acud Documentation

Release 1.0

acud

Sep 21, 2018

Contents

acud is a lightweight capacity expansion, economic dispatch and unit commitment simulation tool written in Python, which uses csv and Excel files as input and output data format. It is an optimisation-based calculation engine that leverages multiple state-of-art data handling Python packages including pandas and xarray, powered by open source (e.g. glpk, cbc) or commercial (e.g. cplex, gurobi) solvers.

CHAPTER 1

User guide

1.1 Overview

The optimisations underlying the acud simulations are defined as a Linear Programming (LP) or Mixed-Integer Linear Programming (MIP) problem, depending on the desired level of accuracy and complexity. acud is able to handle multiple time resolutions, producing both chronological and load duration curve-based results.

1.1.1 Features

- Capacity expansion
- Minimum and maximum power for each unit
- Generation unit derating
- Power plant ramping limits
- Minimum up/down times
- Curtailment
- Detailed hydro modelling, including hydro cascades
- Pumped storage
- Non-dispatchable units (e.g. wind, solar, run-of-river, etc.)
- Start-up, ramping and no-load costs
- Net transfer capacity based network modelling
- Constraints on the targets for renewables and/or CO2 emissions

1.1.2 Output

- Timing, location and type of new capacity built

- Nodal energy prices
- Scheduled generation
- Generating unit starts and stops
- Fuel consumption
- Emissions
- Interconnector flows
- Storage volumes, water releases and spillages
- Pumped energy
- Unserved energy

1.2 Formulation

A partial formulation of the chronological MT stage mathematical optimization problem can be found below.

1.3 Installation

We strongly recommend using a dedicated conda environment to install acud.

Install the package with pip:

```
$ pip install git+[acud git repository url here]
```

1.4 Input preparation

The acud simulation input data is specified through a set of text files which can be grouped as follows:

- configuration yaml files
- content csv files

1.4.1 Configuration yaml files

Referred to in short as yml files, these are two YAML-format files:

- run configuration yml file, and
- model configuration yml file

which jointly define the model outline.

The model yml file specifies the more "structural" data whereas the run yml file contains information related to the analysis to be performed (e.g. which solver to use or the time horizon to be considered). The split of data between run and model configuration data is somewhat arbitrary, the purpose of this split being to ease the reuse of information contained in the model configuration file. In this sense, there is a certain hierarchy between the two config files, with the run file at the top level. Indeed, the run yml file can be thought of as the trigger file: it is the only file that needs to be directly specified when invoking an acud command (the model configuration file is invoked internally through the acud code using the model entry of the run configuration file).

The yml files themselves do store some of the values applied in the acud simulation, e.g. discount rate or VOLL, but most of their content is made up of references to other files (mostly csv files) where the data is actually stored.

yml files store nested mappings of key-value pairs, where values can be either single values, lists of values or further key-value pairs. Mappings are nested through indentation, e.g.

```
timeseries:
    generator:
        derating:
            format: ymdp
            files:
               COAL: input/derating_COAL.csv
```

In the example above the key-value pair COAL: input/derating_COAL.csv is a value to the files key which in turn together with the format key-value pair is a value to the derating key, and so on.

IMPORTANT: use space NOT tab for indentation to ensure correct yaml file parsing.

In these notes we refer to the most inner entry of nesting mappings using a "flattened" dot-based notation, e.g. COAL files entry in the example above would be represented like this

```
timeseries.generator.derating.files.COAL: input/derating_COAL.csv
```

Within a given map nesting level, the order in which the entries are written in the configuration files is not relevant but it is probably best to follow a certain convention about it.

Below we provide a commented version of the run configuration file. As YAML allows comments (after #), the text below is valid config file content.

```
name: basecase_run  # run name, can be different from model name
model: model.yml  # name of (and path to) file with model configuration data
default: default.yml # name of (and path to) file with parameter default values

time:
    startperiod: 2012-03-01 00:00  # use this datetime format
    endperiod: 2012-03-31 23:00 # last period included in modelling horizon
    stepsize: 24 # number of hours in optimization step
    blocks: 3 # placeholder, currently not in use

output:
    format: csv  # only format currently supported
    path: solution  # results destination, will be created if it doesn't exist
    drypath: dry # dry run output destination, will be created if it doesn't exist

settings:
    mode: quantity  # placeholder, currently not in use
    phase: MT  # LT, MT or ST
    solver: glpk # CBC or glpk
    cat: Integer # Linear or Integer solution for integer variables
```

Similarly, a commented version of a sample model configuration file follows.

```
name: basecase   # model name

components:
    bus:
        file: input/buses.csv
        referencenode: GB # if no bus specified for generators, assume they are␣
↪connected to this bus
```

(continues on next page)

(continued from previous page)

```
    branch:
        file: input/branches.csv
    generator:
        file: input/gens.csv
    fuel: # can be empty, but needs to be included for the fuel timeseries input␣
↪specified below to be processed

timeseries:  # this information further qualifies the component instances created␣
↪based on the components entries above
    generator:
        derating:  # this entry could also be a time invariant parameter, in which␣
↪case there would  be no need for a timeseries entry
            format: ymdp
            files: # as the shape is ymdp, each file contains information referring␣
↪to a single component instance
                COAL: input/derating_COAL.csv
                OCGT: input/derating_OCGT.csv
                CCGT: input/derating_CCGT.csv
                OIL: input/derating_OIL.csv
                NUCLEAR: input/derating_NUCLEAR.csv
                OTHER: input/derating_OTHER.csv
                WIND: input/derating_WIND.csv
                PS: input/derating_PS.csv

        units:
            format: dtname
            datetimeformat: '%d/%m/%Y' # as in input data file, the internal format␣
↪is different
            files:  # can be a list of multiple files which will overwrite the␣
↪configuration derived from the component table entries
               - input/units_WIND.csv
    fuel:
        price:
            format: dtname
            datetimeformat: '%d/%m/%Y'
            files:
               - input/fuels.csv

    bus:
        load:
            format: ymdp
            files:
               ES: input/load.csv

scalars:   # these are values that are not indexed neither by datetime nor by␣
↪component instance name
    discountrate: 0
    voll: 5000
    pricedumpenergy: 5000
    loadescalator: 1
    branchescalator: 1
```

yaml templates

Configuration yaml files are a key ingredient of an acud input dataset. They contain a number of fields grouped by sections and subsections, and can be easily created and updated using the simplest of text editors. However, some

complication arise when dealing with configuration files:

- they must abide to the [YAML file format](http://yaml.org/) e.g. regarding indentation
- they involve a long and nested list of required and optional fields.

YAML file templates tackle these two complications. They are well formatted (i.e correct indentation and markup) and contain placeholders for all available fields -whether optional or required- with descriptive comments attached.

Let RUN_CONFIG be an existing yaml file at the case folder location where you want to create new yaml files (note that for the purpose of automatically creating a template-based yaml file, RUN_CONFIG does *not* need to be a properly built acud run configuration file).

To create a draft run config file from an acud template, use the following command:

```
ad dry RUN_CONFIG --run
```

Similarly, for a model or a default file:

```
ad dry RUN_CONFIG --model
ad dry RUN_CONFIG --default
```

You can also create multiple templates simultaneously:

```
ad dry RUN_CONFIG --run --model --default
ad dry RUN_CONFIG --run --model
ad dry RUN_CONFIG --run --default
```

and so on.

This command will create a file called, respectively, run.yml, model.yml and default.yml at the location where RUN_CONFIG is stored. If there is already a file called run.yml, model.yml or default.yml at the destination folder it will *not* be overwritten. Instead, the template-based yaml files will be renamed as run(1).yml, run(2).yml and so on. Extracts of the run, model and default templates look, respectively, like this:

```
# RUN CONFIGURATION

name: <run name>
model: <model yaml file>        # Relative path to model configuration yaml file
#default: <default yaml file>    # Relative path to user-defined default yaml file␣
↪that overrides built-in defaults

time:
    startperiod: <YYYY-MM-DD hh:mm>                # Starting hour of simulation␣
↪horizon
    endperiod: <YYYY-MM-DD hh:mm>                  # Last hour of simulation horizon
#    stepsize: [int]                                # Number of hours in optimization␣
↪step
#    blocks: [int]                                  # Number of blocks applied in␣
↪load duration curve approximation
#    snapshots:
#        file: <snapshots csv filename>             # Full filename to snapshots␣
↪specification
#        datetimeformat: <Python datetime format>   # Pattern used when parsing␣
↪datetime field, see http://strftime.org/ for syntax reference

output:
    path: <solution folder>                   # will be created if it doesn't exist
    drypath: <dry run output folder>          # will be created if it doesn't exist
```

(continues on next page)

(continued from previous page)

```
    xtrapath: <extrapolated solution folder>  # will be created if it doesn't exist
#    format: [csv]                            # File format of reported solution
#    scenario: [scenario name]                # Label for data handling
#    out_lp:  [0 or 1]                        # Switch to output text file with␣
↪problem formulation in CPLEX LP format
#    out_mps:  [0 or 1]                       # Switch to output text file with␣
↪problem formulation in MPS format
```

```
# MODEL CONFIGURATION

name: <model name>

components:
    bus:
        file: <filepath>                          # Table file location
        referencenode: <string>                   # System wide energy price is set␣
↪to nodal price at bus reference

    generator:
        file: <filepath>                          # Table file location
        datetimeformat: <Python datetime format>  # Pattern used when parsing date␣
↪from and date to field, see http://strftime.org/ for syntax reference

#    branch:
#        file: <filepath>                          # Table file location
#
#    fuel:
#        file: <filepath>                          # Table file location

scalars:
#    discountrate: [float]              # Yearly rate to discount cash flows in␣
↪objective function of capacity expansion problem
#    thermalfirm: [0 <= float <= 1]     # Firmness factor of thermal generating units␣
↪in capacity expansion problem
```

```
# SOLVE OPTIONS
solve:
    mode: quantity  # price, quantity
    phase: MT  # LT, MT
    solver: coin  # coin, glpk, xpress, gurobi
    cat: Integer # Integer, Binary or Continuous
    scenario: Base
    allow_use: 1  # allow unserved energy
    allow_dump: 1  # allow dump energy
    include_ramps: 1  # consider max ramp up limits
    include_msl: 1  # consider min stable level
    include_no_load_costs: 1 # consider no load costs
    include_start_costs: 1  # consider start costs
    epgap: 0.0001  # mipgap tolerance termination
    sec: 7200

# TIME OPTIONS
time:
    stepsize: 24
    blocks: 3
```

(continues on next page)

(continued from previous page)

```
# OUTPUT OPTIONS
output:
    verbose: 0 # amount of progress info printed
    out_csv: 1 # write csv files
```

Composing the yaml file is then just a matter of editing the placeholders in the template and saving changes. Note that only the default.yml is ready to use as provided; in this case the file does not include placeholders but rather built-in default values. The run.yml and model.yml need further editing by replacing the placeholder included in the template following the field code described below:

- <item> denotes that user-defined input is required, i.e. acud provides no default value for this entry.
- [item] denotes that a default value is available for this entry, i.e. if you do not provide a value for this entry then acud will apply a default value during the simulation.
- Commented out entries are optional entries; when uncommented during the template modification, these entries need to be further edited by replacing <item> or [item] placeholders with the appropriate value.

1.4.2 Content csv files

Content files are a variable number of csv files containing the bulk of the input data. There are two types of csv files:

- table files
- time series files

Non-data rows can also be included in csv files, regardless of their type. Rows starting with a # character will be ignored by the acud computations.

IMPORTANT: the acud code is case sensitive so upper and lower cases of component instance and parameter names need to match.

Table files

Table files lists the instances of the different components included in the model. There should be one table file per component type contained in the model. The first columns shows the name of the instance, the rest of the columns store various parameters, which will vary depending on the component type.

Component instance parameters can be either numerical, denoting measures of quantifiable metrics, or categorical, representing relationships. To illustrate, generators have numerical parameters such as heat rate or max capacity, and categorical parameters such as bus (to which it connects) or fuel (burned).

When acud reads the input data it starts with the table files. Reading through these files it creates the different component instances that comprise the model, populating their parameters either with provided or default values.

acud input processing then moves on to time series files which further qualify the information provided through the table files.

Time series files

Time series files come in two (for now) shapes:

- dtname
- ymdp

You need to specify the shape of the time series files through the format entry found in the model configuration yml file, e.g. like this:

```
bus.load.format: ymdp
```

or

```
fuel.price.format: dtname
```

In csv files with dtname shape the first column contains datetime values and the header for this column is "datetime". The rest of the columns to the right store parameter values, with each column containing the values of a single component instance. The names of these component instance are stored as headers of the value columns. To avoid faulty datetime parsing it is best to explicitly specify the datetime format through the entry:

```
timeseries.<component type>.<parameter>.datetimeformat
```

csv files with ymdp shape store year, month and day values in the left columns. Each of the remaining columns contain the parameter values, with each column storing values for the same period (of different days). Periods are shown as column headers. csv files with ymdp shape do not include any component instance name information, which means that each csv files contains timeseries data corresponding to a single component instance.

In principle most if not all parameters can be made to hold time series. However the mathematical optimisation problem formulation assumes that only a given subset of parameters are time dependent, with all other parameters being time invariant.

The time dependent parameters in the optimization problem formulation are as follows.

For generators:

- max capacity
- max build
- cincr (incremental costs)
- units
- derating
- bid
- bid escalator

For buses:

- load

For fuels:

- price

All other parameters not listed above will be handled as time invariant in the optimization problem formulation. When a time series is specified in the simulation input data for a time invariant parameter, the value of the time series at the start of the modelling horizon will be assigned to such parameter.

1.5 Input validation

acud input data validation functionality runs two types of input data checks:

- yaml (files) validation
- csv (files) validation

yaml validation covers both run and model configuration files and runs checks such as:

- Can the run and model configuration yaml files be found?
- Do the configuration yaml files include all required fields
- Are field values provided of the right type and within allowed values?

csv validation performs the following checks on input csv files:

- Structural checks - ensure that there are no empty rows, no blank headers, etc.
- Content checks - ensure that the values have the correct types ("string", "number", etc.), that their values are allowed ("datetime format must be either dtname or ymdp"), and that they respect the constraints ("forced outage must be a number greater than 0").

1.5.1 Basic usage

From the command line, change the current directory to the case folder you want to inspect:

```
cd my_case_folder
```

Validate yaml configuration files only:

```
ad validate RUN
```

where RUN stands for the name of the run configuration yaml file (including the file extension, e.g. run_base_case.yml).

Run the yaml validation and display the name of the configuration files that were inspected:

```
ad validate RUN --verbose
```

Run both yaml and csv validations:

```
ad validate RUN --all
```

or just:

```
ad validate RUN -a
```

Again, display the names of the inspected input data files:

```
ad validate RUN --all --verbose
```

Built-in schema files contain the input data validation rules applied for both yaml and csv files, e.g. whether a field is required or not, a field´s name, type, allowed values, etc.

List the built-in schema files:

```
ad schema -l ACME
```

ACME above stands for just any string (but you do need to provide one).

Display a built-in schema specification, e.g. for generators:

```
ad schema generator
```

If you find the schema difficult to read on screen you can write it to a file for easier inspection like this: ad schema generator > check_schema.txt

This will write a text file named check_schema.txt into your current directory.

List all available acud commands and a brief description of each one of them:

```
ad --help
```

Get help about the new validation commands:

```
ad validate --help
```

or:

```
ad schema --help
```

1.5.2 Important usage notes

Datetime formats

run yaml configuration files includes fields startperiod and endperiod expecting datetime values. There is only one datetime format allowed to specify these fields in the run configuration yaml file:

```
%Y-%m-%d %H:%M
```

This is a slightly modified version of the ISO8601 standard datetime format: we drop the seconds, the time zone and replace the 'T' character separating date and time with a blank space. This means that in the run configuration yaml file January, 2, 2018 at 11pm looks like this:

```
2018-01-02 23:00
```

On the other hand, multiple datetime formats are allowed in the timeseries csv files. These formats need to be specified by the user in the corresponding timeseries datetimeformat field included in the model configuration yaml file.

Datetime column position

Timeseries csv files with dtname format MUST include the datetime field in the leftmost column.

1.5.3 Implementation

Most of the heavylifting of the acud input data validation is performed by two mature and well maintained third-party Python packages, namely Cerberus and goodtables-py. Cerberus is a lightweight, extensible data validation library for Python. goodtables-py is a Python framework to validate tabular data.

More information about these packages can be found at http://docs.python-cerberus.org/en/stable/ and https://github.com/frictionlessdata/goodtables-py, respectively. For additional information on table schemas also check https://frictionlessdata.io/specs/table-schema.

1.6 Simulation timeline

A typical acud simulation timeline is a sequence of contiguous hours between the start and the end simulation datetimes, both included. For sizeable models with an extended timeline, this can lead to very long runtimes (hours) and massive hardrive memory consumption. A workaround for this is to run acud simulations with a snapshots-based timeline.

1.6.1 Snapshots-based simulation timeline

When running a simulation through a snapshots-based timeline, , you can specify a set of (non-contiguous) datetimes at which the optimization should be run. If required, these snapshots-based simulation results can then be extrapolated, assigning the nearest hourly results to non-computed datetimes (e.g. the extrapolated result of a non-computed datetime happening at 7a.m. will be the last computed result at 7 a.m.).

Snapshots are specified by:

1. Listing snapshots in a csv file with a single field called datetime, and
2. Referencing this csv file in a `time.snapshots.file` entry in the run configuration yml file.

Note that you also need to provide the date format in the time.snapshots.datetimeformat entry to ensure that the snapshot dates and times are correctly interpreted by acud. An extract of the time section of a run yml file including snapshots-based timeline specification is shown below.

```
time:
    startperiod: 2015-01-01 00:00
    endperiod: 2018-12-31 23:00
    stepsize: 24
    blocks: 3
    snapshots:
        file: csv/snapshots.csv
        datetimeformat: '%d/%m/%Y'
```

To further illustrate, these are the first lines of an example snapshots csv file. Note that the dates are non-contiguous.

```
datetime
01/01/2018
03/01/2018
10/01/2018
20/01/2018
```

Snapshots override start and end datetime specifications. If no snapshots specification is provided in the run config yml file, the simulation timeline will consist of the sequence of contiguous hours between the start and the end datetime, both included.

After completing a snapshots-based simulation, you can *extrapolate* the sparse timeline results to obtain a dense timeline solution using the `xtra` command, e.g. like this:

```
ad xtra run.yml
```

CHAPTER 2

Command reference

The `ad` command is the primary interface for managing acud simulations.

acud provides several commands for data handling and execution of power market simulations. The links on this page provide help for each command. You can also access help from the command line with the `--help` flag:

```
ad run --help
```

TIP: You can abbreviate many frequently used command options that are preceded by 2 dashes (`--`) to just 1 dash and the first letter of the option. So `--list` and `-n` are the same.

2.1 Execution commands

Execution of acud simulations is managed through the following commands:

2.1.1 ad run

2.1.2 ad xtra

2.2 Data commands

acud input data handling commands include:

2.2.1 ad dry

2.2.2 ad schema

2.2.3 ad validate

CHAPTER 3

Developer guide

3.1 Unit testing

The acud developer distribution includes basic pytest-based unit testing. To run these unit tests you first need to install pytest either via conda:

```
conda install pytest
```

or pip:

```
pip install pytest
```

Once pytest is installed, set the test folder as current directory:

```
cd test
```

To run all available pytest unit test:

```
py.test -v
```

or

```
python -m pytest -v
```

To run a specific unit test:

```
py.test test_api.py -v
```

or

```
python -m pytest test_cli.py -v
```

Note that passing the validate test included in the unit test suite means that the validation process completed successfully, not that the validation tests passed. To review the validation test results, run the validate command separately through the command line or via the acud Python api.

CHAPTER 4

License

acud is released under the MIT license with the following terms:

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

4.1 Dependencies

versioneer.py is Public Domain.