simexpal

Release 1.0

Jun 02, 2021

Contents:

CHAPTER 1

Quick Start

1.1 Installation

Simexpal requires Python 3 and can be installed via pip3:

```
$ pip3 install simexpal
```

1.2 Quick Example

The simexpal repository contains a small example that you can try out to quickly get to know the tool. In this example, we compare different (inefficient) sorting algorithms on multiple inputs.

Note: While this constitutes a toy example, more complex examples can be handled using the same workflow. Indeed, simexpal has successfully been used to manage benchmarks that are published in algorithmic research papers.

1. Install simexpal as detailed above.
2. Clone the simexpal repository and navigate to the `examples/sorting/` directory:

   ```
   $ git clone https://github.com/hu-macsy/simexpal.git
   $ cd simexpal/examples/sorting/
   ```

   This directory contains an `experiments.yml` file that stores the configuration of all instances and experiments and some scripts: `generate.py` is used to generate random instances, the `sort.py` executes all algorithms and `eval.py` is a script that uses simexpal's Python interface to evaluate benchmarking results.
3. Generate some instances for the benchmark:

   ```
   # List the instances declared in experiments.yml.
   # Note that missing instances will appear in red.
   $ simex instances
   ```

```
uniform-n1000-s1
uniform-n1000-s2
uniform-n1000-s3
```

```
$ simex instances install # Generate missing instance files.
```

In the `experiments.yml` the instances are declared as part of the `instances` stanza. This stanza also declares how to invoke the generator script `generate.py`.

4. Launch the algorithms on all instances:

```
$ simex experiments # List experiment configurations from experiments.yml.
```

```
Experiment                      Instance                        Status
----------                      --------                        ------
bubble-sort                     uniform-n1000-s1                [0]
bubble-sort                     uniform-n1000-s2                [0]
bubble-sort                     uniform-n1000-s3                [0]
insertion-sort                  uniform-n1000-s1                [0]
insertion-sort                  uniform-n1000-s2                [0]
insertion-sort                  uniform-n1000-s3                [0]
```

```
$ simex experiments launch # Launch all configurations locally.
```

```
$ simex experiments # Review the status of the experiments.
```

```
Experiment                      Instance                        Status
----------                      --------                        ------
bubble-sort                     uniform-n1000-s1                [0]␣
↪finished
bubble-sort                     uniform-n1000-s2                [0]␣
↪finished
bubble-sort                     uniform-n1000-s3                [0]␣
↪finished
insertion-sort                  uniform-n1000-s1                [0]␣
↪finished
insertion-sort                  uniform-n1000-s2                [0]␣
↪finished
insertion-sort                  uniform-n1000-s3                [0]␣
↪finished
```

In the `experiments.yml` file, all experiment configurations (including the invocation of `sort.py`) are declared as part of the `experiments` stanza.

5. Evaluate the results:

```
# Here, we use the popular pandas package to aggregate the results.
# Make sure that pandas is installed on your machine (pip3 install pandas).
$ ./eval.py
```

```
experiment      comparisons          swaps      time
bubble-sort        499500.0  253437.333333  0.091776
insertion-sort     241891.0  257609.000000  0.039501
```

`eval.py` is a simple 25 line script that uses the simexpal Python interface (i.e., the functions `collect_successful_results()` and `open_output_file()`) to gather all results. It uses `pandas`

to aggregate statistics over all experiments.

1.3 Running Experiments

As a simple example, we compare Insertion Sort and Bubble Sort on a set of instances (such an example is available in the within examples). To this purpose, we created a new sorting directory, and wrote a short Python script sort.py where we implemented the two algorithms. `sort.py` accepts two arguments: the algorithm name (i.e. insertion-sort or bubble-sort) and the path to the instance. Then, you generated a bunch of instances and placed them in a directory called "instances" within "project" (in this example we just deal with a single instance random_500.list). Now we can run a sorting algorithm on a specific instance with:

```
python3 sort.py --algo=insertion-sort ./instances/random_500.list
```

To keep the example simple, we assume that instances are lists of integers.

We can now start to configure simexpal to automatize the experimental pipeline. First, we need to create a new experiments.yml file within the `sorting` directory. This is a configuration file that is read by simexpal to run the experiments on the desired instances and it is structured as below:

Listing 1: Example of experiments.yml file

```yaml
instances:
  - generator:
      args: ['./generate.py', '--seed=1', '1000']
    items:
      - uniform-n1000-s1
  - generator:
      args: ['./generate.py', '--seed=2', '1000']
    items:
      - uniform-n1000-s2
  - generator:
      args: ['./generate.py', '--seed=3', '1000']
    items:
      - uniform-n1000-s3

experiments:
  - name: insertion-sort
    args: ['./sort.py', '--algo=insertion-sort', '@INSTANCE@']
    stdout: out
  - name: bubble-sort
    args: ['./sort.py', '--algo=bubble-sort', '@INSTANCE@']
    stdout: out
```

The structure of this file will be better explained later in the guide. At this point, our `sorting` directory looks like this:

```
sorting
├── sort.py
├── experiments.yml
└── instances
    └── random_500.list
```

After having completed this steps, we can start using simexpal to run our experiments. A complete list of experiments and their status can be seen by:

```
$ simex experiments list
```

The color of each line represents the status of the experiment:

- Green: finished
- Yellow: running
- Red: failed
- Default: not executed

Experiments can be launched with:

```
$ simex experiments launch
```

This instruction will launch the non executed experiments on the local machine.

1.4 Evaluating Results

After experiments have been run, simexpal can assist with locating and collecting output data. To do this, simexpal can be imported as a Python package. As simexpal is output format and algorithm agnostic, you need to provide functionality to parse output files and evaluate results. Parsing output files can usually be greatly simplified by using standardized formats and appropriate libraries.

The example below (i.e., `eval.py` from `examples/sorting/`) demonstrates this concept. It uses the simexpal Python package to obtain all output files and meta data about them. In particular, it uses the functions `collect_successful_results()` and `open_output_file()` for this purpose. A user-supplied parsing function is employed to parse the output files.

```
def parse(run, f):
        output = yaml.load(f, Loader=yaml.Loader)
        return {
                'experiment': run.experiment.name,
                'instance': run.instance.shortname,
                'comparisons': output['comparisons'],
                'swaps': output['swaps'],
                'time': output['time']
        }

cfg = simexpal.config_for_dir()
results = []
for successful_run in cfg.collect_successful_results():
        with successful_run.open_output_file() as f:
                results.append(parse(successful_run, f))

df = pandas.DataFrame(results)
print(df.groupby('experiment').agg('mean'))
```

1.5 Managing Instances

Before launching the experiments, make sure that all your instances are available. Instances can be checked with:

```
$ simex instances list
```

Unavailable instances will be shown in red, otherwise they will be shown in green. If instances are taken from a public repository, they can be downloaded automatically. We configured the YAML file below to use instances from SNAP.

Listing 2: experiments.yml with instances from public repositories.

```yaml
instances:
  - repo: snap
    items:
      - 'facebook_combined'
      - 'cit-HepTh'

instdir: "./graphs"
```

All the listed instances can be downloaded within the `./graphs` directory with:

```
$ simex instances install
```

1.6 Dealing with Parameters and Variants of an Algorithm

When benchmarking algorithms, it is often useful to compare different variants or parameter configurations of the same algorithm. simexpal can manage those variants without requiring you to duplicate the `experiments` stanza multiple times.

As an example, imagine that you want to benchmark the running time of merge sort using different minimum block sizes, as well as its running time depending on different algorithms for minimal blocks.

Those variants can be handled by simexpal using the following stanzas:

```yaml
experiments:
  - name: 'merge-sort'
    stdout: out
    args: ['python3', 'sort.py', '--algo=insertion-sort', '@EXTRA_ARGS@', '@INSTANCE@
↪']

variants:
  - axis: 'block-size'
    items:
      - name: 'bbs1'
        extra_args: ['--base-block-size=1']
      - name: 'bbs10'
        extra_args: ['--base-block-size=10']
      - name: 'bbs50'
        extra_args: ['--base-block-size=50']
  - axis: 'block-algo'
    items:
      - name: 'bba-insertion'
        extra_args: ['--base-block-algorithm=insertion-sort']
      - name: 'bba-selection'
        extra_args: ['--base-block-algorithm=selection-sort']
```

simexpal will duplicate the experiment for each possible combination of variants. Such a combination will consist of exactly one variant for every `axis` property.

1.7 Automated Builds and Revision Support

To make sure that experiments are always run from exactly the same binaries, it is possible to let simexpal pull your programs from some VCS (as of version 0.1, only Git is supported) and build them automatically.

Automated builds are controlled by the `builds` and `revisions` stanzas in `experiments.yml`.

In the remainder of this section, we will reconsider the sorting example from the *Quick Example* section. Instead of using a Python implementation of the algorithms, we assume a C++ implementation and use simexpal's automated build support to compile it. In this example, simexpal will invoke a CMake build system to build the program; however, simexpal is independent of the particular build system in use.

To enable automated builds, we need to add `builds` and `revisions` stanzas to `experiments.yml`. In our example, these look like:

```
builds:
  - name: simexpal
    git: 'https://github.com/hu-macsy/simexpal'
    configure:
      - args:
        - 'cmake'
        - '-DCMAKE_INSTALL_PREFIX=@THIS_PREFIX_DIR@'
        - '@THIS_CLONE_DIR@/examples/sorting_cpp/'
    compile:
      - args:
        - 'make'
        - '-j@PARALLELISM@'
    install:
      - args:
        - 'make'
        - 'install'

revisions:
  - name: main
    build_version:
      'simexpal': 'd8d421e3c2eaa32311a6c678b15e9e22ea0d8eac'      # specify the SHA-1␣
↪hash of a tagged commit (recommended)
                                                                  # it is also␣
↪possible to checkout the top commit
                                                                  # of a branch by␣
↪specifying the SHA-1 hash or
                                                                  # branch name (not␣
↪recommended for reproducibility reasons)
```

Next, we have to assign the builds to their respective experiments:

```
experiments:
  - name: quick-sort
    use_builds: [simexpal]  # specify which builds get used for this experiment
    args: ['quicksort', '@INSTANCE@', '@EXTRA_ARGS@']
    stdout: out
```

Simexpal resolves the `@INSTANCE@` variable to the instance paths and the `@EXTRA_ARGS@` to the extra arguments of the variants (that we define below) during runtime.

After `experiments.yml` has been adopted, we can run the automated build using

```
$ simex builds make
```

Once the build process is finished, the experiments can be started as usual.

1.8 Run Matrix

In the *Dealing with Parameters and Variants of an Algorithm* section we saw how we can use simexpal to specify variants of experiments. For the following example we will consider this `variants` stanza:

```
variants:
  - axis: 'block-algo'
    items:
      - name: 'ba-insert'
        extra_args: ['insertion_sort']
      - name: 'ba-bubble'
        extra_args: ['bubble_sort']
  - axis: 'block-size'
    items:
      - name: 'bs32'
        extra_args: ['32']
      - name: 'bs64'
        extra_args: ['64']
```

simexpal will build every possible combination of experiment, instance, variant and revision. There are cases where this is not desired. For example, you might only want to run certain instance/variant combinations.

Assume you want to run Quicksort with Insertionsort as base block algorithm and `32` as minimal block size. Additionally you want to run Quicksort with Bubblesort as base block algorithm and use both `32` and `64` as minimal block sizes.

To achieve this, we need to add a `matrix` stanza to `experiments.yml`. In our example, this looks like:

```
matrix:
  include:
    - experiments: [quick-sort]
      variants: [ba-insert, bs32]
      revisions: [main]
    - experiments: [quick-sort]
      variants: [ba-bubble]      # We could explicitly specify [ba-bubble, bs32, bs64].
↪ In this case it is not
                                 # necessary as bs32 and bs64 are all the possible␣
↪values for the block-size axis
      revisions: [main]
```

(The full `experiments.yml` can be found here .)

Using `simex experiments list` we can confirm that we got our desired experiments:

```
Experiment                                  Instance                  Status
----------                                  --------                  ------
quick-sort ~ ba-bubble, bs32 @ main         uniform-n1000-s1          [0] not␣
↪submitted
quick-sort ~ ba-bubble, bs32 @ main         uniform-n1000-s2          [0] not␣
↪submitted
quick-sort ~ ba-bubble, bs64 @ main         uniform-n1000-s1          [0] not␣
↪submitted
quick-sort ~ ba-bubble, bs64 @ main         uniform-n1000-s2          [0] not␣
↪submitted
```

(continues on next page)

(continued from previous page)

```
quick-sort ~ ba-insert, bs32 @ main          uniform-n1000-s1            [0] not␣
↪submitted
quick-sort ~ ba-insert, bs32 @ main          uniform-n1000-s2            [0] not␣
↪submitted
```

1.9 Launchers / Support for Batch Schedulers

To submit experiments to a batch scheduler, simexpal allows you to define "launchers". A launcher specifies where and how simexpal should submit experiments. If no launcher (not even a default launcher or `--launch-through`) is specified, simexpal launches experiments on the local machine.

Launchers are defined in a file `~/.simexpal/launchers.yml`. For example, to submit jobs to the Slurm partition `fat-nodes`, a launcher configuration could look like this:

```
launchers:
  - name: local-cluster
    default: true
    scheduler: slurm
    queue: fat-nodes
```

When launching experiments using `simex experiments launch`, you can specify the `--launcher` option (e.g., `simex experiments launch --launcher local-cluster`) to select a certain launcher. Note that the `default: true` attribute of a launcher overrides the default behavior of launching on the local machine (hence, there can only be one launcher with `default: true`).

CHAPTER 2

The “experiments.yml” reference

Simexpal needs an `experiments.yml` file to to automatically execute your experiments on the desired instances. In this page we describe the structure of the `experiments.yml` file. The `experiments.yml` is a YAML file that contains a dictionary with several keys:

- `instances`: list of all the instances that will be used for the experiments
- `instdir`: path to the directory that stores all the instances
- `experiments`: list of all the experiments that will be executed on the instances

There are also further keys that allow for customization of the experiments and to automatically build the binaries the experiments run from:

- `builds`: list of Git builds, which also include build instructions
- `revisions`: list of Git revisions of builds
- `variants`: list of additional input parameters for experiments
- `matrix`: specifies which combinations of experiments, instances, variants and revisions are run

2.1 Instances

This entry is a list of instances that will be used for experiments. The following keys are used for specifying instances:

- `extensions`: list of extensions that the instance has
- `files`: list of files the instance consists of
- `items`: list of instances
- `name`: name of the instance (used when dealing with instances that consist of unrelated files)
- `repo`: source of instances
- `set`: list of sets the instance belongs to

For detailed usage examples, see the *Instances* page.

2.2 Builds

This entry is a list of builds that will be used for revisions. The following keys are used for specifying builds:

- `configure`: list of dictionaries containing configuration parameters
- `compile`: list of dictionaries containing compilation parameters
- `environ`: dictionary of (environment variable, value)-pairs
- `git`: link to the Git repository
- `install`: list of dictionaries containing installation parameters
- `name`: (arbitrary) name of the build
- `recursive-clone`: boolean (`true`/`false`) - whether to pull git submodules recursively or not
- `regenerate`: list of dictionaries containing regeneration parameters
- `requires`: list of required builds
- `workdir`: path of the working directory

For detailed usage examples, see the *Builds* page.

2.3 Revisions

This entry is a list of revisions that will be used for experiments. The following keys are used for specifying revisions:

- `build_version`: dictionary of (build, SHA-1 hash/branch)-pairs
- `develop`: boolean (`true`/`false`) - whether this revision is a develop revision or not
- `name`: (arbitrary) name of the revision

For detailed usage examples, see the *Revisions* page.

2.4 Experiments

This entry is a list of experiments that will be executed on all the instances. Each experiment includes three keys:

- `args`: list of experiment arguments
- `environ`: dictionary of (environment variable, value)-pairs
- `name`: name of the experiment
- `num_nodes`: number of nodes on which to run
- `num_threads`: number of cpus required per task
- `output`: dictionary containing all output file extensions
- `procs_per_node`: number of tasks to invoke on each node
- `repeat`: integer - number of times an experiment is repeated
- `slurm_args`: list of additional `sbatch` arguments
- `stdout`: extension of the output file
- `timeout`: integer - timeout in seconds

- `use_builds`: list of used build names

For detailed usage examples, see the *Experiments* page.

2.5 Variants

This entry is a list of variants that will be used for experiments. The following keys are used for specifying variants:

- `axis`: name of the variant axis
- `environ`: dictionary of (environment variable, value)-pairs
- `extra_args`: list of variant arguments
- `items`: list of dictionaries, which specify variants belonging to the same axis.
- `name`: name of the variant
- `num_nodes`: number of nodes on which to run
- `num_threads`: number of cpus required per task
- `procs_per_node`: number of tasks to invoke on each node

For detailed usage examples, see the *Variants* page.

2.6 Run Matrix

This entry is a list of desired experiment combinations. The following keys are used for specifying desired experiment combinations:

- `axes`: list of included axis names
- `experiments`: list of included experiment names
- `include`: list of dictionaries, which specify included experiment combinations
- `instsets`: list of included instance set names
- `repetitions`: integer - number of times all combinations of an `include` entry are repeated
- `revisions`: list of included revision names
- `variants`: list of included variant names

For detailed usage examples, see the *Run Matrix* page.

CHAPTER 3

@-Variables

@-variables are placeholder variables that can be used in the `experiments.yml` file. They are enclosed by at signs, i.e. `@...@` and get resolved during runtime.

For example: In order to avoid the duplication of command line arguments of experiments for each instance, e.g. `experiment.py /path/to/instance1`, `experiment.py /path/to/instance2`, ... we use the `@INSTANCE@` variable that resolves to the respective paths of the instances. Then, we only need to specify `experiments.py @INSTANCE@` as experiment arguments.

Below, we list all @-variables and where they can be used.

- `@BASE_DIR@`: path of the directory containing the `experiments.yml`
- `@COMPILE_DIR_FOR:<build_name>@`: *compilation directory* of `<build_name>` in the same revision
- `@EXTRA_ARGS@`: extra arguments of all variants and the instance of an experiment
- `@INSTANCE@`: path of a *local*/*remote* instance, i.e. `/instance_directory/<instance_name>`
- `@INSTANCE_DIR@`: path of the *Instance Directory*
- `@INSTANCE:<ext>@`: path of a *Multiple Extensions* instance with extension `<ext>`, i.e. `/instance_directory/<instance_name>.<ext>`
- `@INSTANCE:<idx>@`: path of an *Arbitrary Input Files* instance with index `<idx>` in the `files` key, i.e. `/instance_directory/files[<idx>]`
- `@INSTANCE_FILENAME@`: filename of the instance
- `@OUTPUT@`: path to the output file of an experiment
- `@OUTPUT:<ext>@`: path to the output file with extension `<ext>` of an experiment
- `@OUTPUT_SUBDIR@`: output subdirectory of the experiment where the output and status files are stored, i.e. `/path_to_experiments_yml/output/`
- `@PARALLELISM@`: number of available CPUs
- `@PREFIX_DIR_FOR:<build_name>@`: *installation directory* of `<build_name>` in the same revision
- `@REPETITION@`: repetition number of this experiment

- `@SOURCE_DIR_FOR:<build_name>@`: *source directory* of `<build_name>` in the same revision
- `@THIS_CLONE_DIR@`: Deprecated for `@THIS_SOURCE_DIR@`
- `@THIS_COMPILE_DIR@`: *compilation directory* of this build
- `@THIS_PREFIX_DIR@`: *installation directory* of this build
- `@THIS_SOURCE_DIR@`: *source directory* of this build

3.1 Builds

3.1.1 compile

The following @-variables can be used in the `compile` key:

- `@BASE_DIR@`
- `@COMPILE_DIR_FOR:<build_name>@`
- `@INSTANCE_DIR@`
- `@PARALLELISM@`
- `@PREFIX_DIR_FOR:<build_name>@`
- `@SOURCE_DIR_FOR:<build_name>@`
- `@THIS_CLONE_DIR@` (deprecated for `@THIS_SOURCE_DIR@`)
- `@THIS_COMPILE_DIR@`
- `@THIS_PREFIX_DIR@`
- `@THIS_SOURCE_DIR@`

3.1.2 configure

Same as for the *compile* key.

3.1.3 environ

The values of the `environ` key will be substituted and the @-variables are the same as for the *compile* key.

3.1.4 extra_paths

Same as for the *compile* key *without* the `@PARALLELISM@` variable.

3.1.5 install

Same as for the *compile* key.

3.1.6 regenerate

Same as for the *compile* key.

3.1.7 workdir

Same as for the *compile* key.

3.2 Experiments

3.2.1 args

The following @-variables can be used in the `args` key:

- `@BASE_DIR@`
- `@COMPILE_DIR_FOR:<build_name>@` (`<build>` has to be in `used_builds` or be required by a build in it)
- `@EXTRA_ARGS@`
- `@INSTANCE@`
- `@INSTANCE_DIR@`
- `@INSTANCE:<ext>@`
- `@INSTANCE:<idx>@`
- `@OUTPUT@`
- `@OUTPUT:<ext>@`
- `@OUTPUT_SUBDIR@`
- `@PREFIX_DIR_FOR:<build_name>@` (`<build_name>` has to be in `used_builds` or be required by a build in it)
- `@REPETITION@`
- `@SOURCE_DIR_FOR:<build_name>@` (`<build_name>` has to be in `used_builds` or be required by a build in it)

3.2.2 workdir

Same as for the *args* key *without* the `@EXTRA_ARGS@` variable.

3.3 Instances

3.3.1 url

The following @-variables can be used in the `url` key:

- `@INSTANCE_FILENAME@`

CHAPTER 4

Instances

You might want to take a look at the following pages before exploring instances:

- *Quick Start*
- *@-Variables*

On this page we describe how to specify instances in the `experiments.yml` file. You can list local instances that consist of zero or more files. More over simexpal can download remote instances from the SNAP repository, Git repositories and arbitrary URLs. It is also possible to assign instances to instance sets that enable a more efficient usage of the *command line interface* and are useful when defining the run matrix.

4.1 Instance Directory

The instance directory is the directory that stores all the instances. The path can be set via the `instdir` key:

Listing 1: How to set the instance directory in the experiments.yml file.

```
instdir: "<path_to_instance_directory>"
```

If `instdir` is not set, it will default to `<path_to_experiments.yml_directory>/instances`. The instance directory will be created if it does not exist already.

4.2 Local Instances

To add local instances to the `instances` key, we add a list of dictionaries with two keys to its value:

- `repo`: source of the instances
- `items`: a list of instances.

An example of how to list a local set of instances is:

Listing 2: How to list local instances in the experiments.yml file.

```yaml
instances:
  - repo: local
    items:
      - random_500.list
      - partially_sorted_500.list

instdir: "<path_to_instances_directory>"
```

4.3 Remote Instances

It is possible to let simexpal download instances from SNAP, a URL and a Git repository. In the sections below, we will see how to list the different kinds of remote instances in the `experiments.yml`.

After listing the instances we need to use

```
$ simex instances install
```

to download the instances into the instance directory.

Note: 1st December 2020: It is no longer possible to automatically download KONECT instances as the website is no longer publicly available. It is still possible to list them and execute supported actions, e.g, transforming the instances to edgelist format via `simex instances run-transform --transform='to_edgelist'` if you already have them saved locally.

4.3.1 Instances From SNAP

To list instances from the SNAP repository, set the value of `repo` to `snap` and put the file names without the `.txt.gz` extension in the `items` list.

For instances from the KONECT repository, set the value of `repo` to `konect` and put the internal names of the KONECT instances in the `items` list.

Listing 3: How to list instances from the SNAP and KONECT repository in the experiments.yml file.

```yaml
instdir: "<path_to_instance_directory>"
instances:
  - repo: snap
    items:
      - facebook_combined
      - wiki-Vote
  - repo: konect
    items:
      - dolphins
      - ucidata-zachary
```

4.3.2 Instances From a URL

To list instances from a URL, we use the following keys:

- `method`: download method
- `url`: URL of the instance

We set the value of the `method` key to `'url'` and specify the URL of the instance in the `url` key.

Listing 4: How to list instances from a URL in the experiments.yml file.

```yaml
instdir: "<path_to_instance_directory>"
instances:
  - method: url
    url: 'https://raw.githubusercontent.com/hu-macsy/simexpal/master/simexpal/schemes/@INSTANCE_FILENAME@'
    items:
      - 'experiments.json'
      - 'launchers.json'
```

The *@-variable* `@INSTANCE_FILENAME@` in the URL (from the example above) resolves to the elements in the `items` key. Thus, we have listed the two instances `experiments.json` and `launchers.json`, which come from https://raw.githubusercontent.com/hu-macsy/simexpal/master/simexpal/schemes/experiments.json and https://raw.githubusercontent.com/hu-macsy/simexpal/master/simexpal/schemes/launchers.json respectively.

4.3.3 Instances From Git

To list instances from a Git repository, we use the following keys:

- `method`: download method
- `git`: link to the Git repository
- `repo_name`: name of the directory to clone into
- `commit`: SHA-1 hash
- `git_subdir`: subdirectory of the instance in the Git repository

We set the value of the `method` key to `'git'` and specify the Git URL of the instance in the `git` key. The `repo_name` states the local directory name of the Git repository. When installing the instance, the Git repository will be stored in `<instance_dir>/<repo_name>`. The `commit` value specifies the version of the instance given as SHA-1 hash. It is also possible to specify other revision parameters, e.g. ref names. For reproducibility reasons the former variant is recommended. If the instance is not located in the root directory of the Git repository, we will need to specify the subdirectory of the instance in `git_subdir`.

Listing 5: How to list instances from a Git repository in the experiments.yml file.

```yaml
instdir: "<path_to_instance_directory>"
instances:
  - method: git
    git: 'https://github.com/hu-macsy/simexpal'
    repo_name: 'foo'
    commit: 'master'
    items:
      - 'setup.py'
      - 'pytest.ini'
  - method: git
    git: 'https://github.com/hu-macsy/simexpal'
    repo_name: 'foo'
    commit: 'd5e598f292b90cd7ef2e77d7a478ec52d42279df'
```

(continues on next page)

(continued from previous page)

```
    git_subdir: 'simexpal/schemes/'
    items:
      - 'experiments.json'
      - 'launchers.json'
```

In the example above we clone the simexpal repository into `<instance_dir>/foo`. Then `setup.py` and `pytest.ini` of the current `master` branch and `simexpal/schemes/experiments.json` and `simexpal/schemes/launchers.json` of the specified commit `d5e598f292b90cd7ef2e77d7a478ec52d42279df` will be downloaded into the instance directory.

4.4 Multiple Input Files

Until now we only considered experiments with one input file, which might not always be the case. Below we distinguish two cases:

1. The input filenames only differ in the extension, e.g. `foo.graph` and `foo.xyz`.
2. The input filenames are arbitrary.

4.4.1 Multiple Extensions

Listing instances with multiple extensions is similar to listing *Local Instances*. The difference is that we will add the following key:

- `extensions`: list of extensions that the instance has

Listing 6: How to list instances with multiple extensions in the experiments.yml file.

```
instdir: "<path_to_instance_directory>"
instances:
  - repo: local
    extensions:
      - graph
      - xyz
    items:
      - foo
      - bar
```

The `experiments.yml` file above will create the instance `foo` which contains the files `foo.graph` and `foo.xyz` and the instance `bar` which contains the files `bar.graph` and `bar.xyz`.

4.4.2 Arbitrary Input Files

To get an instance with arbitrary input files we will put a list of dictionaries as value for the `items` key. The dictionaries contain two keys:

- `name`: name of the instance
- `files`: list of files the instance consists of

Listing 7: How to list instances with arbitrary input files in the experiments.yml file.

```yaml
instdir: "<path_to_instance_directory>"
instances:
  - repo: local
    items:
      - name: foo
        files:
          - file1
          - file2
      - name: bar
        files:
          - file3
          - file4
```

The `experiments.yml` file above will create the instance `foo` which contains the files `file1` and `file2` and the instance `bar` which contains the files `file3` and `file4`.

4.5 Fileless Instances

There are cases where instances are not defined by a file but rather by some input parameters, e.g. algorithms that generate their data themselves and only need input parameters like `--seed 10`. Specifying fileless instances works similar to specifying *Arbitrary Input Files*. The difference is, that we set `files: []` to indicate that we are dealing with a fileless instance and use the

- `extra_args`: list of extra arguments

key to specify our extra arguments.

Listing 8: How to list fileless instances in the experiments.yml file.

```yaml
instances:
  - repo: local
    items:
      - name: foo
        files: []
        extra_args: ['--seed', '10']
```

4.6 Extra Arguments

We can set extra arguments for instance blocks and individual instances, which can be appended to the experiment arguments when the respective instance is used. In order to specify such instances, we use the

- `extra_args`: list of extra arguments

key.

Note: It is possible to have instances with common extra arguments and additional individual extra arguments by adding the `extra_args` key to the respective places (see below).

In order to specify common extra arguments for instance blocks, we simply add the `extra_args` key to them.

Listing 9: How to list instances with common extra arguments in the experiments.yml file.

```yaml
instdir: "<path_to_instance_directory>"
instances:
  - repo: local
    extra_args: ['some','extra_args']
    ...
    items:
      - instance1
      - instance2
  - repo: local
    extra_args: ['some','extra_args']
    ...
    items:
      - instance3
      - instance4
```

In order to specify individual extra arguments for *Local Instances*, *Remote Instances* and *Multiple Extensions* we need to change the `items` key from a list of instances to a list of dictionaries containing the

- `name`: name of the instance *and*
- `extra_args`

key, e.g.,

Listing 10: How to list local/remote/multiple extension instances with individual extra arguments in the experiments.yml file.

```yaml
instances:
  - repo: local # local instances with extra arguments
    items:
      - name: inst1
        extra_args: ['some','extra_args']
      - name: inst2
        extra_args: ['some','extra_args']
  - repo: snap # remote instances with extra arguments
    items:
      - name: facebook_combined
        extra_args: ['some', 'extra_args']
      - name: wiki-Vote
        extra_args: ['some', 'extra_args']
  - repo: local # multiple extension instances with extra args
    items:
      - name: inst3
        extra_args: ['some', 'extra_args']
      - name: inst4
        extra_args: ['some', 'extra_args']
    extensions: [ext1, ext2]
```

For *Arbitrary Input Files* we only need to add the `extra_args` key to the dictionaries of the instances, e.g,

Listing 11: How to list arbitrary input file instances with individual extra arguments in the experiments.yml file.

```yaml
instances:
  - repo: local
```

(continues on next page)

(continued from previous page)

```
    items:
      - name: inst3
        files: [file1, file2]
        extra_args: ['some', 'extra', 'argument']
      - name: inst4
        files: [file1, file2]
        extra_args: ['some', 'extra', 'argument']
```

4.7 Instance Sets

It is possible to assign instances to instance sets. This is useful when trying to run experiments that have common instances. Assume you want to run an experiment on the two instances `instance1` and `instance2` and a different experiment on `instance2` and `instance3`. To do this, you can use the following key:

- `set`: list of sets the instance belongs to

Listing 12: How to assign instances to instance sets in the experiments.yml file.

```
instdir: "<path_to_instance_directory>"
instances:
  - repo: local
    set: [set1]
    items:
      - instance1
  - repo: local
    set: [set1, set2]
    items:
      - instance2
  - repo: local
    set: [set2]
    items:
      - instance3
```

In this way we have created the instance set `set1`, which contains `instance1` and `instance2` and `set2`, which contains `instance2` and `instance3`.

Instance sets will also be useful when using the *command line interface* of simexpal and when defining the *Run Matrix*.

4.8 Next

To set up your automated builds, visit the *Builds* page. If you do not plan on using automated builds, you can visit the *Experiments* page to set up your experiments.

CHAPTER 5

Builds

You might want to take a look at the following pages before exploring automated builds:

- *Quick Start*
- *@-Variables*

To ensure reproducibility of experiments, it is possible to let simexpal pull programs from a VCS (currently only Git is supported) and build them automatically. To achieve this, we need to specify a `builds` and `revisions` stanza in the `experiments.yml`.

On this page we will explain the `builds` key and mainly use the C++ example from the *Quick Start* guide in order to do so. For example, we will explain how to specify a Git repository or a local build that does not come from a VCS, enable the automated build support and the directories involved in the build process. Furthermore we will present additional options such as pulling Git submodules and setting additional environment variables for the build process.

You can find information regarding the `revisions` key on the *Revisions* page.

5.1 Specifying a Git Repository

To state our git repository, we use the keys

- `name`: (arbitrary) name of the build
- `git`: link to the Git repository

as follows:

Listing 1: How to specify a Git repository in the experiments.yml file.

```
builds:
  - name: simexpal
    git: https://github.com/hu-macsy/simexpal
```

5.2 Pulling Git Submodules

If your program contains Git submodules you need to set the

- `recursive-clone`: boolean (`true`/`false`) - whether to pull git submodules recursively or not

key to `true` in order for simexpal to pull the respective files.

Listing 2: How to let simexpal pull Git submodules.

```
builds:
  - name: networkit
    git: https://github.com/networkit/networkit
    recursive-clone: true
```

If `recursive-clone` is not set, it will default to `false`.

5.3 Specifying Builds Without Version Control

Specifying builds without version control works similar to *Specifying a Git Repository*. The differences are

- we omit the `git` key *and*
- store the local build files in `./develop/<build_name>@<revision_name>`.

As of now local builds are only supported for *Develop Revisions* as we can not guarantee reproducibility without some kind of identifier, e.g, a commit hash.

All the options below also apply for local builds (we just need to omit the `git` key, where applicable).

5.4 Automated Builds

To enable automated build support, we can specify the following keys:

- `regenerate`: list of dictionaries containing regeneration parameters
- `configure`: list of dictionaries containing configuration parameters
- `compile`: list of dictionaries containing compilation parameters
- `install`: list of dictionaries containing installation parameters

Listing 3: How to specify configuration, compilation and installation parameters in the experiments.yml file.

```
builds:
  - name: simexpal
    git: 'https://github.com/hu-macsy/simexpal'
    configure:
      - args:
          - 'cmake'
          - '-DCMAKE_INSTALL_PREFIX=@THIS_PREFIX_DIR@'
          - '@THIS_CLONE_DIR@/examples/sorting_cpp/'
    compile:
      - args:
          - 'make'
```

(continues on next page)

(continued from previous page)

```
          - '-j@PARALLELISM@'
    install:
      - args:
          - 'make'
          - 'install'
```

The order of the build steps is as follows:

1. the specified *revision* of the build (and possibly its submodules) will be pulled
2. regeneration
3. configuration
4. compilation
5. installation

The purpose of the *regeneration* step is to prepare the source directory before the build starts, e.g., by downloading additional dependencies or subprojects.

During the *configuration* step we can configure our project, e.g, by running `cmake` or using a `./configure` script.

Analogously for the *compilation* and *installation* step we can compile and install our project during those steps, e.g, by running `make` and `make install`.

To specify the build parameters, we will use the `args` key and set the value to a list of arguments. Arguments are stated separately, e.g., `make install` becomes a list containing `make` and `install`. In the example above, we used CMake as build system; however, simexpal is independent of the particular build system in use.

5.4.1 Setting Environment Variables

It is possible to set environment variables for each build step. To achieve this, we can use the

- `environ`: dictionary of (environment variable, value)-pairs

key as follows:

Listing 4: How to specify environment variables for the configuration step in the experiments.yml file.

```
builds:
  - name: simexpal
    git: 'https://github.com/hu-macsy/simexpal'
    configure:
      - args:
          - 'cmake'
          - '-DCMAKE_INSTALL_PREFIX=@THIS_PREFIX_DIR@'
          - '@THIS_CLONE_DIR@/examples/sorting_cpp/'
        environ:
          'CXX': '/path/to/g++'
          'CC': '/path/to/gcc'
    compile:
      ...
    install:
      ...
```

Specifying environment variables for other steps works analogously to specifying environment variables for the configuration step (as seen above). If an environment variable already exists, then the given path will be preprended to it.

5.4.2 Setting the Working Directory

The default working directories (see *Build Directories*) for each build step are the same for *normal revisions* and *develop revisions* and are as follows:

Step	Default Working Directory
regeneration	clone directory
configuration	compilation directory
compilation	compilation directory
installation	installation directory

We can change the working directories by adding the

- `workdir`: path of the working directory

key to the respective dictionaries of the build steps.

Listing 5: How to specify the working directory for the configuration step in the experiments.yml file.

```
builds:
  - name: simexpal
    git: 'https://github.com/hu-macsy/simexpal'
    configure:
      - args:
          - 'cmake'
          - '-DCMAKE_INSTALL_PREFIX=@THIS_PREFIX_DIR@'
          - '@THIS_CLONE_DIR@/examples/sorting_cpp/'
        workdir: '/arbitrary/directory/path'
    compile:
      ...
    install:
      ...
```

Specifying the working directory for other steps works analogously to specifying the working directory for the configuration step (as seen above).

5.5 Extra Paths

For many UNIX packages it is standard to install the executable in the `@THIS_PREFIX_DIR@/bin` directory. This is why simexpal only checks those directories by default when looking for an executable. However, this assumption might not always be correct, for example, when using a custom build system. To cover those cases, we specify the

- `extra_paths`: list of extra paths, which simexpal should check when running an experiment that uses this build

key.

Listing 6: How to specify extra paths of builds in the experiments.yml file.

```yaml
builds:
  - name: build1
    ...
    extra_paths: ['/path/to/executable']
```

When running an experiment that uses this build, simexpal will prepend the paths given in `extra_paths` to the `PATH` environment variable.

5.6 Dependent Builds

There are cases where a build is dependent on other builds e.g. it needs the path to certain builds which are built before. For this case we use the

- `requires`: list of required builds

key, which contains a list of builds that need to be built before the current build. In this way we make sure that simexpal builds the required builds beforehand.

Listing 7: How to specify dependent builds in the experiments.yml file.

```yaml
builds:
  - name: build1
    ...
    requires:
      - build2
      - build3
    ...
  - name: build2
    ...
  - name: build3
    ...
```

5.7 Build Directories

Depending on the kind of the *revision* used for the builds, simexpal uses different directories. In the following subsections we will cover the directories for *normal revisions* and *develop revisions*.

5.7.1 Build Directories for Normal Builds

A *normal revision* in simexpal uses the `/builds` directory, which contains the four subdirectories

- repository directory,
- clone directory,
- compilation directory and
- installation/prefix directory,

during the build process.

The *repository directory* contains some internal information related to the builds e.g. internal tags that are used by simexpal to handle multiple *revisions* of programs. This directory should normally not be of interest for a user.

The *clone directory* contains the actual program files from a checked out branch.

The *compilation directory* contains the compilation and internal simexpal files.

The *install/prefix directory* contains the installation (usually) and internal simexpal files.

Below you can find the shortened directory structure of our C++ example example. The repository directory has `<build_name>` as prefix and `.repo` as suffix. The clone, compilation and installation directory have `<build_name>@<revision_name>` as prefix and the first two have `.clone` and `.compile` as suffix respectively. The installation directory does not have any suffix. The internal simexpal files have the suffix `.simexpal`.

Listing 8: Build directories for normal builds used by simexpal during the build process.

```
/path/to/experiments.yml/directory
├── CMakeLists.txt
├── builds
│   ├── simexpal.repo                       # repository directory
│   │   ├── internal simexpal
│   │   ├── ...
│   │   └── files/directories
│   ├── simexpal@main                       # installation/prefix directory
│   │   ├── bin
│   │   │   └── quicksort                   # our executable
│   │   └── installed.simexpal              # internal simexpal file
│   ├── simexpal@main.clone                 # clone directory
│   │   ├── checkedout.simexpal             # internal simexpal file
│   │   ├── regenerated.simexpal            # internal simexpal file
│   │   ├── project
│   │   ├── ...
│   │   └── files/directories
│   └── simexpal@main.compile               # compilation directory
│       ├── configuration and compilation
│       ├── ...
│       ├── files/directories
│       ├── compiled.simexpal               # internal simexpal file
│       └── configured.simexpal             # internal simexpal file
├── experiments.yml
└── quicksort.cpp
```

5.7.2 Build Directories for Develop Builds

A *develop revision* in simexpal uses the `/dev-builds` directory, which contains the two subdirectories

- compilation directory and
- installation/prefix directory

and the `/develop` directory, which contains the

- clone directory,

during the build process:

The functions of the respective directories are as *before* (for local builds the clone directory contains the local program files).

Below you can find the shortened directory structure of our C++ example example (if `recursive-clone` was set to `True`). The clone, compilation and installation directory have `<build_name>@<revision_name>` as prefix. Additionally, the compilation directory has `.compile` as suffix. The clone directory is located in the `/develop` directory, whereas the compilation and installation directories are located in the `/dev-builds` directory. The internal simexpal files have the suffix `.simexpal`.

Listing 9: Build directories for dev-builds used by simexpal during the build process.

```
/path/to/experiments.yml/directory
├── CMakeLists.txt
├── dev-builds
│   ├── simexpal@main                       # installation/prefix directory
│   │   ├── bin
│   │   │   └── quicksort                   # our executable
│   │   └── installed.simexpal              # internal simexpal file
│   └── simexpal@main.compile               # compilation directory
│       ├── configuration and compilation
│       ├── ...
│       ├── files/directories
│       ├── compiled.simexpal               # internal simexpal file
│       └── configured.simexpal             # internal simexpal file
├── develop
│   └── simexpal@main                       # clone directory
│       ├── project
│       ├── ...
│       ├── files/directories
│       ├── checkedout.simexpal             # internal simexpal file
│       └── regenerated.simexpal            # internal simexpal file
├── experiments.yml
└── quicksort.cpp
```

5.8 Next

To get a more detailed understanding of revisions and fully set up your builds, visit the *Revisions* page.

CHAPTER 6

Revisions

You might want to take a look at the following pages before exploring revisions:

- *Quick Start*
- *@-Variables*
- *Builds*

Simexpal supports two kinds of revisions: “normal” (static) revisions and develop (dynamic) revisions. If you have a finished project and only want to run experiments, normal revisions will suffice. If your project is not finished yet and needs refinement depending on factors like runtime or experiment outputs, you can use develop revisions.

Revisions contain builds and their desired versions. The versions can be specified in the following ways:

- SHA-1 hash of a tagged commit (recommended)
- SHA-1 hash of a top commit
- branch name (resolves to top commit of the branch)

For reproducibility reasons we recommend specifying the SHA-1 hash of a tagged commit.

6.1 “Normal” Revisions

To specify a normal revisions, we use the `revisions` key and set the value to a list of dictionaries containing the keys

- `name`: name of the revision
- `build_version`: dictionary of (build, SHA-1 hash/branch)-pairs

Listing 1: How to specify normal revisions in the experiments.yml.

```
1 revisions:
2   - name: main
3     build_version:
```

(continues on next page)

(continued from previous page)

```
      'simexpal': 'd8d421e3c2eaa32311a6c678b15e9e22ea0d8eac'    # SHA-1 hash of a␣
↪tagged commit
  - name: secondary
    build_version:
      'simexpal': 'master'
```

In the example above we created the revisions `main` and `secondary`, which both contain the build `simexpal`. The `main` revision contains a tagged version of simexpal, whereas the `secondary` revision contains the latest commit of the `master` branch. If our revision had more than one build, we would simply add new lines of `'<build_name>': '<SHA-1 hash>'` below the `'simexpal': '...'` line.

6.2 Develop Revisions

Specifying develop revisions works similarly to specifying *normal revisions*. The differences are:

- we add another key `develop` and set its value to `true` *and*
- we leave the values of the `build_version` dict as empty string `''`.

Values specified in the `build_version` dict will be ignored for develop revisions and simexpal will clone the latest project files.

Listing 2: How to specify a develop revision in the experiments.yml.

```
revisions:
  - name: main
    develop: true
    build_version:
      'simexpal': ''
```

Note: It is possible to have normal and develop revisions at the same time.

6.3 Next

Now that you have set up your automated builds, you can visit the *Experiments* page to define your experiments.

CHAPTER 7

Experiments

You might want to take a look at the following pages before exploring experiments:

- *Quick Start*
- *@-Variables*
- *Instances*
- *Builds*
- *Revisions*
- *Variants*

On this page we describe how to specify experiments in our `experiments.yml` file. We will see how to use builds in experiments, redirect the output and enable options such as repeating experiments or setting a timeout. Moreover, we will see that simexpal can be used together with batch schedulers (e.g. Slurm, Oracle Grid Engine (formerly SGE), ...), and how to set environment variables.

7.1 Specifying Experiments

In this section we will see how to specify experiments with different kinds of *Instances* and *Variants* by using the keys:

- `name`: name of the experiment
- `args`: list of experiment arguments.

7.1.1 Experiments with Local/Remote Instances

Assuming we have a `./sort.py` file in the same directory as the `experiments.yml` that takes a keyword argument `--algo` and a path to a single (*local*/*remote*) instance as input, we can define our `experiments.yml` as follows:

Listing 1: How to specify experiments for local and remote instances in the experiments.yml file.

```
experiments:
  - name: insertion-sort
    args: ['./sort.py', '--algo=insertion-sort', '@INSTANCE@']
    stdout: out
```

In the examples above we created an experiment named *insertion-sort*. As experiment arguments we have a list of strings (instead of one space separated string). Note that the *@-variable* `@INSTANCE@` resolves to the paths of the instances given in the `instances` stanza, i.e, `/instance_directory/<instance_name>`.

If the instances have *Extra Arguments*, we further need to add the *@-variable* `@EXTRA_ARGS@` to the experiment arguments, e.g, `args: ['./sort.py', '--algo=insertion-sort', '@INSTANCE@', '@EXTRA_ARGS@']` for the example above. `@EXTRA_ARGS@` will resolve to the specified extra arguments of the respective instance (and also the used *variants*, see below).

7.1.2 Experiments with Multiple Extension Instances

Specifying experiments with *multiple extension* instances works similarly to specifying experiments with *local/remote intances*. They only differ in the used *@-variable* in the experiment arguments. Here, we use the @-variable `@INSTANCE:<ext>@`, where `<ext>` is an extension that is specified in the `extensions` key of an instance in the `instances` stanza.

Assuming you have an algorithm that takes a path to a `.graph` and a `.yxz` file as input, you can specify your experiment as follows:

Listing 2: How to specify experiments for multiple extension instances in the experiments.yml file.

```
experiments:
  - name: graph-algorithm
    args: ['./algorithm.py', '@INSTANCE:graph', '@INSTANCE:xyz@']
    stdout: out
```

The `@INSTANCE:graph@` variable will resolve to `/instance_directory/<instance_name>.graph` during runtime. Analogously for the `@INSTANCE:xyz@` variable.

Extra Arguments are handled analogously to the case of *Experiments with Local/Remote Instances*.

7.1.3 Experiments with Arbitrary Input File Instances

Specifying experiments with *arbitrary input file* instances works similarly to specifying experiments with *multiple extension intances*. They only differ in the used *@-variable* in the experiment arguments. Here, we use the @-variable `@INSTANCE:<index>@`, where `<index>` is the index of the desired file specified in the `files` key of an instance in the `instances` stanza. Note that indices start at `0`.

Assuming you have an algorithm that takes two input files as input and you want to pass the path to the first file of the `files` key and then the path to the second file to your algorithm, you can specify your experiment as follows:

Listing 3: How to specify experiments for arbitrary input file instances in the experiments.yml file.

```yaml
experiments:
  - name: algorithm
    args: ['./algorithm.py', '@INSTANCE:0', '@INSTANCE:1@']
    stdout: out
```

The `@INSTANCE:0@` variable will resolve to `/instance_directory/files[0]`, where `files[0]` is the first filename of the `files` key. Analogously for the `@INSTANCE:1` variable.

Extra Arguments are handled analogously to the case of *Experiments with Local/Remote Instances*.

7.1.4 Experiments with Variants

To specify experiments with *Variants* we need to add the `@EXTRA_ARGS@` variable to the experiment arguments:

Listing 4: How to specify experiments with variants in the experiments.yml file.

```yaml
experiments:
  - name: algorithm
    args: ['./algorithm.py', '@INSTANCE@', '@EXTRA_ARGS@']
    stdout: out
```

The `@EXTRA_ARGS@` variable resolves to the extra arguments of all variants (and also the used instance, see above) of the experiment during runtime. For example, assume we have the following `variants` stanza:

```yaml
variants:
  - axis: 'block-algo'
    items:
      - name: 'ba-insert'
        extra_args: ['insertion_sort']
      - name: 'ba-bubble'
        extra_args: ['bubble_sort']
  - axis: 'block-size'
    items:
      - name: 'bs32'
        extra_args: ['32']
      - name: 'bs64'
        extra_args: ['64']
```

Then `@EXTRA_ARGS@` will resolve to

- `'ba-bubble', 'bs32'`,
- `'ba-bubble', 'bs64'`,
- `'ba-insert', 'bs32'` and
- `'ba-insert', 'bs64'`

in the respective experiments.

7.2 Use Builds

On the *Builds* page we explained how to set up automated builds. In order to use those builds for our experiments we need to specify them with the

- `use_builds`: list of used build names

key. Assuming that we have defined `build1` in our `builds` stanza, we can link the build to the experiment as follows:

Listing 5: How to specify used builds for experiments in the experiments.yml file.

```
experiments:
  - name: experiment1
    args: ['<name_of_executable_of_build1>', ...]
    use_builds: [build1]
    ...
```

In this way simexpal will check the *installation directory* and the `extra_paths` of the builds specified in `use_builds` for the executable. If a build *requires other builds* and they are properly specified in the `requires` key, then simexpal will also check the installation directories and `extra_paths` of those builds.

7.3 Output

To redirect the output of an experiment to the `./output/` folder, we specify the

- `stdout`: extension of the output file
- `output`: dictionary containing all output file extensions

keys.

Assume the following `experiments` stanza in our `experiments.yml`:

Listing 6: How to specify the output file extensions for experiments in the experiments.yml file.

```
experiments:
  - name: experiment1
    ...
    stdout: 'out'
    output:
      extensions: ['out', 'foo']
```

Simexpal will then store the outputs in `<instance_name>.out` files, which are located in the

- `./output/<experiment_name>~<variant_names>@<revision_name>`

directory.

Note: In previous versions of simexpal we would specify the `output` key with `'stdout'` as value, i.e `output: 'stdout'`, to achieve the behaviour above. This is deprecated and might be removed in future versions.

The substring `~<variant_names>` only appears, if the experiment has variants. `<variant_names>` will then be a comma separated enumeration of the used variants. The suffix `@<revision_name>` appears if the experiment

uses builds and shows the name of the used revision.

To access the output files with other extensions, we can use the *@-variable* `@OUTPUT:<ext>@`, where `<ext>` is an extension specified in the `extensions` key. This @-variable can be used in the `args` key of experiments and is useful for use cases like the following:

The experiments that we are running store all intermediate steps and results. Thus, when taking a look at the output files, we could encounter thousands (or even more) lines of information even though we might only be interested in the last couple of lines. To avoid this, we add another input parameter, which takes a file path, to our experiments. We then store the final experiment results in this file. Our experiment `args` could then look like this:

- `args: ['experiments.py', '@INSTANCE@', '@OUTPUT:foo@'],`

where the first file path is the path to the instance and the second file path is the path to the output file that contains the final results (`@OUTPUT:foo@` will resolve to the output file with extension `.foo`).

7.4 Repeat

Sometimes it might be useful to validate experiment results by repeating the experiment. In order to avoid duplicating an `experiments` entry we can use the

- `repeat`: integer - number of times an experiment is repeated

key. To repeat an experiment twice we define our `experiments` stanza as follows:

Listing 7: How to specify repetitions for experiments in the experiments.yml file.

```
experiments:
  - name: experiment1
    ...
    repeat: 2
```

The default value of `repeat` is `1`.

7.5 Timeout

It is possible to set a timer for experiments. Once the timer expires, simexpal will terminate the experiment. In order to do so we use the

- `timeout`: integer - timeout in seconds

key. For example we can set the timer for an experiment to two hours as follows:

Listing 8: How to specify a timeout for experiments in the experiments.yml file.

```
experiments:
  - name: experiment1
    ...
    timeout: 7200
```

7.6 Setting Environment Variables

When using APIs like OpenMP it is sometimes necessary to specify settings as environment variables. Thus, simexpal supports setting environment variables in experiments by specifying the

- `environ`: dictionary of (environment variable, value)-pairs

key. For example you can specify the `OMP_NUM_THREADS` environment variable as follows:

Listing 9: How to specify environment variables for experiments in the experiments.yml file.

```
experiments:
  - name: experiment1
    args: ...
    ...
    environ:
      OMP_NUM_THREADS: 2
   - name: experiment2
     args: ...
     ...
     environ:
      OMP_NUM_THREADS: 4
```

7.7 Slurm

7.7.1 sbatch: `--ntasks-per-node`, `-c`, `-N`

When using a job scheduler like Slurm it might be useful to run your software using different node/cpu settings.

Currently, simexpal supports the following three `sbatch` parameters by using its own keywords in the `experiments.yml`:

- `procs_per_node`: number of tasks to invoke on each node (slurm: `--ntasks-per-node=n`)
- `num_threads`: number of cpus required per task (slurm: `-c`, `--cpus-per-task=ncpus`)
- `num_nodes`: number of nodes on which to run (N = min[-max]) (slurm: `-N`, `--nodes=N`)

Listing 10: How to specify supported Slurm parameters for experiments in the experiments.yml file.

```
experiments:
  - name: experiment1
    ...
    num_nodes: 1
    procs_per_node: 24
    num_threads: 2
  - name: experiment2
    ...
    num_nodes: 2
    procs_per_node: 24
    num_threads: 2
```

When launching your experiments with slurm, the line `-N 1 --ntasks-per-node 24 -c 2` will be appended to the sbatch command for `experiment1`. Analogously for `experiment2`.

7.7.2 Arbitrary `sbatch` Arguments

In the section before, we saw how to set the values of three supported `sbatch` arguments. In this section, we will see how to set the value of any supported `sbatch` command. To do so, we use the

- `slurm_args`: list of additional `sbatch` arguments

key. For example, we can set the job name of an experiment by using the `-J` parameter of the `sbatch` command:

Listing 11: How to specify additional Slurm parameters for experiments in the experiments.yml file.

```
experiments:
  - name: experiment1
    ...
    slurm_args: ['-J', 'arbitrary_jobname']
```

7.8 Next

To get a more detailed understanding of experiment variants and fully set up your experiments, you can visit the *Variants* page. If you do not plan on having experiments, you can visit the *Run Matrix* page to modify the experiment combinations that you want to run.

CHAPTER 8

Variants

You might want to take a look at the following pages before exploring variants:

- *Quick Start*
- *@-Variables*
- *Builds*
- *Revisions*
- *Experiments*

When benchmarking algorithms, it is often useful to compare different variants or parameter configurations of the same algorithm. Simexpal can manage those variants without requiring you to duplicate the `experiments` stanza multiple times.

8.1 Specifying Variants

To specify variants we will use the following keys:

- `axis`: name of the variant axis
- `items`: list of dictionaries, which specify variants belonging to the same axis.

For the variants in the `items` list we have to specify the

- `name`: name of the variant
- `extra_args`: list of variant arguments

keys.

Listing 1: How to specify variants in the experiments.yml file.

```yaml
variants:
  - axis: 'block-algo'
    items:
      - name: 'ba-insert'
        extra_args: ['insertion_sort']
      - name: 'ba-bubble'
        extra_args: ['bubble_sort']
  - axis: 'block-size'
    items:
      - name: 'bs32'
        extra_args: ['32']
      - name: 'bs64'
        extra_args: ['64']
```

In this way we created the two variant axes `block-algo` and `block-size`. The former axis has the variants `ba-insert` and `ba-bubble` and the latter has the variants `bs32` and `bs64`. If the `extra_args` contain more than one argument, e.g., a keyword argument `--block-algo ba-insert`, then the arguments are stated separately in a list, i.e. `['--block-algo', 'ba-insert']`.

Experiments will then be created from the cross product of the variants of each axis, i.e, `(ba-insert, bs32)`, `(ba-insert, bs64)`, `(ba-bubble, bs32)` and `(ba-bubble, bs32)` for the `experiments.yml` above.

Note: The elements in each cross product will be sorted lexicographically. This plays a role when the `extra_args` contain positional arguments as they are resolved in the order of the elements in a cross product.

Later we will see how to choose only certain combinations of variants (*Run Matrix*).

8.2 Setting Environment Variables

Note: Environment variables specified in variants will override the values of environment variables specified in the `experiments` stanza (if the same variable occurs).

Setting environment variables for variants works similar to *setting environment variables for experiments*. The difference is that we set the

- `environ`: dictionary of (environment variable, value)-pairs

key for each item in the `items` key. For example you can specify the `OMP_NUM_THREADS` environment variable as follows:

Listing 2: How to specify environment variables for variants in the experiments.yml file.

```yaml
variants:
  - axis: num_threads
    items:
      - name: ONT2
        ...
        environ:
          OMP_NUM_THREADS: 2
```

(continues on next page)

(continued from previous page)

```
      - name: ONT4
        ...
        environ:
          OMP_NUM_THREADS: 4
```

8.3 Slurm: --ntasks-per-node, -c, -N

Note: Supported Slurm arguments specified in variants will override the values of supported Slurm arguments specified in the `experiments` stanza (if the same slurm argument occurs).

The *supported Slurm arguments in experiments* are also supported for variants. Here, we can specify the

- `procs_per_node`: number of tasks to invoke on each node (slurm: `--ntasks-per-node=n`)
- `num_threads`: number of cpus required per task (slurm: `-c`, `--cpus-per-task=ncpus`)
- `num_nodes`: number of nodes on which to run (N = min[-max]) (slurm: `-N`, `--nodes=N`)

keys for each item in the `items` key.

Listing 3: How to specify supported Slurm parameters for variants in the experiments.yml file.

```
variants:
  - axis: num_cores
    items:
      - name: c24
        num_nodes: 1
        procs_per_node: 24
        num_threads: 2
        extra_args: []          # empty extra_args as we only want to benchmark with
                                # different node/cpu settings; do NOT omit this key
      - name: c48
        num_nodes: 2
        procs_per_node: 24
        num_threads: 2
        extra_args: []
```

When launching your experiments with slurm, the variant `c24` will append `-N 1 --ntasks-per-node 24 -c 2` to the sbatch command. Analogously for the experiment with the variant `c48`.

8.4 Next

Now that you have entirely set up your experiments, you can modify the experiment combinations that you want to run. Visit the *Run Matrix* page for a detailed explanation.

CHAPTER 9

Run Matrix

You might want to take a look at the following pages before exploring the run matrix:

- *Quick Start*
- *@-Variables*
- *Instances*
- *Builds*
- *Revisions*
- *Experiments*
- *Variants*

Simexpal will build every possible combination of *experiment*, *instance*, *variant* and *revision*. There are cases where this is not desired. For example, you might only want to run certain instance/variant combinations. On this page we describe how to use the `matrix` key in the `experiments.yml` file and will mainly use the C++ example from the *Quick Start* guide in order to do so.

9.1 Include Experiments

Currently the `matrix` key works by specifying all the experiment combinations that you want to include. Therefore, we will specify the

- `include`: list of dictionaries, which specify included experiment combinations

key.

Each entry of `include` can contain the following keys:

- `experiments`: list of included experiment names
- `instsets`: list of included instance set names
- `axes`: list of included axis names

- `variants`: list of included variant names
- `revisions`: list of included revision names

Simexpal will then build the cross product of each experiment, instance set, variant and revision for each entry in `include`. Omitting a key in `include` will select all possible elements of the respective key, e.g., omitting `revisions` is equivalent to specifying the `revisions` key with all possible revisions.

The `matrix` key of

```yaml
matrix:
  include:
    - experiments: [quick-sort]
      variants: [ba-insert, bs32]
      revisions: [main]
    - experiments: [quick-sort]
      variants: [ba-bubble]
      revisions: [main]
```

(The full `experiments.yml` can be found here .)

results in the following experiments (using `simex experiments list` to display them):

```
Experiment                                  Instance                  Status
----------                                  --------                  ------
quick-sort ~ ba-bubble, bs32 @ main         uniform-n1000-s1          [0] not␣
↪submitted
quick-sort ~ ba-bubble, bs32 @ main         uniform-n1000-s2          [0] not␣
↪submitted
quick-sort ~ ba-bubble, bs64 @ main         uniform-n1000-s1          [0] not␣
↪submitted
quick-sort ~ ba-bubble, bs64 @ main         uniform-n1000-s2          [0] not␣
↪submitted
quick-sort ~ ba-insert, bs32 @ main         uniform-n1000-s1          [0] not␣
↪submitted
quick-sort ~ ba-insert, bs32 @ main         uniform-n1000-s2          [0] not␣
↪submitted
```

Note: If the `axes` key is omitted and the `variants` key does not contain any variant of an axis, then every variant of this axis will be selected. If this is not intended you need to specify the `axes` key with the names of the desired axes.

If we wanted the second entry of the `include` key in the `experiments.yml` above, to only include the variant `ba-bubble` (without any other variants of other axes), we would need to specify the `axes` key like this:

```yaml
matrix:
  include:
    - experiments: [quick-sort]
      variants: [ba-insert, bs32]
      revisions: [main]
    - experiments: [quick-sort]
      axes: [block-algo]
      variants: [ba-bubble]
      revisions: [main]
```

In this way we obtain the experiments:

```
Experiment                                  Instance                  Status
----------                                  --------                  ------
quick-sort ~ ba-bubble @ main               uniform-n1000-s1          [0] not␣
↪submitted
quick-sort ~ ba-bubble @ main               uniform-n1000-s2          [0] not␣
↪submitted
quick-sort ~ ba-insert, bs32 @ main         uniform-n1000-s1          [0] not␣
↪submitted
quick-sort ~ ba-insert, bs32 @ main         uniform-n1000-s2          [0] not␣
↪submitted
```

9.2 Repetitions

Note: Repetitions specified in the run matrix will override the values of `repeat` specified in the `experiments` stanza.

Similarly to *repeating experiments* we can repeat all experiment combinations of an `include` entry by specifying

- `repetitions`: integer - number of times all combinations of an `include` entry are repeated.

To repeat experiments twice, we can define the key as follows:

Listing 1: How to specify repetitions in the matrix key of an experiments.yml file.

```yaml
matrix:
  include:
    - experiments: [...]
      variants: [...]
      ...
      repetitions: 2
```

The default value of `repetitions` is `1`.

9.3 More Examples

9.3.1 Experiments with Different Instance Types

When you are dealing with experiments that have different instance types, e.g. *Multiple Extensions* Instances and *Local Instances*, you need to use *Instance Sets* and the `instsets` key appropriately. This means:

First, you have to assign your instances with different types to separate instance sets. For example, you can assign your multiple extension instances to the instance set `multiple_extension_set` and your local instances to `local_instance_set`. Assuming you have defined two experiments `multiple_ext_experiment` and `local_experiment` that take multiple extension and local instances as input respectively, you can specify your run matrix as follows:

Listing 2: How to specify experiments with different instance types in the run matrix of the experiments.yml file.

```yaml
matrix:
  include:
    - experiments: [multiple_ext_experiment]
      instsets: [multiple_extension_set]
      ...
    - experiments: [local_experiment]
      instsets: [local_instance_set]
      ...
```

In this way, we can assure that the right instance paths are passed to each experiment.

CHAPTER 10

Launcher

On this page we describe the different launchers for experiments. It is possible to use the local machine or batch schedulers like Slurm or Oracle Grid Engine (formerly SGE) to run experiments. The launchers can be selected by adding further arguments to the `simex experiments launch` command or by setting up a `launchers.yml` file. The structure of this file will also be explained on this page.

In the following sections, we consider the sorting expample from the *Quick Start* guide.

10.1 Queue Launcher

The queue launcher is mainly useful in scenarios where no sophisticated batch scheduler (e.g. Slurm) is available, especially when launching experiments on a remote machine over `ssh`. In this case it does not require an active connection. As the name suggests, the queue launcher puts the jobs in a queue and executes them sequentially. It is possible to start a queue launcher as a daemon or interactively in a terminal. You can launch experiments and monitor/stop/kill the launcher.

10.1.1 Starting the Launcher

As Daemon

To start the queue launcher as a daemon you need to have the systemd service manager installed. Then, use `simex queue daemon` to start the launcher:

```
$ simex queue daemon
Running as unit run-rdc0128bd4cd343e393681898359e5c69.service. # systemd output
```

Terminal

You can start the queue launcher in a terminal so that it is possible to directly see internal states of the launcher. You will then work from another terminal window in order to send commands to the launcher. To start the launcher, use

`simex queue interactive`:

```
$ simex queue interactive
Serving on /home/username/.extl.sock  # internal message of the queue launcher
```

Troubleshooting

If the daemon was not closed properly via the `stop` or `kill` command, a UNIX socket remains on the file system. In this case you need to add the `--force` argument to the `simex queue interactive` or `simex queue daemon` command to start the launcher. Alternatively, you can delete the socket manually. The default path for the socket is `~/.extl.sock`.

10.1.2 Launching Experiments

To launch experiments through the queue launcher we need to *set up a launchers.yml file* or add the `--launch-through=queue` argument to the `simex experiments launch` command:

```
$ simex experiments launch --launch-through=queue
Launching run bubble-sort/uniform-n1000-s1[0] on local machine
Launching run bubble-sort/uniform-n1000-s2[0] on local machine
Launching run bubble-sort/uniform-n1000-s3[0] on local machine
Launching run insertion-sort/uniform-n1000-s1[0] on local machine
Launching run insertion-sort/uniform-n1000-s2[0] on local machine
Launching run insertion-sort/uniform-n1000-s3[0] on local machine
```

10.1.3 Show the Status

It is possible to query the launcher for the current run, pending runs and the number of completed runs. To do that, you can use `simex queue show`:

```
$ simex queue show
Currently running:   bubble-sort/uniform-n1000-s3[0]
Pending runs:        insertion-sort/uniform-n1000-s1[0]
                     insertion-sort/uniform-n1000-s2[0]
                     insertion-sort/uniform-n1000-s3[0]
Completed runs:      2
```

10.1.4 Terminate

The launcher can be terminated by using the `stop` or `kill` command. The differences of those commands are stated below.

- `simex queue stop`: terminate the launcher after finishing all pending jobs
- `simex queue kill`: terminate the launcher immediately

10.2 “launchers.yml” File

The `launchers.yml` file contains a list of launchers. By setting up a `launchers.yml` file we can omit additional arguments in the `simex experiments launch` command or select launchers, which are defined in it.

In the following sections, we will see how to setup and use our `launchers.yml`. First, we need to create the `launchers.yml` file in the `~/.simexpal/` folder:

```
$ mkdir ~/.simexpal      # Create the ~/.simexpal/ folder if it does not exist already
$ cd ~/.simexpal         # Navigate into ~/.simexpal/
$ touch launchers.yml    # Create an empty launchers.yml file
```

10.2.1 Launchers

To specify launchers in the `launchers.yml` file we need to set the

- `launchers`: list of dictionaries, which contain launchers

key.

Queue Launcher

To define a queue launcher we need to add a list entry to the `launchers` key, which contains a dictionary with the

- `name`: name of the launcher
- `default`: boolean (`true`/`false`) - whether this is the default launcher or not
- `scheduler`: type of the launcher

key.

Listing 1: How to specify a queue launcher in the launchers.yml file.

```
launchers:
    - name: local-queue
      default: true
      scheduler: queue
```

In this way we created a queue launcher with the name `local-queue`. We also set it to be the default launcher.

10.2.2 Command Line Interface

Default Launcher

When setting `default: true` for a launcher, `simex experiments launch` will run experiments with this launcher.

Warning: There can only be one launcher with `default: true`. Having multiple launchers with `default: true` will lead to a `RuntimeError`.

Selecting the Launcher

When launching experiments using `simex experiments launch`, you can specify the `--launcher` option to select a certain launcher defined in the `launchers.yml` file. For example:

Assume you have a `launchers.yml` file set up as in the *Queue Launcher* section, then `simex experiments launch --launcher local-queue` will select the launcher named `local-queue` from the `launchers.yml` file to run experiments.

CHAPTER 11

Command-line Reference

Simexpal instructions are written as:

```
simex <instruction> [action] [selection option] [args...]
```

In the following, we list all simexpal instructions and their arguments.

11.1 archive

Archives all the experimental data into a `data.tar.gz` file within the same directory where the `experiments.yml` file is located.

11.2 builds

Responsible to download Git repositories and install executables. It supports the following actions:

make downloads a Git repository and executes all build commands.

purge deletes all related build files. To confirm this action it needs the `-f` argument.

remake rebuilds a build from scratch.

All the above actions can be applied to a subset of builds according to the `--revisions` and positional `builds` argument. Not specifying an argument will select all respective elements, e.g., not specifying the `--revisions` argument will lead to the selection of every revision.

11.3 develop

Responsible to download Git repositories and install executables. It also allows you to redo arbitrary build steps after changing local Git files to take over the local changes.

The `develop` action can be applied to a subset of builds according to the `--revisions` and positional `builds` argument (analogously to how it works for the *simex builds* command above).

It further supports the following additional arguments:

`--checkout` deletes the local Git repository and (re-)clones it.

`--compile` (re-)compiles the build.

`--configure` (re-)configures the build.

`--delete-source` deletes the source directory when purging.

`--install` (re-)installs the build.

`--purge` deletes all related build files.

`--recheckout` deletes the local build Git repository, reclones, regenerates, reconfigures, recompiles and reinstalls it.

`--recompile` recompiles and reinstall the build.

`--reconfigure` reconfigures, recompiles and reinstalls the build.

`--regenerate` (re-)regenerates the build.

`--reinstall` reinstalls the build.

`--reregenerate` regenerates, reconfigures, recompiles and reinstalls the build.

The `--checkout`, `--recheckout` and `--purge` arguments further require the `-f` argument to confirm their actions.

11.4 experiments

Responsible to check, execute and remove experiments. It supports the following actions:

info displays all related instances, instance sets, variant axes and variants of experiments on the command-line.

list lists all the experiments. Executed experiments are shown in green, failed ones are shown in red, running ones in yellow, and the non executed ones in the default command-line color. With the argument `--detailed` it will show every single run. With `--compact` all runs with the same experiment will be grouped together. The `--full` option forces simexpal to display the full experiment name.

launch launches all the non executed experiments.

purge deletes the experimental data. To confirm this action it needs the `-f` argument.

print displays all experimental output, including error outputs, on the command-line.

All the above actions can be applied to a subset of experiments according to a *selection option*, which can be specified as an additional argument. Supported selection options are:

`--all` selects all the experiments.

`--axes [axes...]` selects all experiments with the variant axes from the space separated list of axes.

`--run <r>` selects the single run given as `<experiment_display_name>/<instance>`, where `<experiment_display_name>` is the name of the experiment as displayed on the command-line and `<instance>` is the instance name as displayed on the command-line.

`--experiment <e>` selects the experiment named `e`.

`--failed` selects all the failed experiments.

--instance <i> selects all experiments with the instance named `i`.

--instset <i> selects all experiments with the instance set named `i`.

--unfinished selects all the unfinished experiments.

--revision <i> selects all experiments with the revision named `r`.

--variants [variants...] selects all experiments with the variants from the space separated list of variants.

11.5 instances

Checks and eventually downloads the instances for the experiments. It supports the following actions:

list lists all the instances found in the experiments.yml file. Available instances are shown in green, unavailable instances in red.

install downloads all the missing instances if they are taken from a public repository. With the argument `--overwrite` it will also download the available instances and overwrite them.

process caches information about instances.

run-transform manually runs a transformation on instance files.

CHAPTER 12

Python API reference

12.1 The "base" module

class simexpal.base.**Config**(*basedir*, *yml*)
Represents the entire configuration (i.e., an experiments.yml file).

collect_successful_results(*parse_fn=None*)
Collects all successful runs and optionally parses their output.

Parameters **parse_fn** – Function to parse the output. Takes two parameters `(run, f)` where `run` is a `simexpal.base.Run` object and `f` is a Python file object.

Returns list of parsed outputs if `parse_fn` is given, generator of successful `simexpal.base.Run` objects otherwise

export_experiments(*included_statuses=None*)
Exports experiments based on their status.

Parameters **included_statuses** – List of *simexpal.base.Status* objects which get exported. Alternatively an integer list can be used, where the following translation is used: 0 - not submitted, 1 - in submission, 2 - submitted, 3 - started, 4 - finished, 5 - timeout, 6 - killed, 7 - failed.

Returns List of (`exp_name`, `variation-tuple`, `inst_shortname`, `status`)-tuples

instance_dir()
Path of the directory that stores all the instances.

class simexpal.base.**Experiment**(*cfg*, *info*, *revision*, *variation*)
Represents an experiment (see below).

An experiment is defined as a combination of command line arguments and environment (from the experiment stanza in a experiments.yml file), a revision that is used to build the experiment's program and a set of variants (from the variants stanza in a experiments.yml file).

class simexpal.base.**Instance**(*cfg*, *inst_yml*, *index*)
Represents a single instance

class simexpal.base.**Status**
An enumeration.

CHAPTER 13

Recipes

13.1 Automated Builds of Python Packages

Python packages are often installed using `pip3` (or `pip`) and use a different directory layout than native programs. In particular, they do not make use of directories like `bin/`, `lib/` or `include/`. To still support automated builds of Python packages, simexpal offers the `exports_python` property for builds. This property takes a directory name (relative to `@THIS_PREFIX_DIR@`) and exports this name via the `PYTHONPATH` environment variable such that the Python interpreter is able to load packages from this directory. It can be used in conjunction with the `--target` option of `pip3` to locally install Python packages during automated builds.

```
builds:
  - name: some_python_package
    exports_python: 'python-packages/'
    # [...]
    install:
      - args: ['pip3', 'install', '--target=@THIS_PREFIX_DIR@/python-packages',
↪'@THIS_SOURCE_DIR@']
```

Note that on Debian-based Linux distributions, you need to pass `--system` to `pip3` to override the default of `--user` (which does not work with `--target`).

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Python Module Index

S

Index