra_params_example.txt* file:

```
# Extra parameters included in the extensive examples.

extra_param1 (int): 7
extra_param2 (string): "Extra params added."
extra_param3 (bool): False
```

4.4 Filename

The filename of the text file with the parameters can be anything (to describe what simulation it is used for) and just passed to the `add_sim` and `add_and_run` commands with the `--filename` or `-f` option. That option can however be omitted be naming the parameter file *sim_params.txt* or any other name added under the *Parameter filenames* header in the *settings.txt* file in the *.sim_db/* directory.

4.5 Commands Realated to Parameters

The parameters in a parameter file can be added to the database with the `add` command or added and run with `add_and_run`. The file can then be edited to add new simulations, but a parameter can also be edited or added to an already added simulation with the `update` command. One can also generate a new parameter file from a simulation in the database with the `extract_params` commands, which can be a quick way of running simulations similar to that one. Finally it is very useful to get familiar with the `print` command to print the parameters and other things from simulations in the database.

CHAPTER

FIVE

COMMAND LINE TOOL

The command line tool is called sim_db, but can also be called with sdb. It has a syntax simular to git, where commands are passed to sim_db followed by the arguments to the command: $ sim_db <command> [<args>].

All the available commands can be listed with the list_commands (run as $ sim_db list_commands). What they do and which arguments they take is found by passing the --help or -h option to any of the commands. The same information is found below.

Commands ending in *_sim* can also be used without this ending, so add is the same commend as add_sim.

All the commands can be called from anywhere is your project after the init command is called in your projects root directory. The only exception is inside the sim_db/ directory if that is included.

5.1 Example of Commands to Run the Other Examples

The minimal and extensive examples of the Python, C and C++ versions of **sim_db** can all be run with the bash script run_all_examples.sh found inside the *sim_db/examples/* directory of the source code. The part of the script that run the minimal python example is show here:

```
######################## Run minimal python example ###########################

# Add example parameters to database for minimal python example.
sim_db add --filename root/examples/params_minimal_python_example.txt

# Get hold of the ID of the exampel parameters for minimal python example.
id_for_minimal_python_example=`sim_db print -n 1 --columns id --no_headers`

# Run minimal_example.py.
sim_db run --id ${id_for_minimal_python_example}

# Delete example simulation from database.
sim_db delete --id ${id_for_minimal_python_example} --no_checks
```

Note that *sim_db/* is here the root directory of the project, and sim_db init have already been called in that directory.

5.2 sim_db

For running simulations and keeping track of its parameters, results and metadata.

```
usage: sim_db [--help] [--version] <command> [<args>]

Some common sim_db commands:

init           Initialise 'sim_db' for use in project.
add            Add set of simulation parameters to database.
print          Print parameters in database.
run            Run simulation with parameters from database.
list_commands  List all available commands.
```

5.2.1 Positional Arguments

command The command, 'list_commands', will print all available commands.

5.2.2 Named Arguments

--version Print version of sim_db. Must be passed as the only parameter.

Default: False

5.3 Commands

5.3.1 add_and_run

Add simulation and submit it.

```
usage: sim_db add_and_run [-h] [--filename FILENAME] [-n N]
                          [--add_unique_results_dir]
```

Named Arguments

--filename, -f Name of parameter file to add and run.

-n Number of threads/core to run the simulation on.

--add_unique_results_dir, -u Add a unique subdirectory for the simulation in the 'superdir_for_results' directory in the settings and write it to 'results_dir' in the database.

Default: False

5.3.2 add_and_submit

Add simulation and submit it.

```
usage: sim_db add_and_submit [-h] [--filename FILENAME]
                             [--max_walltime MAX_WALLTIME] [--n_tasks N_TASKS]
                             [--n_nodes N_NODES]
                             [--additional_lines ADDITIONAL_LINES]
                             [--notify_all] [--notify_fail] [--notify_end]
                             [--no_confirmation] [--do_not_submit_job_script]
                             [--add_unique_results_dir]
```

Named Arguments

--filename, -f Name of parameter file added and submitted.

--max_walltime Maximum walltime the simulation can use, given in 'hh:mm:ss' format.

--n_tasks Number of tasks to run the simulation with. A warning is given if it is not a multiple of the number of logical cores on a node.

--n_nodes Number of nodes to run the simulation on.

--additional_lines Additional lines added to the job script.

--notify_all Set notification for when simulation begins and ends or if it fails.

Default: False

--notify_fail Set notification for if simulation fails.

Default: False

--notify_end Set notification for when simulation ends or if it fails.

Default: False

--no_confirmation Does not ask for confirmation about submitting all simulations with status 'new'

Default: False

--do_not_submit_job_script Makes the job script, but does not submit it.

Default: False

--add_unique_results_dir, -u Add a unique subdirectory for the simulation in the 'superdir_for_results' directory in the settings and write it to 'results_dir' in the database.

Default: False

5.3.3 add_column

Add column to database.

```
usage: sim_db add_column [-h] --column COLUMN --type TYPE
```

Named Arguments

--column, -c <Required> Name of the new column.

--type, -t <Required> Type of the column. 'INTEGER', 'REAL', 'TEXT', 'int', 'float', 'string', 'bool' and 'int/float/string/bool array' are the valid choices.

5.3.4 add_comment

Add comment to simulation in database.

```
usage: sim_db add_comment [-h] --id ID [--comment COMMENT]
                          [--filename FILENAME] [--append]
```

Named Arguments

--id, -i <Required> ID of the simulation to add the comment.

--comment, -c Comment to add.

--filename, -f Filename of a file which content are to be added as a comment. Only the last 3000 characters will be added.

--append, -a Append comment or file to the current comment.

Default: False

5.3.5 add_range

Add a range of simulations to the database.

```
usage: sim_db add_range_sim  [-h] [--filename FILENAME] --columns COLUMNS
                             [COLUMNS ...]
                             [--lin_steps LIN_STEPS [LIN_STEPS ...]]
                             [--exp_steps EXP_STEPS [EXP_STEPS ...]]
                             [--end_steps END_STEPS [END_STEPS ...]]
                             [--n_steps N_STEPS [N_STEPS ...]]
```

Named Arguments

--filename, -f Name of parameter file added as the first in the range.

--columns, -c <Required> Names of the column for which the range varies. The cartisian products of the varing columns are added to the database. The column type MUST be a integer or a float.

Default: []

--lin_steps Linear step distance. NEXT_STEP = PREV_STEP + LIN_STEP. If columns have both linear and exponential steps, both will be used. NEXT_STEP = LIN_STEP + PREV_STEP * EXP_STEP

Default: []

--exp_steps Exponential step distance. NEXT_STEP = PREV_STEP * EXP_STEP. If columns have both linear and exponential steps, both will be used. NEXT_STEP = LIN_STEP + PREV_STEP * EXP_STEP

Default: []

--end_steps End step of range. The range includes the end, but not anything past it. If both 'end_steps' and 'n_steps' are used, both endpoint need to be reached.

Default: []

--n_steps Number of steps in the range. That means one step gives to simulations added. If both 'end_steps' and 'n_steps' are used, both endpoint need to be reached.

Default: []

5.3.6 add_sim

Add simulation to database.

```
usage: sim_db add_sim [-h] [--filename FILENAME]
```

Named Arguments

--filename, -f Name of parameter file added and submitted.

5.3.7 cd_results_dir

Change the current working directory to the 'results_dir' of the specified simulation or the last entry if not specified. (This is done by creating a new subshell, so '$ exit' can be used to return to the original directory and shell instance.)

```
usage: sim_db cd_results_dir [-h] [--id ID] [-n N]
```

Named Arguments

--id, -i 'ID' of the simulation in the 'sim.db' database.

-n n'th last entry in the 'sim.db' database.

5.3.8 combine_dbs

Combine two databases into a new one.

```
usage: sim_db combine_dbs [-h] path_db_1 path_db_2 name_new_db
```

Positional Arguments

path_db_1 <Required> Path to ‘ database 1.

path_db_2 <Required> Path to ‘ database 2.

name_new_db <Required> Name of the new database.

5.3.9 delete_empty_columns

Delete all empty columns in the sim.db, except the default ones.

```
usage: sim_db delete_empty_columns [-h]
```

5.3.10 delete_results_dir

Delete results in ‘results_dir’ of specified simulations.

```
usage: sim_db delete_results_dir [-h] [--id ID [ID ...]] [--where WHERE]
                                 [--no_checks]
                                 [--not_in_db_but_in_dir NOT_IN_DB_BUT_IN_DIR]
```

Named Arguments

--id, -i ID’s of simulation which ‘results_dir’ to deleted.

Default: []

--where, -w Condition for which simulation’s ‘results_dir’ to deleted. Must be a valid SQL (sqlite3) command when added after WHERE in a SELECT command.

--no_checks No questions are asked about wheather you really want to delete the ‘results_dir’ of specified simulation.

Default: False

--not_in_db_but_in_dir Delete every folder in the specified directory that is not a ‘results_dir’ in the ‘, so use with care. Both relative and absolute paths can be used.

5.3.11 delete_sim

Delete simulations from sim.db.

```
usage: sim_db delete_sim [-h] [--id ID [ID ...]] [--where WHERE] [--all]
                         [--no_checks]
```

Named Arguments

--id, -i	ID's of runs to delete. Default: []
--where, -w	Condition for which entries should be deleted. Must be a valid SQL (sqlite3) command when added after WHERE in a DELETE command.
--all	Delete all simulation from database. Default: False
--no_checks	No questions are asked about wheter you really want to delete simulation or the 'results_dir' of the simulation. Default: False

5.3.12 duplicate_and_run

Duplicate simulation in database and run it. All parameters (including possible results) of specified simulation is duplicated with the exception of 'id' and 'status', which is kept unique and set to 'new' respectfully.

```
usage: sim_db duplicate_and_run [-h] --id ID [-n N]
```

Named Arguments

--id, -i	<Required> 'ID' of the simulation parameters in the 'sim.db' database that should be duplicated.
-n	Number of threads/core to run the simulation on.

5.3.13 duplicate_sim

Duplicate simulation in database. All parameters (including possible results) of specified simulation is duplicated with the exception of 'id' and 'status', which is kept unique and set to 'new' respectfully.

```
usage: sim_db duplicate_sim  [-h] --id ID
```

Named Arguments

--id, -i	<Required> 'ID' of the simulation parameters in the 'sim.db' database that should be duplicated.

5.3.14 extract_params

Extract parameter file from sim.db.

```
usage: sim_db extract_params [-h] --id ID [--filename FILENAME]
                             [--default_file] [--also_results] [--all]
```

Named Arguments

--id, -i <Required> ID of the simulation which parameter one wish to extract.

--filename, -f Name of parameter file generated.

--default_file, -d Write parameters to the first of the 'Parameter filenames' in settings.txt. Ask for confirmation if file exists already.

Default: False

--also_results Also extract results - parameters added during the simulation excluding metadata. Default is to just extract the parameters added before the simulation was run and found in 'initial_parameters' column.

Default: False

--all Extract all non empty parameters, including metadata.

Default: False

5.3.15 get

Get value from 'column' of simulation specified or last entry if not specified.

```
usage: sim_db get [-h] [--id ID] [-n N] column
```

Positional Arguments

column Column in database from where to get the value.

Named Arguments

--id, -i 'ID' of the simulation in the 'sim.db' database.

-n n'th last entry in the 'sim.db' database.

5.3.16 init

Initialises 'sim_db' and must be called before using 'sim_db'. Will create a '.sim_db/' directory.

```
usage: sim_db init [-h] [--path PATH]
```

Named Arguments

--path Path to the top directory of project. If not passed as an argument, the current working directory is assumed to be the top directory.

5.3.17 list_commands

Print a list of all the commands.

```
usage: sim_db list_commands [-h]
```

5.3.18 list_print_configs

Print a list of all the personalized print configurations.

```
usage: sim_db list_print_configs [-h]
```

5.3.19 print_sim

Print content in sim.db. The default configuration corresponding to the '-p default' option is applied first, as long as the '–columns'/'-c' option is not passed. It can can however be overwritten, as only the last occcurence of any flag is used.

```
usage: sim_db print_sim [-h] [--id ID [ID ...]] [--not_id NOT_ID [NOT_ID ...]]
                        [-n N] [--columns COLUMNS [COLUMNS ...]]
                        [--not_columns NOT_COLUMNS [NOT_COLUMNS ...]]
                        [--col_by_num COL_BY_NUM [COL_BY_NUM ...]]
                        [--where WHERE] [--sort_by SORT_BY] [--column_names]
                        [--all_columns] [--empty_columns] [--params]
                        [--results] [--metadata] [--no_headers]
                        [--max_width MAX_WIDTH] [--first_line] [--vertically]
                        [-p P] [--diff]
```

Named Arguments

--id, -i List of ID's.

--not_id List of ID's not to print. Takes president over '–id'.

-n Number of row printed from the bottom up.

--columns, -c Name of the columns to print.

--not_columns Name of the columns not to print. Takes presidents over '–columns'.

--col_by_num Number of the columns to print. All non empty columns are printed by default.

--where, -w Add constraints to which columns to print. Must be a valid SQL (sqlite3) command when added after WHERE in a SELECT command.

Default: "id > -1"

--sort_by What to sort the output by. Must be a valid SQL (sqlite3) command when added after ORDER BY in a SELECT search. Defalut is id.

Default: "id"

--column_names Print name and type of all columns.

Default: False

--all_columns Print all columns. Takes president over '–not_columns'.

Default: False

--empty_columns Print empty columns. Otherwise only non empty columns are printed.

Default: False

--params Print the parameters added before the simulation run.

Default: False

--results Print results - the parameters added during the simulation, excluding metadata.

Default: False

--metadata Print metadata. '–params', '–results' and '–metadata' will together print all non empty columns.

Default: False

--no_headers Print without any headers.

Default: False

--max_width Upper limit for the width of each column. Default is no limit.

--first_line Print only the first line of any entry.

Default: False

--vertically, -v Print columns vertically.

Default: False

-p Personal print configuration. Substituted with the print configuration in 'settings.txt' corresponding to the provided key string.

--diff, -d Remove columns with the same value for all the simulations. This leaves only the parameters that are different between the simulations.

Default: False

5.3.20 run_seriel_sims

Run multiple simulations in series. If no ID's or conditions are given all the new simulations are run.

```
usage: sim_db run_serial_sims [-h] [--id ID [ID ...]]
                              [--where WHERE [WHERE ...]]
```

Named Arguments

--id, -i 'IDs' of the simulation parameters in the 'sim.db' database that should be used in the simulation.

Default: []

--where, -w Conditions of the simulation parameters in the 'sim.db' database that should be used in the simulation.

Default: []

5.3.21 run_sim

Run simulation with ID in database.

```
usage: sim_db run_sim [-h] [--id ID] [-n N] [--allow_reruns]
                      [--add_unique_results_dir]
```

Named Arguments

--id, -i 'ID' of the simulation parameters in the 'sim.db' database that should be used in the simulation.

-n Number of threads/core to run the simulation on.

--allow_reruns Allow simulations with non 'new' status to run.

Default: False

--add_unique_results_dir, -u Add a unique subdirectory for the simulation in the 'superdir_for_results' directory in the settings and write it to 'results_dir' in the database.

Default: False

5.3.22 settings

Print and change settings. The settings can also be changed be editing the '.settings.txt' file.

```
usage: sim_db settings [-h] command
```

Positional Arguments

command 'print', 'add', 'remove' or 'reset_to_default'

5.3.23 submit_sim

Submit job

```
usage: sim_db submit_sim [-h] [--id ID [ID ...]] [--allow_reruns]
                         [--max_walltime MAX_WALLTIME [MAX_WALLTIME ...]]
                         [--n_tasks N_TASKS [N_TASKS ...]]
                         [--n_nodes N_NODES [N_NODES ...]]
                         [--additional_lines ADDITIONAL_LINES [ADDITIONAL_LINES ...]]
                         [--notify_all] [--notify_fail] [--notify_end]
                         [--no_confirmation] [--do_not_submit_job_script]
                         [--add_unique_results_dir]
```

Named Arguments

--id, -i ID of simulations to submit.

--allow_reruns Allow simulations with non 'new' status to be submitted.

Default: False

--max_walltime Maximum walltime the simulation can use, given in 'hh:mm:ss' format.

--n_tasks Number of tasks to run the simulation with. A warning is given if it is not a multiple of the number of logical cores on a node.

--n_nodes Number of nodes to run the simulation on.

--additional_lines Additional lines added to the job script.

Default: []

--notify_all Set notification for when simulation begins and ends or if it fails.

Default: False

--notify_fail Set notification for if simulation fails.

Default: False

--notify_end Set notification for when simulation ends or if it fails.

Default: False

--no_confirmation Does not ask for confirmation about submitting all simulations with status 'new'

Default: False

--do_not_submit_job_script Makes the job script, but does not submit it.

Default: False

--add_unique_results_dir, -u Add a unique subdirectory for the simulation in the 'superdir_for_results' directory in the settings and write it to 'results_dir' in the database.

Default: False

5.3.24 update_sim

Update content in sim.db.

```
usage: sim_db update_sim [-h] [--id ID] [--where WHERE] --columns COLUMNS
                         [COLUMNS ...] --values VALUES [VALUES ...]
                         [--db_path DB_PATH]
```

Named Arguments

--id, -i ID of run to update.

--where, -w Condition for which entries should be updated. Must be a valid SQL (sqlite3) command when added after WHERE in a UPDATE command.

Default: "id > -1"

--columns, -c <Required> Name of column to update in runs.

--values, -v <Required> New value updated at run with id and column as specifed.

--db_path Full path to the database used.

CHAPTER SIX

SIM_DB FOR PYTHON

6.1 Minimal Example using Python

A parameter file called *params_mininal_python_example.txt* is located in the *sim_db/examples/* directory in the source code. The file contains the following:

```
name (string): minimal_python_example

run_command (string): python root/examples/minimal_example.py

param1 (string): "Minimal Python example is running."

param2 (int): 42
```

A python script called *minimal_example.py* and is found in the same directory:

```python
import sim_db # 'sim_db/src/' have been include in the path.

# Open database and write some initial metadata to database.
sim_database = sim_db.SimDB()

# Read parameters from database.
param1 = sim_database.read("param1") # String
param2 = sim_database.read("param2") # Integer

# Print param1 just to show that the example is running.
print(param1)

# Write final metadata to database and close connection.
sim_database.close()
```

Add the those simulations parameters to the **sim_db** database and run the simulation from the *sim_db/examples/* directory with:

```
$ sim_db add_and_run -f params_minimal_python_example.txt
```

6.2 Extensive Example using Python

A parameter file called params_extensive_python_example.txt is found in the *sim_db/examples/* directory in the source code. This parameter file contains all the possible types available in addition to some comments:

```
This is a comment, as any line without a colon is a comment.
# Adding a hashtag to the start of a comment line, make the comment easier to␣
↪recognize.

# The name parameter is highly recommended to include.
name (string): extensive_python_example

# It is also recommended to include a description to further explain the intention of
# the simulation.
description (string): Extensive Python example to demonstrate most features in sim_db.

run_command (string): python root/examples/extensive_example.py

# A parameter is added for each of the avaiable types.
param1_extensive (int): 3
param2_extensive (float): -0.5e10
param3_extensive (string): "Extensive Python example is running."
param4_extensive (bool): True
param5_extensive (int array): [1, 2, 3]
param6_extensive (float array): [1.5, 2.5, 3.5]
param7_extensive (string array): ["a", "b", "c"]
param8_extensive (bool array): [True, False, True]

# Include parameters from another parameter file.
include_parameter_file: root/examples/extra_params_example.txt

# Change a parameter value from the included parameter file to demonstrate that
# it is the last parameter value that count for a given parameter name.
extra_param1 (int): 9
```

The line in the parameter file starting with *include_parameter_file:* will be substituted with the contain of the specified *extra_params_example.txt* file, found in the same directory:

```
# Extra parameters included in the extensive examples.

extra_param1 (int): 7
extra_param2 (string): "Extra params added."
extra_param3 (bool): False
```

extensive_example.py is also found in the same directory:

```python
import sim_db # 'sim_db/' have been included in the path.

# Open database and write some initial metadata to database.
sim_database = sim_db.SimDB()

# Read parameters from database.
param1 = sim_database.read("param1_extensive") # Integer
param2 = sim_database.read("param2_extensive") # Float
param3 = sim_database.read("param3_extensive") # String
param4 = sim_database.read("param4_extensive") # Bool
param5 = sim_database.read("param5_extensive") # List of integers
```

(continues on next page)

(continued from previous page)

```
param6 = sim_database.read("param6_extensive") # List of floats
param7 = sim_database.read("param7_extensive") # List of strings
param8 = sim_database.read("param8_extensive") # List of bools

# Demonstrate that the simulation is running.
print(param3)

# Write to database.
sim_database.write("example_result_1", param1, type_of_value="int")
sim_database.write("example_result_2", param2, type_of_value="float")
sim_database.write("example_result_3", param3, type_of_value="string")
sim_database.write("example_result_4", param4, type_of_value="bool")
sim_database.write("example_result_5", param5, type_of_value="int array")
sim_database.write("example_result_6", param6, type_of_value="float array")
sim_database.write("example_result_7", param7, type_of_value="string array")
sim_database.write("example_result_8", param8, type_of_value="bool array")

# Make unique subdirectory for storing results and write its name to database.
results = np.array(param6)
name_results_dir = sim_database.unique_results_dir("root/examples/results")
np.savetxt(name_results_dir + "/results.txt", results)

# Check if column exists in database.
is_column_in_database = sim_database.column_exists("column_not_in_database")

# Check is column is empty and then set it to empty.
sim_database.is_empty("example_result_1")
sim_database.set_empty("example_result_1")

# Get the 'ID' of the connected simulation and the path to the root directory.
db_id = sim_database.get_id()
path_proj_root = sim_database.get_path_proj_root()

# Write final metadata to the database and close the connection.
sim_database.close()

# Add an empty simulation to database, open connection and write to it.
sim_database_2 = sim_db.add_empty_sim(False)
sim_database_2.write("param1_extensive", 7, type_of_value="int")

# Delete simulation from the database.
sim_database_2.delete_from_database()

# Close connection to the database.
sim_database_2.close()
```

Add the those simulations parameters to the **sim_db** database and run the simulation from the *sim_db/examples/* directory with:

```
$ sdb add_and_run -f params_extensive_python_example.txt
```

6.3 Python API Referance

Read and write parameters, results and metadata to the 'sim_db' database.

class SimDB(*store_metadata=True*, *db_id=None*, *rank=None*, *only_write_on_rank=0*)

To interact with the **sim_db** database.

For an actuall simulation it should be initialised at the very start of the simulation (with 'store_metadata' set to True) and closed with `close()` at the very end of the simulation. This must be done to add the corrrect metadata.

For multithreading/multiprocessing each thread/process MUST have its own connection (instance of this class) and MUST provide it with its rank.

__init__(*store_metadata=True*, *db_id=None*, *rank=None*, *only_write_on_rank=0*)

Connect to the **sim_db** database.

Parameters

- **store_metadata** (*bool*) – If False, no metadata is added to the database. Typically used when postprocessing (visualizing) data from a simulation.
- **db_id** (*int*) – ID number of the simulation parameters in the **sim_db** database. If it is 'None', then it is read from the argument passed to the program after option '–id'.
- **rank** (*int*) – Number identifing the calling process and/or thread. (Typically the MPI or OpenMP rank.) If provided, only the 'rank' matching 'only_write_on_rank' will write to the database to avoid too much concurrent writing to the database. Single process and threaded programs may ignore this, while multithreading/multiprocessing programs need to provide it.
- **only_write_on_rank** (*int*) – Number identifing the only process/thread that will write to the database. Only used if 'rank' is provided.

read(*column*, *check_type_is=''*)

Read parameter in 'column' from the database.

Return None if parameter is empty.

Parameters

- **column** (*str*) – Name of the column the parameter is read from.
- **check_type_is** – Throws ValueError if type does not match 'check_type_is'.The valid types the strings 'int', 'float', 'bool', 'string' and 'int/float/bool/string array' or the types int, float, bool, str and list.

Raises

- **ColumnError** – If column do not exists.
- **ValueError** – If return type does not match 'check_type_is'.
- **sqlite3.OperationalError** – Waited more than 5 seconds to read from the database, because other threads/processes are busy writing to it. Way too much concurrent writing is done and it indicates an design error in the user program.

write(*column*, *value*, *type_of_value=''*, *only_if_empty=False*)

Write value to 'column' in the database.

If 'column' does not exists, a new is added.

If value is None and type_of_value is not set, the entry under 'column' is set to empty.

For multithreaded and multiprocess programs only a single will process/thread write to the database to avoid too much concurrent writing to the database. This is as long as the 'rank' was passed to SimDB under initialisation.

Parameters

- **column** (*str*) – Name of the column the parameter is read from.
- **value** – New value of the specified entry in the database.
- **type_of_value** (*str or type*) – Needed if column does note exists or if value is empty list. The valid types the strings 'int', 'float', 'bool', 'string' and 'int/float/bool/string array' or the types int, float, bool and str.
- **only_if_empty** (*bool*) – If True, it will only write to the database if the simulation's entry under 'column' is empty.

Raises **ValueError** – If column exists, but type does not match, or empty list is passed without type_of_value given.

unique_results_dir(*path_directory*)

Get path to subdirectory in 'path_directory' unique to simulation.

The subdirectory will be named 'date_time_name_id' and is intended to store results in. If 'results_dir' in the database is empty, a new and unique directory is created and the path stored in 'results_dir'. Otherwise the path in 'results_dir' is just returned.

Parameters **path_directory** (*str*) – Path to directory of which to make a subdirectory. If 'path_directory' starts with 'root/', that part will be replaced by the full path of the root directory of the project.

Returns Full path to new subdirectory.

Return type str

column_exists(*column*)

Return True if column is a column in the database.

Raises **sqlite3.OperationalError** – Waited more than 5 seconds to read from the database, because other threads/processes are busy writing to it. Way too much concurrent writing is done and it indicates an design error in the user program.

is_empty(*column*)

Return True if entry in the database under 'column' is empty.

Raises **sqlite3.OperationalError** – Waited more than 5 seconds to read from the database, because other threads/processes are busy writing to it. Way too much concurrent writing is done and it indicates an design error in the user program.

set_empty(*column*)

Set entry under 'column' in the database to empty.

get_id()

Return 'ID' of the connected simulation.

get_path_proj_root()

Return the path to the root directory of the project.

The project's root directory is assumed to be where the '.sim_db/' directory is located.

update_sha1_executables(*paths_executables*)

Update the 'sha1_executable' column in the database.

Sets the entry to the sha1 of all the executables. The order will affect the value.

Parameters **paths_executables** (*[str]*) – List of full paths to executables.

Raises **sqlite3.OperationalError** – Waited more than 5 seconds to write to the database, because other threads/processes are busy writing to it. Way too much concurrent writing is done and it indicates an design error in the user program.

delete_from_database()

Delete simulation from database.

Raises **sqlite3.OperationalError** – Waited more than 5 seconds to write to the database, because other threads/processes are busy writing to it. Way too much concurrent writing is done and it indicates an design error in the user program.

close()

Closes connection to **sim_db** database and add metadata.

add_empty_sim(*store_metadata=False*)

Add an empty entry into the database and SimDB connected to it.

Parameters **store_metadata** (*bool*) – If False, no metadata is added to the database. Typically used when postprocessing (visualizing) data from a simulation.

CHAPTER

SEVEN

SIM_DB FOR C++

7.1 Minimal Example using C++

A parameter file called *params_mininal_cpp_example.txt* is located in the *sim_db/examples/* directory in the source code. The file contains the following:

```
name (string): minimal_cpp_example

{cmake_config} (alias): cmake -Hroot/ -Broot/examples/build
{cmake_build} (alias): {cmake_config}; cmake --build root/examples/build --target

run_command (string): {cmake_build} minimal_cpp_example; root/examples/build/minimal_
↪cpp_example

param1 (string): "Minimal C++ example is running."

param2 (int): 42
```

A C++ file called *minimal_example.cpp* and is found in the same directory:

```cpp
#include "sim_db.hpp"  // Parts from the standard library is also included.

int main(int argc, char** argv) {
    // Open database and write some initial metadata to database.
    sim_db::Connection sim_db(argc, argv);

    // Read parameters from database.
    auto param1 = sim_db.read<std::string>("param1");
    auto param2 = sim_db.read<int>("param2");

    // Demonstrate that the simulation is running.
    std::cout << param1 << std::endl;
}
```

Add the those simulations parameters to the **sim_db** database and run the simulation from within the *sim_db/examples* directory with:

```
$ sim_db add_and_run -f params_minimal_cpp_example.txt
```

Notice that when it is run, it first call two `cmake` commands to compile the code if needed. What `cmake` does is equvalient to the following command called from *sim_db/examples/* (given that the static C library are compiled and located in *sim_db/build*):

```
$ c++ -o build/minimal_cpp_example minimal_example.cpp -I../include -L../build -
↪lsimdbcpp -lpthread -ldl -m
```

The example is not really a minimal one. If you already have compiled your program into a executable called `program` located in the current directory, the lines starting with `{...} (alias):` can be removed and the `run_command` can be replaced with simpy `run_command (string): ./program`.

7.2 Extensive Example using C++

A parameter file called params_extensive_cpp_example.txt is found in the *sim_db/examples/* directory in the source code. This parameter file contains all the possible types available in addition to some comments:

```
This is a comment, as any line without a colon is a comment.
# Adding a hashtag to the start of a comment line, make the comment easier to␣
↪recognize.

# The name parameter is highly recommended to include.
name (string): extensive_c++_example

# It is also recommended to include a description to further explain the intention of
# the simulation.
description (string): Extensive C++ example to demonstrate most features in sim_db.

# Aliases for cmake commands for compiling the example.
{cmake_config} (alias): cmake -Hroot/ -Broot/examples/build
{cmake_build} (alias): {cmake_config}; cmake --build root/examples/build --target

# This 'run_command' starts with an alias that is replaced with the above two cmake
# commands that compile the extensitve example if needed. The last part of the
# 'run_command' then run the compiled example. Each command is seperated by a
# semicolon, but they all need to be on the same line.
run_command (string): {cmake_build} extensive_cpp_example; root/examples/build/
↪extensive_cpp_example

# A parameter is added for each of the avaiable types.
param1_extensive (int): 3
param2_extensive (float): -0.5e10
param3_extensive (string): "Extensive C++ example is running."
param4_extensive (bool): True
param5_extensive (int array): [1, 2, 3]
param6_extensive (float array): [1.5, 2.5, 3.5]
param7_extensive (string array): ["a", "b", "c"]
param8_extensive (bool array): [True, False, True]

# Include parameters from another parameter file.
include_parameter_file: root/examples/extra_params_example.txt

# Change a parameter value from the included parameter file to demonstrate that
# it is the last parameter value that count for a given parameter name.
extra_param1 (int): 9
```

The line in the parameter file starting with *include_parameter_file:* will be substituted with the contain of the specified *extra_params_example.txt* file, found in the same directory:

```
# Extra parameters included in the extensive examples.

extra_param1 (int): 7
extra_param2 (string): "Extra params added."
extra_param3 (bool): False
```

extensive_example.py is also found in the same directory:

```
#include "sim_db.hpp"  // Parts from the standard library is also included.

int main(int argc, char** argv) {
    // Open database and write some initial metadata to database.
    sim_db::Connection sim_db(argc, argv);

    // Read parameters from database.
    auto param1 = sim_db.read<int>("param1_extensive");
    auto param2 = sim_db.read<double>("param2_extensive");
    auto param3 = sim_db.read<std::string>("param3_extensive");
    auto param4 = sim_db.read<bool>("param4_extensive");
    auto param5 = sim_db.read<std::vector<int> >("param5_extensive");
    auto param6 = sim_db.read<std::vector<double> >("param6_extensive");
    auto param7 = sim_db.read<std::vector<std::string> >("param7_extensive");
    auto param8 = sim_db.read<std::vector<bool> >("param8_extensive");

    // Demonstrate that the simulation is running.
    std::cout << param3 << std::endl;

    // Write all the possible types to database.
    // Only these types are can be written to the database.
    sim_db.write("example_result_1", param1);
    sim_db.write("example_result_2", param2);
    sim_db.write("example_result_3", param3);
    sim_db.write("example_result_4", param4);
    sim_db.write("example_result_5", param5);
    sim_db.write("example_result_6", param6);
    sim_db.write("example_result_7", param7);
    sim_db.write("example_result_8", param8);

    // Make unique subdirectory for storing results and write its name to
    // database. Large results are recommended to be saved in this subdirectory.
    std::string name_results_dir =
            sim_db.unique_results_dir("root/examples/results");

    // Write some results to a file in the newly create subdirectory.
    std::ofstream results_file;
    results_file.open(name_results_dir + "/results.txt");
    for (auto i : param6) {
        results_file << i << std::endl;
    }

    // Check if column exists in database.
    bool is_column_in_database = sim_db.column_exists("column_not_in_database");

    // Check if column is empty and then set it to empty.
    bool is_empty = sim_db.is_empty("example_result_1");
    sim_db.set_empty("example_result_1");
```

(continues on next page)

(continued from previous page)

```
    // Get the 'ID' of the connected simulation an the path to the project's
    // root directory.
    int id = sim_db.get_id();
    std::string path_proj_root = sim_db.get_path_proj_root();

    // Add an empty simulation to the database, open connection and write to it.
    sim_db::Connection sim_db_2 = sim_db::add_empty_sim(path_proj_root, false);
    sim_db_2.write<int>("param1_extensive", 7);

    // Delete simulation from database.
    sim_db_2.delete_from_database();
}
```

Add the those simulations parameters to the **sim_db** database and run the simulation from within the *sim_db/examples* directory with:

```
$ sdb add_and_run -f params_extensive_cpp_example.txt
```

Notice that when it is run, it first call `cmake` to compile the code if needed. What `cmake` does is equvalient to the following command called from *sim_db/examples/* (given that the static C library are compiled and located in *sim_db/build/*):

```
$ cc -o build/extensive_cpp_example extensive_example.cpp -I../include -L../build -
↪lsimdbcpp -lpthread -ldl -m
```

7.3 C++ API Referance

class Connection

To interact with the **sim_db** database.

For an actuall simulation it should be initialised at the very start of the simulation (with 'store_metadata' set to True) in a scope that last the entirety of the simulation. This must be done to add the corrrect metadata.

For multithreading/multiprocessing each thread/process MUST have its own connection (instance of this class).

> **Warning:** doxygenfunction: Unable to resolve multiple matches for function "sim_db::Connection::Connection" with arguments (int, char**, bool) in doxygen xml output for project "sim_db" from directory: /home/docs/checkouts/readthedocs.org/user_builds/sim-db/checkouts/latest/docs/xml/. Potential matches:
>
> ```
> - sim_db::Connection::Connection(int argc, char **argv, bool store_metadata = true)
> - sim_db::Connection::Connection(int id, bool store_metadata = true)
> - sim_db::Connection::Connection(std::string path_proj_root, int id, bool store_
> ↪metadata = true)
> ```

> **Warning:** doxygenfunction: Unable to resolve multiple matches for function "sim_db::Connection::Connection" with arguments (std::string, int, bool) in doxygen xml output for project "sim_db" from directory: /home/docs/checkouts/readthedocs.org/user_builds/sim-db/checkouts/latest/docs/xml/. Potential matches:
>
> ```
> - sim_db::Connection::Connection(int argc, char **argv, bool store_metadata = true)
> - sim_db::Connection::Connection(int id, bool store_metadata = true)
> - sim_db::Connection::Connection(std::string path_proj_root, int id, bool store_
> ↪metadata = true)
> ```

template<typename **T**>
T `sim_db::`*`Connection`*`::`**`read`**(std::string *column*)

Read parameter from database.

Return Parameter read from database.

Parameters

- `column`: Name of the parameter and column in the database.

Exceptions

- `std::invalid_argument`: `column` not a column in the database.
- `sim_db::TimeoutError`: Waited more than 5 seconds to read from the database, because other threads/processes are busy writing to it. Way too much concurrent writing is done and it indicates an design error in the user program.

template<typename **T**>
void `sim_db::`*`Connection`*`::`**`write`**(std::string *column*, *T* *value*, bool *only_if_empty* = false)

Write `value` to database.

Parameters

- `column`: Name of the parameter and column in the database.
- `value`: To be written to database.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

Exceptions

- `sim_db::TimeoutError`: Waited more than 5 seconds to write to the database because other threads/processes are busy writing to it. Way too much concurrent writing is done and indicates an design error in the user program.

std::string `sim_db::`*`Connection`*`::`**`unique_results_dir`**(std::string *path_directory*)

Get path to subdirectory in `path_directory` unique to simulation.

The subdirectory will be named 'date_time_name_id' and is intended to store results in. If 'results_dir' in the database is empty, a new and unique directory is created and the path stored in 'results_dir'. Otherwise the path in 'results_dir' is just returned.

Return Path to new subdirectory.

Parameters

- `path_directory`: Path to where the new directory is created. If it starts with 'root/', that part will be replaced with the full path to the root directory of the project.

bool `sim_db::`*`Connection`*`::`**`column_exists`**(std::string *column*)

Return true if `column` is a column in the database.

Exceptions

- `sim_db::TimeoutError`: Waited more than 5 seconds to write to the database because other threads/processes are busy writing to it. Way too much concurrent writing is done and indicates an design error

int `sim_db::Connection::get_id()`
Return ID number of simulation in the database that is connected.

std::string `sim_db::Connection::get_path_proj_root()`
Return path to root directory of the project, where *.sim_db/* is located.

void `sim_db::Connection::update_sha1_executables`(std::vector<std::string> *paths_executables*, bool *only_if_empty* = false)
Save the sha1 hash of the file `paths_executables` to the database.

Parameters

- `paths_executables`: Paths to executable files.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

Exceptions

- `sim_db::TimeoutError`: Waited more than 5 seconds to write to the database because other threads/processes are busy writing to it. Way too much concurrent writing is done and indicates an design error in the user program.

void `sim_db::Connection::delete_from_database()`
Delete simulation from database.

Warning: doxygenfunction: Unable to resolve multiple matches for function "sim_db::add_empty_sim" with arguments (bool) in doxygen xml output for project "sim_db" from directory: /home/docs/checkouts/readthedocs.org/user_builds/sim-db/checkouts/latest/docs/xml/. Potential matches:

```
- Connection sim_db::add_empty_sim(bool store_metadata = false)
- Connection sim_db::add_empty_sim(std::string path_proj_root, bool store_metadata
↪= false)
```

Warning: doxygenfunction: Unable to resolve multiple matches for function "sim_db::add_empty_sim" with arguments (std::string, bool) in doxygen xml output for project "sim_db" from directory: /home/docs/checkouts/readthedocs.org/user_builds/sim-db/checkouts/latest/docs/xml/. Potential matches:

```
- Connection sim_db::add_empty_sim(bool store_metadata = false)
- Connection sim_db::add_empty_sim(std::string path_proj_root, bool store_metadata
↪= false)
```

CHAPTER

EIGHT

SIM_DB FOR C

8.1 Minimal Example using C

A parameter file called *params_mininal_c_example.txt* is located in the *sim_db/examples/* directory in the source code. The file contains the following:

```
name (string): minimal_c_example

{cmake_config} (alias): cmake -Hroot/ -Broot/examples/build
{cmake_build} (alias): {cmake_config}; cmake --build root/examples/build --target

run_command (string): {cmake_build} minimal_c_example; root/examples/build/minimal_c_
↪example

param1 (string): "Minimal C example is running."

param2 (int): 42
```

A C file called *minimal_example.c* and is found in the same directory:

```
#include "sim_db.h"  // Parts from the standard library is also included.

int main(int argc, char** argv) {
    // Open database and write some initial metadata to database.
    SimDB* sim_db = sim_db_ctor(argc, argv);

    // Read parameters from database.
    char* param1 = sim_db_read_string(sim_db, "param1");
    int param2 = sim_db_read_int(sim_db, "param2");

    // Demonstrate that the simulation is running.
    printf("%s\n", param1);

    // Write final metadata to database and free memory allocated by sim_db.
    sim_db_dtor(sim_db);
}
```

Add the those simulations parameters to the **sim_db** database and run the simulation from within the *sim_db/examples* directory with:

```
$ sim_db add_and_run -f params_minimal_c_example.txt
```

Notice that when it is run, it first call two `cmake` commands to compile the code if needed. What `cmake` does is equvalient to the following command called from *sim_db/examples/* (given that the static C library are compiled and located in *sim_db/build/*):

```
$ cc -o build/minimal_c_example minimal_example.c -I../include -L../build -lsimdbc -
↪lpthread -ldl -m
```

The example is not really a minimal one. If you already have compiled your program into a executable called `program` located in the current directory, the lines starting with `{...} (alias):` can be removed and the `run_command` can be replaced with simpy `run_command (string): ./program`.

8.2 Extensive Example using C

A parameter file called params_extensive_c_example.txt is found in the *sim_db/examples/* directory in the source code. This parameter file contains all the possible types available in addition to some comments:

```
This is a comment, as any line without a colon is a comment.
# Adding a hashtag to the start of a comment line, make the comment easier to␣
↪recognize.

# The name parameter is highly recommended to include.
name (string): extensive_c_example

# It is also recommended to include a description to further explain the intention of
# the simulation.
description (string): Extensive C example to demonstrate most features in sim_db.

# Aliases for cmake commands for compiling the example.
{cmake_config} (alias): cmake -Hroot/ -Broot/examples/build
{cmake_build} (alias): {cmake_config}; cmake --build root/examples/build --target

# This 'run_command' starts with an alias that is replaced with the above two cmake
# commands that compile the extensitve example if needed. The last part of the
# 'run_command' then run the compiled example. Each command is seperated by a
# semicolon, but they all need to be on the same line.
run_command (string): {cmake_build} extensive_c_example; root/examples/build/
↪extensive_c_example

# A parameter is added for each of the avaiable types.
param1_extensive (int): 3
param2_extensive (float): -0.5e10
param3_extensive (string): "Extensive C example is running."
param4_extensive (bool): True
param5_extensive (int array): [1, 2, 3]
param6_extensive (float array): [1.5, 2.5, 3.5]
param7_extensive (string array): ["a", "b", "c"]
param8_extensive (bool array): [True, False, True]

# Include parameters from another parameter file.
include_parameter_file: root/examples/extra_params_example.txt

# Change a parameter value from the included parameter file to demonstrate that
# it is the last parameter value that count for a given parameter name.
extra_param1 (int): 9
```

The line in the parameter file starting with *include_parameter_file:* will be substituted with the contain of the specified *extra_params_example.txt* file, found in the same directory:

```
# Extra parameters included in the extensive examples.

extra_param1 (int): 7
extra_param2 (string): "Extra params added."
extra_param3 (bool): False
```

extensive_example.py is also found in the same directory:

```
#include "sim_db.h"  // Parts from the standard library is also included.

int main(int argc, char** argv) {
    // Open database and write some initial metadata to database.
    SimDB* sim_db = sim_db_ctor(argc, argv);

    // Read parameters from database.
    int param1 = sim_db_read_int(sim_db, "param1_extensive");
    double param2 = sim_db_read_double(sim_db, "param2_extensive");
    char* param3 = sim_db_read_string(sim_db, "param3_extensive");
    bool param4 = sim_db_read_bool(sim_db, "param4_extensive");
    SimDBIntVec param5 = sim_db_read_int_vec(sim_db, "param5_extensive");
    SimDBDoubleVec param6 = sim_db_read_double_vec(sim_db, "param6_extensive");
    SimDBStringVec param7 = sim_db_read_string_vec(sim_db, "param7_extensive");
    SimDBBoolVec param8 = sim_db_read_bool_vec(sim_db, "param8_extensive");

    // Show that SimDBIntVec contain array of integers and size.
    int* int_array = param5.array;
    int size_int_array = param5.size;

    // Demonstrate that the simulation is running.
    printf("%s\n", param3);

    // Write all the possible types to database.
    // Only these types are can be written to the database.
    sim_db_write_int(sim_db, "example_result_1", param1, false);
    sim_db_write_double(sim_db, "example_result_2", param2, true);
    sim_db_write_string(sim_db, "example_result_3", param3, false);
    sim_db_write_bool(sim_db, "example_result_4", param4, true);
    sim_db_write_int_array(sim_db, "example_result_5", param5.array,
                           param5.size, false);
    sim_db_write_double_array(sim_db, "example_result_6", param6.array,
                              param6.size, true);
    sim_db_write_string_array(sim_db, "example_result_7", param7.array,
                              param7.size, false);
    sim_db_write_bool_array(sim_db, "example_result_8", param8.array,
                            param8.size, true);

    // Make unique subdirectory for storing results and write its name to
    // database. Large results are recommended to be saved in this subdirectory.
    char* name_subdir =
            sim_db_unique_results_dir(sim_db, "root/examples/results");

    // Write some results to a file in the newly create subdirectory.
    FILE* result_file = fopen(strcat(name_subdir, "/results.txt"), "w");
    for (size_t i = 0; i < param6.size; i++) {
        fprintf(result_file, "%f\n", param6.array[i]);
    }
    fclose(result_file);
```

(continues on next page)

(continued from previous page)

```
    // Check if column exists in database.
    bool is_column_in_database =
            sim_db_column_exists(sim_db, "column_not_in_database");

    // Check if column is empty and then set it to empty.
    bool is_empty = sim_db_is_empty(sim_db, "example_result_1");
    sim_db_set_empty(sim_db, "example_result_1");

    // Get the 'ID' of the connected simulation and the path to the project's
    // root directoy.
    int id = sim_db_get_id(sim_db);
    char path_proj_root[PATH_MAX + 1];
    strcpy(path_proj_root, sim_db_get_path_proj_root(sim_db));

    // Write final metadata to the database, close the connection and free
    // memory allocated by sim_db.
    sim_db_dtor(sim_db);

    // Add an empty simulation to the database, open connection and write to it.
    SimDB* sim_db_2 =
            sim_db_add_empty_sim_without_search(path_proj_root, false);
    sim_db_write_int(sim_db_2, "param1_extensive", 7, false);

    // Delete simulation from database.
    sim_db_delete_from_database(sim_db_2);

    // Close connection to the database and free memory allocated by sim_db.
    sim_db_dtor(sim_db_2);
}
```

Add the those simulations parameters to the **sim_db** database and run the simulation from within the *sim_db/examples* directory with:

```
$ sdb add_and_run -f params_extensive_c_example.txt
```

Notice that when it is run, it first call `cmake` to compile the code if needed. What `cmake` does is equvalient to the following command called from *sim_db/examples/* (given that the static C library are compiled and located in *sim_db/build/*):

```
$ cc -o build/extensive_c_example extensive_example.c -I../include -L../build -
↪lsimdbc -lpthread -ldl -m
```

8.3 C API Referance

Warning: doxygenfunction: Unable to resolve multiple matches for function “sim_db_ctor” with arguments (int, char**) in doxygen xml output for project “sim_db” from directory: /home/docs/checkouts/readthedocs.org/user_builds/sim-db/checkouts/latest/docs/xml/. Potential matches:

```
- SimDB *sim_db_ctor(int argc, char **argv)
- type(sim_db) function sim_db_mod::sim_db::sim_db_ctor (store_metadata)
- type(sim_db) function sim_db_mod::sim_db_ctor (store_metadata)
```

SimDB ***sim_db_ctor_no_metadata**(int *argc*, char ***argv*)

Initialize SimDB and connect to the **sim_db** database.

No metadata store automatically, and only explicit calls will write to the database. Should be used instead of sim_db_ctor() for postprocessing.

sim_db_dtor(SimDB*) MUST be called to clean up.

For multithreading/multiprocessing each thread/process MUST have its own connection.

Parameters

- `argc`: Length of `argv`.
- `argv`: Array of command line arguments containing `--id 'ID'` and optionally `--path_proj_root 'PATH'`. *PATH* is the root directory of the project, where *.sim_db/* is located. If not passed, the current working directory and its parent directories will be searched until *.sim_db/* is found.

Warning: doxygenfunction: Unable to resolve multiple matches for function "sim_db_ctor_with_id" with arguments (int, bool) in doxygen xml output for project "sim_db" from directory: /home/docs/checkouts/readthedocs.org/user_builds/sim-db/checkouts/latest/docs/xml/. Potential matches:

```
- SimDB *sim_db_ctor_with_id(int id, bool store_metadata)
- type(sim_db) function sim_db_mod::sim_db::sim_db_ctor_with_id (id, store_metadata)
- type(sim_db) function sim_db_mod::sim_db_ctor_with_id (id, store_metadata)
```

Warning: doxygenfunction: Unable to resolve multiple matches for function "sim_db_ctor_without_search" with arguments (const char*, int, bool) in doxygen xml output for project "sim_db" from directory: /home/docs/checkouts/readthedocs.org/user_builds/sim-db/checkouts/latest/docs/xml/. Potential matches:

```
- SimDB *sim_db_ctor_without_search(const char *path_proj_root, int id, bool store_
↪metadata)
- type(sim_db) function sim_db_mod::sim_db::sim_db_ctor_without_search (path_proj_
↪root, id, store_metadata)
- type(sim_db) function sim_db_mod::sim_db_ctor_without_search (path_proj_root, id,␣
↪store_metadata)
```

int **sim_db_read_int**(SimDB **self*, **const** char **column*)

Read parameter from the database.

Return Integer read from database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the parameter and column in the database.

double **sim_db_read_double**(SimDB **self*, **const** char **column*)

Read parameter from the database.

Return Double read from database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the parameter and column in the database.

char ***sim_db_read_string**(SimDB **self*, **const** char **column*)
Read parameter from the database.

Return String read from database. Do NOT free the string, as sim_db_dtor() will do that.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the parameter and column in the database.

bool **sim_db_read_bool**(SimDB **self*, **const** char **column*)
Read parameter from the database.

Return Bool read from database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the parameter and column in the database.

struct SimDBIntVec
Vector of integers.

Public Members

size_t **size**
Length of array.

int ***array**
Array of integers.

SimDBIntVec **sim_db_read_int_vec**(SimDB **self*, **const** char **column*)
Read parameter from the database.

Return Vector of integers read from database. Do NOT free array as sim_db_dtor() will do that.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the parameter and column in the database.

struct SimDBDoubleVec
Vector of doubles.

Public Members

size_t **size**
Length of array.

double ***array**
Array of doubles.

SimDBDoubleVec **sim_db_read_double_vec**(SimDB **self*, **const** char **column*)
Read parameter from the database.

Return Vector of doubles read from database. Do NOT free array as sim_db_dtor() will do that.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the parameter and column in the database.

struct SimDBStringVec
Vector of strings.

Public Members

size_t **size**
Length of array.

char ****array**
Array of strings.

SimDBStringVec **sim_db_read_string_vec**(SimDB **self*, **const** char **column*)
Read parameter from the database.

Return Vector of strings read from database. Do NOT free array as sim_db_dtor() will do that.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the parameter and column in the database.

struct SimDBBoolVec
Vector of booleans.

Public Members

size_t **size**
Length of array.

bool ***array**
Array of booleans.

SimDBBoolVec **sim_db_read_bool_vec**(SimDB **self*, **const** char **column*)
Read parameter from the database.

Return Vector of booleans read from database. Do NOT free array as sim_db_dtor() will do that.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.

- `column`: Name of the parameter and column in the database.

void **sim_db_write_int** (SimDB **self*, **const** char **column*, int *value*, bool *only_if_empty*)
Write `value` to database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the column in the database to write to.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid any potential timeouts for concurrect applications.

void **sim_db_write_double** (SimDB **self*, **const** char **column*, double *value*, bool *only_if_empty*)
Write `value` to database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the column in the database to write to.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid any potential timeouts for concurrect applications.

void **sim_db_write_string** (SimDB **self*, **const** char **column*, **const** char **value*, bool *only_if_empty*)
Write `value` to database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the column in the database to write to.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid any potential timeouts for concurrect applications.

void **sim_db_write_bool** (SimDB **self*, **const** char **column*, bool *value*, bool *only_if_empty*)
Write `value` to database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the column in the database to write to.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid any potential timeouts for concurrect applications.

void **sim_db_write_int_array** (SimDB **self*, **const** char **column*, int **arr*, size_t *len*, bool *only_if_empty*)
Write `arr` to database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the column in the database to write to.
- `arr`: Array to be written to simulation database.

- `len`: Length of `arr`.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid any potential timeouts for concurrect applications.

void **sim_db_write_double_array**(SimDB **self*, **const** char **column*, double **arr*, size_t *len*, bool *only_if_empty*)

Write `arr` to database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the column in the database to write to.
- `arr`: Array to be written to simulation database.
- `len`: Length of `arr`.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid any potential timeouts for concurrect applications.

void **sim_db_write_string_array**(SimDB **self*, **const** char **column*, char ***arr*, size_t *len*, bool *only_if_empty*)

Write `arr` to database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the column in the database to write to.
- `arr`: Array to be written to simulation database.
- `len`: Length of `arr`.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid any potential timeouts for concurrect applications.

void **sim_db_write_bool_array**(SimDB **self*, **const** char **column*, bool **arr*, size_t *len*, bool *only_if_empty*)

Write `arr` to database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `column`: Name of the column in the database to write to.
- `arr`: Array to be written to simulation database.
- `len`: Length of `arr`.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'column' is empty. Will avoid any potential timeouts for concurrect applications.

char ***sim_db_unique_results_dir**(SimDB **self*, **const** char **path_to_dir*)

Get path to subdirectory in `abs_path_to_dir` unique to simulation.

The subdirectory will be named 'date_time_name_id' and is intended to store results in. If 'results_dir' in the database is empty, a new and unique directory is created and the path stored in 'results_dir'. Otherwise the path in 'results_dir' is just returned.

Return Path to new subdirectory.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `path_to_dir`: Path to where the new directory is created. If it starts with 'root/', that part will be replaced with the full path to the root directory of the project.

char ***sim_db_unique_results_dir_abs_path**(SimDB **self*, **const** char **abs_path_to_dir*)

Get path to subdirectory in `abs_path_to_dir` unique to simulation.

The subdirectory will be named 'date_time_name_id' and is intended to store results in. If 'results_dir' in the database is empty, a new and unique directory is created and the path stored in 'results_dir'. Otherwise the path in 'results_dir' is just returned.

Return Path to new subdirectory.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `abs_path_to_dir`: Absolute path to where the new directory is created.

bool **sim_db_column_exists**(SimDB **self*, **const** char **column*)

Return true if `column` is a column in the database.

int **sim_db_get_id**(SimDB **self*)

Return ID number of simulation in the database that is connected.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.

char ***sim_db_get_path_proj_root**(SimDB **self*)

Return path to root directory of the project, where *.sim_db/* is located.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.

void **sim_db_update_sha1_executables**(SimDB **self*, char ***paths_executables*, size_t *len*, bool *only_if_empty*)

Save the sha1 hash of the files `paths_executables` to the database.

Parameters

- `self`: Return value of sim_db_ctor() or similar functions.
- `paths_executables`: Paths to executable files.
- `len`: Length of `paths_executables`.
- `only_if_empty`: If True, it will only write to the database if the simulation's entry under 'sha1_executables' is empty. Will avoid any potential timeouts for concurrect applications.

void **sim_db_allow_timeouts**(SimDB **self*, bool *allow_timeouts*)

Allow timeouts to occure without exiting if set to true.

A timeout occures after waiting more than 5 seconds to access the database because other threads/processes are busy writing to it. **sim_db** will exit with an error in that case, unless allow timeouts is set to true. It is false by default. If allowed and a timeout occures the called funciton will have had no effect.

bool **sim_db_have_timed_out** (SimDB **self*)

Checks if a timeout have occured since last call to this function.

void **sim_db_delete_from_database** (SimDB **self*)

Delete simulation from database.

final sim_db_mod::sim_db::sim_db_dtor

SimDB ***sim_db_add_empty_sim** (bool *store_metadata*)

Add empty simulation to database and return a SimDB connected to it.

The current working directory and its parent directories will be searched until *.sim_db/* is found.

Return SimDB of the added simulation.

Parameters

- `store_metadata`: Stores metadata if true. Set to 'false' for postprocessing (e.g. visualization) of data from simulation.

Warning: doxygenfunction: Unable to resolve multiple matches for function "sim_db_add_empty_sim_without_search" with arguments (const char*, bool) in doxygen xml output for project "sim_db" from directory: /home/docs/checkouts/readthedocs.org/user_builds/sim-db/checkouts/latest/docs/xml/. Potential matches:

```
- SimDB *sim_db_add_empty_sim_without_search(const char *path_proj_root, bool store_
↪metadata)
- type(sim_db) function sim_db_mod::sim_db_add_empty_sim_without_search (path_proj_
↪root, store_metadata)
```

CHAPTER NINE

SIM_DB FOR FORTRAN

9.1 Minimal Example using Fortran

A parameter file called *params_mininal_fortran_example.txt* is located in the *sim_db/examples/* directory in the source code. The file contains the following:

```
name (string): minimal_fortran_example

{cmake_config} (alias): cmake -Hroot/ -Broot/examples/build -DFortran=ON
{cmake_build} (alias): {cmake_config}; cmake --build root/examples/build --target

run_command (string): {cmake_build} minimal_fortran_example; root/examples/build/
↪minimal_fortran_example

param1 (string): "Minimal Fortran example is running."

param2 (int): 42
```

A Fortran file called *minimal_example.f90* and is found in the same directory:

```fortran
program minimal_example
    use sim_db_mod
    implicit none

    type(sim_db) :: sim_database
    character(len=:), allocatable :: param1
    integer :: param2

    ! Open database and write some initial metadata to database.
    sim_database = sim_db()

    ! Read parameters from database.
    call sim_database%read("param1", param1)
    call sim_database%read("param2", param2)

    ! Demonstrate that the simulation is running.
    print *, param1

    ! Write final metadata to database and close connection.
    call sim_database%close()
end program minimal_example
```

Add the those simulations parameters to the **sim_db** database and run the simulation from within the *sim_db/examples* directory with:

```
$ sim_db add_and_run -f params_minimal_cpp_example.txt
```

Notice that when it is run, it first call two `cmake` commands to compile the code if needed. What `cmake` does is equvalient to the following command called from *sim_db/examples/* (given that the static Fortran library are compiled and located in *sim_db/build*):

```
$ gfortran -o build/minimal_fortran_example minimal_example.f90 -I../build -L../build␣
↪-lsimdbf -lpthread -ldl -m
```

The example is not really a minimal one. If you already have compiled your program into a executable called `program` located in the current directory, the lines starting with `{...} (alias):` can be removed and the `run_command` can be replaced with simpy `run_command (string): ./program`.

9.2 Extensive Example using Fortran

A parameter file called params_extensive_cpp_example.txt is found in the *sim_db/examples/* directory in the source code. This parameter file contains all the possible types available in addition to some comments:

```
This is a comment, as any line without a colon is a comment.
# Adding a hashtag to the start of a comment line, make the comment easier to␣
↪recognize.

# The name parameter is highly recommended to include.
name (string): extensive_fortran_example

# It is also recommended to include a description to further explain the intention of
# the simulation.
description (string): Extensive Fortran example to demonstrate most features in sim_
↪db.

# Aliases for cmake commands for compiling the example.
{cmake_config} (alias): cmake -Hroot/ -Broot/examples/build -DFortran=ON
{cmake_build} (alias): {cmake_config}; cmake --build root/examples/build --target

# This 'run_command' starts with an alias that is replaced with the above two cmake
# commands that compile the extensitve example if needed. The last part of the
# 'run_command' then run the compiled example. Each command is seperated by a
# semicolon, but they all need to be on the same line.
run_command (string): {cmake_build} extensive_fortran_example; root/examples/build/
↪extensive_fortran_example

# A parameter is added for each of the avaiable types.
param1_extensive (int): 3
param2_extensive (float): -0.5e10
param3_extensive (string): "Extensive C++ example is running."
param4_extensive (bool): True
param5_extensive (int array): [1, 2, 3]
param6_extensive (float array): [1.5, 2.5, 3.5]
param7_extensive (string array): ["a", "b", "c"]
param8_extensive (bool array): [True, False, True]

# Include parameters from another parameter file.
include_parameter_file: root/examples/extra_params_example.txt

# Change a parameter value from the included parameter file to demonstrate that
```

(continues on next page)

(continued from previous page)

```
# it is the last parameter value that count for a given parameter name.
extra_param1 (int): 9
```

The line in the parameter file starting with *include_parameter_file:* will be substituted with the contain of the specified *extra_params_example.txt* file, found in the same directory:

```
# Extra parameters included in the extensive examples.

extra_param1 (int): 7
extra_param2 (string): "Extra params added."
extra_param3 (bool): False
```

extensive_example.py is also found in the same directory:

```
program extensive_example
    use sim_db_mod
    implicit none

    type(sim_db) :: sim_database, sim_database_2
    integer :: param1, i, id
    real :: param2
    real(kind=kind(1.0d0)) :: param2_dp
    character(len=:), allocatable :: param3, name_results_dir, path_proj_root
    logical :: param4, is_column_in_database, is_empty
    integer, dimension(:), allocatable :: param5
    real, dimension(:), allocatable :: param6
    real(kind=kind(1.0d0)), dimension(:), allocatable :: param6_dp
    character(len=:), dimension(:), allocatable :: param7
    logical, dimension(:), allocatable :: param8

    ! Open database and write some inital metadata to database.
    sim_database = sim_db()

    ! Read parameters from database.
    call sim_database%read("param1_extensive", param1)
    call sim_database%read("param2_extensive", param2)
    call sim_database%read("param2_extensive", param2_dp)
    call sim_database%read("param3_extensive", param3)
    call sim_database%read("param4_extensive", param4)
    call sim_database%read("param5_extensive", param5)
    call sim_database%read("param6_extensive", param6)
    call sim_database%read("param6_extensive", param6_dp)
    call sim_database%read_string_array("param7_extensive", param7)
    call sim_database%read("param8_extensive", param8)

    ! Demostrate that the simulation is running.
    print *, param3

    ! Write all the possible types to database.
    ! Only these types are can be written to the database.
    call sim_database%write("example_result_1", param1)
    call sim_database%write("example_result_2", param2)
    call sim_database%write("example_result_2", param2_dp)
    call sim_database%write("example_result_3", param3)
    call sim_database%write("example_result_4", param4)
    call sim_database%write("example_result_5", param5)
```

(continues on next page)

(continued from previous page)

```
    call sim_database%write("example_result_6", param6)
    call sim_database%write("example_result_6", param6_dp)
    call sim_database%write("example_result_7", param7)
    call sim_database%write("example_result_8", param8)


    ! Make unique subdirectory for storing results and write its name to
    ! database. Large results are recommended to be saved in this subdirectory.
    name_results_dir = sim_database%unique_results_dir("root/examples/results")

    ! Write some results to a file in the newly create subdirectory.
    open(1, file=name_results_dir // "/results.txt")
    do i = 1, size(param6)
        write (1,*) param6(i)
    end do

    ! Check if column exists in database.
    is_column_in_database = sim_database%column_exists("column_not_in_database")

    ! Check if column is empty and then set it to empty.
    is_empty = sim_database%is_empty("example_results_1")
    call sim_database%set_empty("example_result_1")

    ! Get the 'ID' of the connected simulation an the path to the project's
    ! root directory.
    id = sim_database%get_id()
    path_proj_root = sim_database%get_path_proj_root()

    ! Add an empty simulation to the database, open connection and write to it.
    sim_database_2 = sim_db_add_empty_sim(path_proj_root);
    call sim_database_2%write("param1_extensive", 7)

    ! Delete simulation from database.
    call sim_database_2%delete_from_database()

    ! Write final metadata to database and close connection.
    ! (Final method (destructor) is not called at end of program, so close()
    !  MUST be called manually. It is always recommended to call close()
    !  explicitly to avoid unexpected, because of this.)
    call sim_database%close()
end program extensive_example
```

Add the those simulations parameters to the **sim_db** database and run the simulation from within the *sim_db/examples* directory with:

```
$ sdb add_and_run -f params_extensive_fortran_example.txt
```

Notice that when it is run, it first call `cmake` to compile the code if needed. What `cmake` does is equvalient to the following command called from *sim_db/examples/* (given that the static Fortran library are compiled and located in *sim_db/build/*):

```
$ gfortran -o build/extensive_fortran_example extensive_example.f90 -I../build -L../
↪build -lsimdbf -lpthread -ldl -m
```

9.3 Fortran API Referance

Doxygen and Breathe that is used to generate the documentation for C, C++ and Fortran, does not work as well for Fortran as it does for C and C++. The Fortran documentation is therefor less than ideal, but hopefully still somewhat useful, espesially in combination with the examples. The parameter types are written in bold at the start of the parameter description.

`interface sim_db`

Class to interact with the *sim_db* database.

Constructor is an overload of the **sim_db_dtor*** functions below, which gives the valid types of parameters.

Should be called at the very begin of the simulation and **close** should be called at the very end to add the correct metadata and to clean up.

For multithreading/multiprocessing each thread/process MUST have its own connection (instance of this class).

`type(sim_db) function sim_db_mod::sim_db_ctor (store_metadata)`

Connect to the database using the command line arguments containing `--id 'ID'` and optionally `--path_proj_root 'PATH'`. *PATH* is the root directory of the project, where *.sim_db/* is located. If not passed, the current working directory and its parent directories will be searched until *.sim_db/* is found.

Parameters

- `store_metadata`: **logical, optional** Stores metadata to database if true. Set to 'false' for postprocessing (e.g. visualization) of data from simulation.

`type(sim_db) function sim_db_mod::sim_db_ctor_with_id (id, store_metadata)` **Parameters**

- `id`: **integer, intent(in)** ID number of the simulation paramters in the *sim_db* database.
- `store_metadata`: **logical, optional** Stores metadata to database if true. Set to 'false' for postprocessing (e.g. visualization) of data from simulation.

`type(sim_db) function sim_db_mod::sim_db_ctor_without_search (path_proj_root, id, store_met` **Parameters**

- `path_proj_root`: **character(len=*), intent(in)** Path to the root directory of the project, where *.sim_db/* is located.
- `id`: **integer, intent(in)** ID number of the simulation paramters in the *sim_db* database.
- `store_metadata`: **logical, optional** Stores metadata to database if true. Set to 'false' for postprocessing (e.g. visualization) of data from simulation.

`generic sim_db_mod::sim_db::read => read_int, read_real_sp, read_real_dp, read_string, reac`

Read parameter from database.

Overload of the **read_*** subroutines below, which gives the valid types of parameters.

subroutine `sim_db_mod::`**`read_int`**(self, column, int_value)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `[out] int_value`: **integer, intent(out)** Value read from database.

subroutine `sim_db_mod::`**`read_real_sp`**(self, column, real_value)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.

- `[out] real_value`: **real, intent(out)** Value read from database.

subroutine `sim_db_mod::`**`read_real_dp`**(self, column, real_value)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `[out] real_value`: **real(kind=kind(1.0d0)), intent(out)** Value read from database.

subroutine `sim_db_mod::`**`read_string`**(self, column, string_value)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `[out] string_value`: **character(len=:), allocatable, intent(out)** Value read from database.

subroutine `sim_db_mod::`**`read_logical`**(self, column, logical_value)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `[out] logical_value`: **logical, intent(out)** Value read from database.

subroutine `sim_db_mod::`**`read_int_array`**(self, column, int_array)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `[out] int_array`: **integer, dimension(:), allocatable, intent(out)** Value read from database.

subroutine `sim_db_mod::`**`read_real_sp_array`**(self, column, real_array)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `[out] real_array`: **real, dimension(:), allocatable, intent(out)** Value read from database.

subroutine `sim_db_mod::`**`read_real_dp_array`**(self, column, real_array)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `[out] real_array`: **real(kind=kind(1.0d0)), dimension(:), allocatable, intent(out)** Value read from database.

subroutine `sim_db_mod::`**`read_string_array`**(self, column, string_array)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `[out] **allocatable[out] intent(out)`: :: string_array character(len(:), dimension(** Value read from databas.

subroutine `sim_db_mod::`**`read_logical_array`**(self, column, logical_array)

Parameters

- `[in] column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `[out] logical_array`: **logical, dimension(:), allocatable, intent(out)** Value read from database.

generic sim_db_mod::sim_db::write => write_int, write_real_sp, write_real_dp, write_string,

Write parameter to database.

Overload of the **write_*** below, which gives the valid types of parameters.

subroutine `sim_db_mod::`**`write_int`**`(self, column, int_value, only_if_empty)`

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `int_value`: **integer, intent(in)** To be written to database.
- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_int_false`**`(self, column, int_value)`

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `int_value`: **integer, intent(in)** To be written to database.

subroutine `sim_db_mod::`**`write_real_sp`**`(self, column, real_value, only_if_empty)`

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `real_value`: **real, intent(in)** To be written to database.
- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_real_sp_false`**`(self, column, real_value)`

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `real_value`: **real, intent(in)** To be written to database.

subroutine `sim_db_mod::`**`write_real_dp`**`(self, column, real_value, only_if_empty)`

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `real_value`: **real(kind=kind(1.0d0)), intent(in)** To be written to database.
- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_real_dp_false`**`(self, column, real_value)`

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `real_value`: **real(kind=kind(1.0d0)), intent(in)** To be written to database.

subroutine `sim_db_mod::`**`write_string`**`(self, column, string_value, only_if_empty)`

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `string_value`: **character(len=*), intent(in)** To be written to database.

- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_string_false`**(self, column, string_value)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `string_value`: **character(len=*), intent(in)** To be written to database.

subroutine `sim_db_mod::`**`write_logical`**(self, column, logical_value, only_if_empty)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `logical_value`: **logical, intent(in)** To be written to database.
- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_logical_false`**(self, column, logical_value)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `logical_value`: **logical, intent(in)** To be written to database.

subroutine `sim_db_mod::`**`write_int_array`**(self, column, int_array, only_if_empty)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `int_array`: **integer, dimension(:), intent(in)** To be written to database.
- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_int_array_false`**(self, column, int_array)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `int_array`: **integer, dimension(:), target, intent(in)** To be written to database.

subroutine `sim_db_mod::`**`write_real_sp_array`**(self, column, real_array, only_if_empty)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `real_array`: **real, dimension(:), intent(in)** To be written to database.
- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_real_sp_array_false`**(self, column, real_array)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `real_array`: **real, dimension(:), intent(in)** To be written to database.

subroutine `sim_db_mod::`**`write_real_dp_array`**(self, column, real_array, only_if_empty)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `real_array`: **real(kind=kind(1.0d0)), dimension(:), intent(in)** To be written to database.
- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_real_dp_array_false`**(self, column, real_array)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `real_array`: **real(kind=kind(1.0d0)), dimension(:), target, intent(in)** To be written to database.

subroutine `sim_db_mod::`**`write_string_array`**(self, column, string_array, only_if_empty)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `string_array`: **character(len=*), dimension(:), intent(in)** n=*), dimension**(To be written to database.
- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_string_array_false`**(self, column, string_array)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `intent`: **character(len=*), dimension**(To be written to database.

subroutine `sim_db_mod::`**`write_logical_array`**(self, column, logical_array, only_if_empty)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `logical_array`: **logical, dimension(:), intent(in)** To be written to database.
- `only_if_empty`: **logical** If .true., it will only write to the database if the simulation's entry under 'column' is empty. Will avoid potential timeouts for concurrect applications.

subroutine `sim_db_mod::`**`write_logical_array_false`**(self, column, logical_array)

Parameters

- `column`: **character(len=*), intent(in)** Name of the parameter and column in the database.
- `logical_array`: **logical, dimension(:), intent(in)** To be written to database.

`character(len=:) function, allocatable sim_db_mod::unique_results_dir (self, path_to_dir)`

Get path to subdirectory in `path_directory` unique to simulation.

The subdirectory will be named 'date_time_name_id' and is intended to store results in. If 'results_dir' in the database is empty, a new and unique directory is created and the path stored in 'results_dir'. Otherwise the path in 'results_dir' is just returned.

Return character(len=:), allocatable Path to new subdirectory.

Parameters

- `path_directory`: **character(len=*), intent(in)** Path to where the new directory is created. If it starts with 'root/', that part will be replaced with the full path to the root directory of the project.

character(len=:) function, allocatable sim_db_mod::unique_results_dir (self, path_to_dir)
Get path to subdirectory in `path_directory` unique to simulation.

The subdirectory will be named 'date_time_name_id' and is intended to store results in. If 'results_dir' in the database is empty, a new and unique directory is created and the path stored in 'results_dir'. Otherwise the path in 'results_dir' is just returned.

Return character(len=:), allocatable Path to new subdirectory.

Parameters

- `path_directory`: **character(len=*), intent(in)** Path to where the new directory is created. If it starts with 'root/', that part will be replaced with the full path to the root directory of the project.

character(len=:) function, allocatable sim_db_mod::unique_results_dir_abs_path (self, abs_p
Get path to subdirectory in `abs_path_to_dir` unique to simulation.

The subdirectory will be named 'date_time_name_id' and is intended to store results in. If 'results_dir' in the database is empty, a new and unique directory is created and the path stored in 'results_dir'. Otherwise the path in 'results_dir' is just returned.

Return character(len=:), allocatable Path to new subdirectory.

Parameters

- `abs_path_to_dir`: **character(len=*), intent(in)** Absolute path to where the new directory is created.

logical function sim_db_mod::column_exists (self, column)
Return true if `column` is a column in the database.

Parameters

- `column`: **character(len=*), intent(in)** Name of column in database.

logical function sim_db_mod::is_empty (self, column)
Return true if entry in database under `column` is empty.

Parameters

- `column`: **character(len=*), intent(in)** Name of column in database.

subroutine `sim_db_mod::`**`set_empty`**`(self, column)`
Set entry under `column` in database to empty.

Parameters

- `column`: **character(len=*), intent(in)** Name of column in database.

integer function sim_db_mod::get_id (self)
Return ID number of simulation in the database that is connected.

character(len=:) function, allocatable sim_db_mod::get_path_proj_root (self)
Return path to root directory of the project, where *.sim_db/* is located.

generic sim_db_mod::sim_db::update_sha1_executables => update_sha1_executables_conditionall
Save the sha1 hash of the files `paths_executables` to the database.

Overload of the **update_sha1_executables_*** functions, which gives the valid types of parameters.

subroutine `sim_db_mod`**`::update_sha1_executables_conditionally`**(self, paths_executables, only_if_empty)

Parameters

- `paths_executables`: **character(len=:), dimension(:), allocatable, intent(in)** Paths to executable files.
- `len`: Length of `paths_executables`.
- `<strong>logicalintent(in)</strong>`: only_if_empty If True, it will only write to the database if the simulation's entry under 'sha1_executables' is empty. Will avoid any potential timeouts for concurrect applications.

subroutine `sim_db_mod`**`::update_sha1_executables_unconditionally`**(self, paths_executables)

Parameters

- `paths_executables`: **character(len=:), dimension(:), allocatable, intent(in)** Paths to executable files.
- `len`: Length of `paths_executables`.

subroutine `sim_db_mod`**`::allow_timeouts`**(self, is_allowing_timeouts)

Allow timeouts to occure without exiting if set to true.

A timeout occures after waiting more than 5 seconds to access the database because other threads/processes are busy writing to it. ***sim_db*** will exit with an error in that case, unless allow timeouts is set to true. It is false by default. If allowed and a timeout occures the called funciton will have had no effect.

`logical function sim_db_mod::have_timed_out (self)`

Checks if a timeout have occured since last call to this function.

subroutine `sim_db_mod`**`::delete_from_database`**(self)

Delete simulation from database.

subroutine `sim_db_mod`**`::close`**(self)

Add metadate for 'used_walltime' to database and update 'status' to 'finished' and cleans up.

CHAPTER

TEN

TIPS AND RECOMMENDATIONS

A number of tips, recommendations and explainations that might be useful is listed here:

- `sdb` is a shorter way of calling the `sim_db` command line tool and does not differ in any other way.
- If the *sim_db/* directory is empty after having cloned a projet that uses **sim_db**, go into the *sim_db/* directory and run these two commands; `$ git submodule init` and `$ git submodule update`.
- For C++ the instance of `SimDB` must be initialized at the beginning of the simulation and kept to the end. This is because its initializer and destructor take the time of the simulation as well as writing a number of thing to the database, including the simulations status. The same is true for Python, but in addition the `close()` method need to be called. For C `sim_db_ctor()` and `sim_db_dtor()` must for the same reason, be called at the beginning and end of the simulation. `sim_db_dtor()` also does clean up and must be called to avoid memory leaks.
- It is recommended to add a '*name* (string): *name of simulation run*' and a '*describtion* (string): *describtion of simulation*' to explain which simulation it is and the intent of the simulation. This makes it much easier to navigate all the simulations that accumulates at a later time.
- The `run` and `add_and_run` commands will print the output from the *run_command* to the terminal while the program runs, but it may take some time before start printing the output.
- It is recommended to use '*root/*' in the *run_command* to give the path to the *executable_program* relative to the project's root directory. This is because the '*root/*' will be replaced with the full path to the file when running the simulation, which may be necessary when running on a cluster or supercomputer.
- Any stand alone hashtages, `#`, that occure in the *run_command* will be replaced with the number passed after the `-n` flag in the `run` command. Ex. : `mpirun -n # python program.py`.
- Small results can be written to the database, but large results are recommended to be saved in a subdirectory in a result made by `unique_results_dir` inside a result directory.
- If **sim_db** is not working as expected, it might be because **sim_db** is both included and installed (usually not done on purpose) and an unexpected version of **sim_db** is therefore used. So, if **sim_db** is not working as expected run `sim_db --version` and `which sim_db` to check that you are using the expected version and that it is located in the expected directory (will indirectly tell you if it is installed or included).
- For all commands that end with *_sim*, this ending can be omitted. `add` can for instance be used instead of `add_sim`.
- The `cd_results_dir` command replaces the currect shell process with a new one, so one can return to the original directory and shell instance with the `$ exit` command.
- The command line tool can be called with `python sim_db/sim_db/__main__.py'` instead of just `'sim_db'` or `'sdb'`, if it is perferable.
- Numpy arrays can be passed to the Python write method as long as type_of_value is set.

- Multiple default names for the parameter files can be added in prioritized order in *settings.txt* to replace or in addition to *sim_params.txt*.
- If add error message occur during the simulation, consider adding the error message as a comment, `add_comment --id 'ID' --filename 'standard_error.out'`. An explaination may be appended in addition.

PYTHON MODULE INDEX

S

INDEX

U

W