smif Documentation

Release 0.3.2

Will Usher
Tom Russell

April 24, 2017

Contents

Simulation Modelling Integration Framework

CHAPTER 1

Description

smif is a framework for handling the creation of system-of-systems models. The framework handles inputs and outputs, dependencies between models, persistence of data and the communication of state across years.

This early version of the framework handles simple models that simulate the operation of a system. **smif** will eventually implement optimisation routines which will allow, for example, the solution of capacity expansion problems.

CHAPTER 2

Setup and Configuration

smif is written in Python (Python>=3.5) and has a number of dependencies. See *requirements.txt* for a full list.

GLPK

The optimisation routines currently use GLPK - the GNU Linear Programming Kit. To install the **glpk** solver:

- on Linux or Mac OSX, you can likely use a package manager, e.g. `apt install python-glpk glpk-utils` for Ubuntu or `brew install glpk` for OSX.
- on Windows, GLPK for Windows provide executables. For 64bit Windows, download and unzip the distribution files then add the `w64` folder to your `PATH`.

fiona, GDAL and GEOS

We use fiona, which depends on GDAL and GEOS libraries.

On Mac or Linux these can be installed with your OS package manager, then install the python packages as usual using:

```
# On debian/Ubuntu:
apt-get install gdal-bin libspatialindex-dev libgeos-dev

# or on Mac
brew install gdal
brew install spatialindex
brew install geos

pip install -r requirements.txt
```

On Windows, the simplest approach seems to be using conda, which handles packages and virtual environments, along with the *conda-forge* channel which has a host of pre-built libraries and packages.

Create a conda environment:

```
conda create --name smif python=3.5 numpy scipy
```

Activate it (run each time you switch projects):

```
activate smif
```

Note that you `source activate smif` on OSX and Linux.

Add the conda-forge channel, which has shapely and fiona available:

```
conda config --add channels conda-forge
```

Install python packages, along with GDAL and dependencies:

```
conda install fiona shapely rtree
pip install -r requirements.txt
```

CHAPTER 3

Installing *smif*

Once the dependencies are installed on your system, a normal installation of *smif* can be achieved using pip on the command line:

```
pip install smif
```

Versions under development can be installed from github using pip too:

```
pip install git+http://github.com/nismod/smif#egg=v0.2
```

The suffix `#egg=v0.2` refers to a specific version of the source code. Omitting the suffix installs the latest version of the library.

To install from the source code in development mode:

```
git clone http://github.com/nismod/smif
cd smif
python setup.py develop
```

CHAPTER 4

A word from our sponsors

smif was written and developed at the Environmental Change Institute, University of Oxford within the EPSRC sponsored MISTRAL programme, as part of the Infrastructure Transition Research Consortium.

CHAPTER 5

Contents

Getting Started

To specify a system-of-systems model, you must configure one or more simulation models, outlined in the section below, and configure a system-of-systems model, as outlined immediately below.

First, setup a new system-of-systems modelling project with the following folder structure:

```
/config
/planning
/data
/models
```

This folder structure is optional, but helps organise the configuration files, which can be important as the number and complexity of simulation models increases.

The `config` folder contains the configuration for the system-of-systems model:

```
/config/model.yaml
/config/timesteps.yaml
```

The `planning` folder contains one file for each

```
/planning/pre-specified.yaml
```

The `data` folder contains a subfolder for each sector model:

```
/data/<sector_model_1>
/data/<sector_model_2>
```

The `/data/<sector_model>` folder contains all the configuration files for a particular sector model. See adding a sector model for more information.:

```
/data/<sector_model>/inputs.yaml
/data/<sector_model>/outputs.yaml
/data/<sector_model>/time_intervals.yaml
/data/<sector_model>/regions.geojson
/data/<sector_model>/interventions.yaml
```

The `/models/<sector_model/` contains the executable for a sector model, as well as a Python file which implements *smif.sector_model.SectorModel* and provides a way for *smif* to run the model, and access model outputs. See adding a sector model for more information.:

```
/models/<sector_model>/run.py
/models/<sector_model>/<executable or library>
```

System-of-Systems Model File

The `model.yaml` file contains the following:

```
timesteps: timesteps.yaml
region_sets:
- name: energy_regions
  file: regions.shp
interval_sets:
- name: energy_timeslices
  file: time_intervals.yaml
- name: annual_interval
  file: annual_interval.yaml
scenario_data:
- file: electricity_demand.yaml
  parameter: electricity_demand
  spatial_resolution: energy_regions
  temporal_resolution: annual_interval
- file: gas_demand.yaml
  parameter: gas_demand
  spatial_resolution: energy_regions
  temporal_resolution: annual_interval
sector_models:
- name: energy_supply
  path: ../../../models/energy_supply/run.py
  classname: EnergySupplyWrapper
  config_dir: .
  initial_conditions:
  - initial_conditions.yaml
  interventions:
  - interventions.yaml
planning:
  pre_specified:
    use: true
    files:
    - pre-specified.yaml
  rule_based:
    use: false
    files: []
  optimisation:
    use: false
    files: []
```

System-of-Systems Planning Years

The `timesteps.yaml` should contain a list of planning years:

```
- 2010
- 2011
- 2012
```

This is a list of planning years over which the system of systems model will run. Each of the simulation models will be run once for each planning year.

Wrapping a Sector Model

To integrate a sector model into the system-of-systems model, it is necessary to write a Python wrapper, which implements *smif.sector_model.SectorModel*.

The key methods which need to be overridden are:

- *smif.sector_model.SectorModel.simulate()*

- *smif.sector_model.SectorModel.extract_obj()*

The path to the location of the `run.py` file should be entered in the `model.yaml` file under the `path` key (see System-of-Systems Model File above).

To integrate an infrastructure simulation model within the system-of-systems modelling framework, it is also necessary to provide the following configuration data.

Geographies

Define the set of unique regions which are used within the model as polygons. Inputs and outputs are assigned a model-specific geography from this list allowing automatic conversion from and to these geographies.

Model regions are specified in `regions.*`.

The file format must be possible to parse with GDAL, and must contain an attribute "name" to use as an identifier for the region.

The sets of geographic regions are specified in the `model.yaml` file using a `region_sets` attributes as shown below:

```
region_sets:
- name: energy_regions
  file: regions.shp
```

This links a name, used elsewhere in the configuration with inputs, outputs and scenarios with a file containing the geographic data.

Temporal Resolution

The attribution of hours in a year to the temporal resolution used in the sectoral model.

Within-year time intervals are specified in yaml files, and as for regions, specified in the `model.yaml` file with an `interval_sets` attribute:

```
interval_sets:
- name: energy_timeslices
  file: time_intervals.yaml
- name: annual_interval
  file: annual_interval.yaml
```

This links a unique name with the definitions of the intervals in a yaml file. The data in the file specify the mapping of model timesteps to durations within a year (assume modelling 365 days: no extra day in leap years, no leap seconds)

Each time interval must have

- start (period since beginning of year)
- end (period since beginning of year)
- id (label to use when passing between integration layer and sector model)

use ISO 8601 [1] duration format to specify periods:

```
P[n]Y[n]M[n]DT[n]H[n]M[n]S
```

For example:

```
- end: P7225H
  name: '1_0'
  start: P7224H
- end: P7226H
```

[1] https://en.wikipedia.org/wiki/ISO_8601#Durations

```
  name: '1_1'
  start: P7225H
- end: P7227H
  name: '1_2'
  start: P7226H
- end: P7228H
  name: '1_3'
  start: P7227H
- end: P7229H
  name: '1_4'
  start: P7228H
```

Inputs

Define the collection of inputs required from external sources to run the model. For example "electricity demand (<region>, <interval>)". Inputs are defined with a name, spatial resolution and temporal-resolution.

Only those inputs required as dependencies are defined here, although dependencies are activated when configured in the system-of-systems model.

The `inputs.yaml` file defines the dependencies of one model upon another. Enter a list of dependencies, each with three keys, `name`, `spatial_resolution` and `temporal_resolution`. For example, in energy supply:

```
- name: electricity_demand
  spatial_resolution: energy_regions
  temporal_resolution: annual_interval
- name: gas_demand
  spatial_resolution: energy_regions
  temporal_resolution: annual_interval
```

The keys `spatial_resolution` and `temporal_resolution` define the resolution at which the data are required.

Outputs

Define the collection of outputs model parameters used for the purpose of optimisation or rule-based planning approaches (so normally a cost-function), and those outputs required for accounting purposes, such as operational cost and emissions, or as a dependency in another model.

The `outputs.yaml` file defines the output parameters from the model. For example:

```
- name: total_cost
  spatial_resolution: energy_regions
  temporal_resolution: annual_interval
- name: water_demand
  spatial_resolution: energy_regions
  temporal_resolution: annual_interval
- name: total_emissions
  spatial_resolution: energy_regions
  temporal_resolution: annual_interval
```

Scenarios

The `scenario_date:` section of the system-of-systems configuration file allows you to define static sources for simulation model dependencies.

In the case of the example show above, reproduced below:

```
scenario_data:
- file: electricity_demand.yaml
  parameter: electricity_demand
  spatial_resolution: energy_regions
  temporal_resolution: annual_interval
- file: gas_demand.yaml
  parameter: gas_demand
  spatial_resolution: energy_regions
  temporal_resolution: annual_interval
```

we define two yaml files, one each for the parameters *electricity_demand* and *gas_demand*. The `temporal_resolution` attribute allows the use of time intervals in the scenario files which are at a different temporal resolution to that expected by the sector model. In this case, both electricity_demand and gas_demand are linked to the same `annual_interval.yaml` file.

The scenario data should contain entries for (time_interval) `name`, region, value, units and timestep (year). For example:

```
- name: 1_0
  region: "England"
  value: 23.48
  units: GW
  year: 2015
- name: 1_1
  region: "England"
  value: 17.48
  units: GW
  year: 2015
  - name: 1_2
  region: "England"
  value: 16.48
  units: GW
  year: 2015
```

State Parameters

Some simulation models require that state is passed between years, for example reservoir level in the water-supply model. These are treated as self-dependencies with a temporal offset. For example, the sector model depends on the result of running the model for a previous timeperiod.

Interventions

An Intervention is an investment which has a name (or name), other attributes (such as capital cost and economic lifetime), and location, but no build date.

An Intervention is a possible investment, normally an infrastructure asset, the timing of which can be decided by the logic-layer.

An exhaustive list of the Interventions (normally infrastructure assets) should be defined. These are represented internally in the system-of-systems model, collected into a gazateer and allow the framework to reason on infrastructure assets across all sectors. Interventions are instances of *`Intervention`* and are held in *`InterventionRegister`*. Interventions include investments in assets, supply side efficiency improvements, but not demand side management (these are incorporated in the strategies).

Define all possible interventions in an `interventions.yaml` file. For example:

```
- name: nuclear_power_station_england
  capital_cost:
    value: 3.5
    units: £(million)/MW
  economic_lifetime:
```

```
    value: 30
    units: years
  operational_life:
    value: 40
    units: years
  operational_Year:
    value: 2030
    units: year
  capacity:
    value: 1000
    units: MW
  location:
    value: England
    units: string
  power_generation_type:
    value: 4
    units: number
- name: IOG_gas_terminal_expansion
  capital_cost:
    value: 10
    units: £(million)/mcm
  economic_lifetime:
    value: 25
    units: years
  operational_life:
    value: 30
    units: years
  operational_Year:
    value: 2020
    units: year
  capacity:
    value: 10
    units: mcm
  location:
    value: England
    units: string
  gas_terminal_number:
    value: 8
    units: number
```

Planning

Existing Infrastructure

Existing infrastructure is specified in a `*.yaml` file. This uses the following format:

```
- name: CCGT
  description: Existing roll out of gas-fired power stations
  timeperiod: 1990 # 2010 is the first year in the model horizon
  location: "oxford"
  new_capacity:
    value: 6
    unit: GW
  lifetime:
    value: 20
    unit: years
```

Pre-Specified Planning

A fixed pipeline of investments can be specified using the same format as for existing infrastructure, in the `*.yaml` files.

The only difference is that pre-specified planning investments occur in the future (in comparison to the initial modelling date), whereas existing infrastructure occur in the past. This difference is semantic at best, but a warning is raised if future investments are included in the existing infrastructure files in the situation where the initial model timeperiod is altered.

Define a pipeline of interventions in a `pre-specified.yaml` file:

```
- name: nuclear_power_station_england
  build_date: 2017
```

Rule Based Planning

This feature is not yet implemented

Optimisation

This feature is not yet implemented

References

Concept

The section outlines the underlying principles of **smif**.

Running a System-of-Systems Model

Once **smif** has been used to configure a system-of-systems model, all that is needed to run the model is the command `smif run`.

smif handles the loading of input data, spinning up the simulation models, extracting a graph of dependencies from the network of inputs and outputs, running the models in the order defined by this graph and finally persisting state and results from the simulation models to a data store.

Operational Simulation and Capacity Expansion

Fundamental to the design of **smif** is the distinction between the simulation of the operation of a particular system, and the long-term expansion of the capacity which underpin this operation.

The former is the domain of the simulation models, while the latter is handled by **smif**. **smif** provides the architecture to handle the capacity expansion problem using one of three approaches: a fully specified approach, a rule based approach and an optimisation approach.

In each of these three approaches, decisions regarding the increase or decrease in the capacity of an asset are propagated into the model inputs via a *state transition function*.

State

State refers to the information which must be persisted over time. Normally, this will refer to the capacity of an asset (e.g. number of wind turbines), the level of storage (e.g. the volume of water stored in a reservoir). Other information, including metrics, such as CO_2 emissions, or cumulative costs, may also be relevant.

smif handles *State* for the management of the capacity expansion. The process of passing state from one time-period to another is managed by **smif**. In this respect, note the distinction between time-steps for the capacity expansion problem, which will normally be measured in years or decades, versus the time-steps for each instance of a simulation model, which will run within a year or decade.

Wrapping Simulation Models

At the core of **smif** are the target simulation models which we wish to integrate into a system-of-systems model. A simple example simulation model is included in `tests.fixtures.water_supply.ExampleWaterSupplySimulation`. A simulation model has inputs, and produces outputs, which are a function of the inputs. The `smif.abstract.SectorModel` is used to wrap an individual simulation model, and provides a uniform API to other parts of **smif**.

An input can correspond to:

- model parameters, whose source is either from a scenario, or from the outputs from another model (a dependency)
- model state (not yet implemented)

Developing *smif*

smif is under active development at github.com/nismod/smif

Testing

We use pytest for testing, with tests under `tests/` matching the module and class structure of `smif/`.

Install requirements for testing:

```
pip install -r test-requirements.txt
```

Run tests:

```
python setup.py test
```

Versioning

`smif` is currently pre-1.0, so API and usage are liable to change. After releasing a first major version, we intend to follow semantic versioning, with major versions for any incompatible changes to the public API.

Releases

smif is deployed as a package on the Python Package Index, PyPI. A full guide to packaging and distributing projects is *available online <https://packaging.python.org/distributing/>*

To make a release, first register with PyPI and contact a project owner (currently Will Usher or Tom Russell) to be made a maintainer.

Set up a *.pypirc* file in your home directory with your access details:

```
[distutils]
index-servers =
    pypi

[pypi]
repository: https://pypi.python.org/pypi
username: <username>
password: <password>
```

Create an annotated tag for release:

```
git tag -a v0.2.0       # create annotated tag (will need a message)
git describe            # show current commit in relation to tags
git push origin v0.2.0  # push the tag to the origin remote repository
```

Create a source distribution (this creates a gzipped package in *dist*):

```
python setup.py sdist
ls dist/
```

Use twine to upload the distribution:

```
pip install twine
twine upload dist/smif-0.2.0.tar.gz
```

Conventions

The numpydoc docstring conventions are used for inline documentation, which is used to generate the module reference documentation visible at readthedocs and which can also be generated by running `python setup.py docs`

Linting is handled by pre-commit hooks, which can be installed from the root of the repository using:

```
pre-commit install
```

How to contribute

First things first, thank you for considering contributing to *smif*!

Getting started

Please raise an issue for bugs spotted and features proposed.

Github pull requests can be made against the master branch so long as we continue prototyping before a v1.0 release.

See the developer documentation for details of testing and other conventions.

License

```
The MIT License (MIT)

Copyright (c) 2017 Will Usher, Tom Russell

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.
```

Developers

- Will Usher <william.usher@ouce.ox.ac.uk>
- Tom Russell <tom.russell@ouce.ox.ac.uk>

Changelog

Version 0.3

- Fast, more compact YAML
- Input, output and pre-specified planning files can now be empty
- State is passed between successive time steps
- Interdependencies (cycles in dependencies) are now supported, models are run in cycles stopping at convergence or timeout
- Non-unique time interval definitions are supported

Version 0.2

- Basic conversion of time intervals (aggregation, disaggregation, remapping) and regions (aggregation, disaggregation)
- Results are written out in a yaml dump with the `-o` flag e.g. `smif run -o results.yaml model.yaml`
- Single one-way dependencies with spatio-temporal conversion are supported
- Simplified and harmonised implementation of model inputs and outputs

Version 0.1

- Run a single simulation model for a single timestep
- Provide a model with scenario data and planned interventions
- Configure a model with sets of regions and sets of time intervals for within- timestep simulation

smif

smif package

Subpackages

smif.cli package

Module contents A command line interface to the system of systems framework

This command line interface implements a number of methods.

- *setup* creates a new project folder structure in a location
- ***run* performs a simulation of an individual sector model, or the whole** system of systems model
- *validate* performs a validation check of the configuration file

Folder structure When configuring a system-of-systems model for the CLI, the folder structure below should be used. In this example, there is one sector model, called `water_supply`:

```
/main_config.yaml
/timesteps.yaml
/water_supply.yaml
/data/all/inputs.yaml
/data/water_supply/
/data/water_supply/inputs.yaml
/data/water_supply/outputs.yaml
/data/water_supply/assets/assets1.yaml
/data/water_supply/planning/
/data/water_supply/planning/pre-specified.yaml
```

The `data` folder contains one subfolder for each sector model.

The sector model implementations can be installed independently of the model run configuration. The main_config.yaml file specifies which sector models should run, while each set of sector model config

`smif.cli.`**`confirm`**(*prompt=None*, *response=False*)

Prompts for a yes or no response from the user

Parameters

- **`prompt`** (*`str, default=None`*) –
- **`response`** (*`bool, default=False`*) –

Returns True for yes and False for no.

Return type bool

Notes

response should be set to the default value assumed by the caller when user simply types ENTER.

Examples

```
>>> confirm(prompt='Create Directory?', response=True)
Create Directory? [y]|n:
True
>>> confirm(prompt='Create Directory?', response=False)
Create Directory? [n]|y:
```

```
    False
    >>> confirm(prompt='Create Directory?', response=False)
    Create Directory? [n]|y: y
    True
```

smif.cli.**log_validation_errors**()
Log validation errors

smif.cli.**main**(*arguments=None*)
Parse args and run

smif.cli.**parse_arguments**()
Parse command line arguments

Returns

Return type argparse.ArgumentParser

smif.cli.**path_to_abs**(*relative_root*, *path*)
Return an absolute path, given a possibly-relative path and the relative root

smif.cli.**read_sector_model_data**(*config_basepath*, *config*)
Read sector-specific data from the sector config folders

smif.cli.**run_model**(*args*)
Runs the model specified in the args.model argument

smif.cli.**setup_configuration**(*args*)
Sets up the configuration files into the defined project folder

smif.cli.**setup_project_folder**(*project_path*)
Creates folder structure in the target directory

Parameters **project_path** (*str*) – Absolute path to an empty folder

smif.cli.**validate_config**(*args*)
Validates the model configuration file against the schema

Parameters **args** – Parser arguments

smif.convert package

Submodules

smif.convert.area module Handles conversion between the sets of regions used in the *SosModel*

class smif.convert.area.**NamedShape**(*name*, *shape*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *name*, *shape*)
Create new instance of NamedShape(name, shape)

__repr__()
Return a nicely formatted representation string

name
Alias for field number 0

shape
Alias for field number 1

class smif.convert.area.**RegionRegister**
Bases: object

Holds the sets of regions used by the SectorModels and provides conversion between data values relating to compatible sets of regions.

convert(*data*, *from_set_name*, *to_set_name*)
Convert a list of data points for a given set of regions to another set of regions.

Parameters

- **data** (*dict*) –
- **from_set_name** (*str*) –
- **to_set_name** (*str*) –

get_regions_in_set(*set_name*)
Return regions for a given set

region_set_names
Names of registered region sets

Returns sets

Return type list of str

register(*region_set*)
Register a set of regions as a source/target for conversion

class smif.convert.area.**RegionSet**(*set_name*, *fiona_shape_iter*)
Bases: object

Hold a set of regions, spatially indexed for ease of lookup when constructing conversion matrices.

Parameters

- **set_name** (*str*) – Name to use as identifier for this set of regions
- **fiona_shape_iter** (*iterable*) – Iterable (probably a list or a reader handle) of fiona feature records e.g. the 'features' entry of a GeoJSON collection

intersection(*bounds*)
Return the subset of regions intersecting with a bounding box

smif.convert.area.**proportion_of_a_intersecting_b**(*shape_a*, *shape_b*)
Calculate the proportion of shape a that intersects with shape b

smif.convert.interval module Handles conversion between the set of time intervals used in the *SosModel*

There are three main classes, which are currently rather intertwined. *Interval* represents an individual definition of a period within a year. This is specified using the ISO8601 period syntax and exposes methods which use the isodate library to parse this into an internal hourly representation of the period.

TimeIntervalRegister holds the definitions of time-interval sets specified for the sector models at the *SosModel* level. This class exposes one public method, `add_interval_set()` which allows the SosModel to add an interval definition from a model configuration to the register.

TimeSeries is used to encapsulate any data associated with a time interval definition set, and handles conversion from the current time interval resolution to a target time interval definition held in the register.

Quantities Quantities are associated with a duration, period or interval. For example 120 GWh of electricity generated during each week of February.:

```
Week 1: 120 GW
Week 2: 120 GW
Week 3: 120 GW
Week 4: 120 GW
```

Other examples of quantities:

- greenhouse gas emissions
- demands for infrastructure services
- materials use
- counts of cars past a junction
- costs of investments, operation and maintenance

Upscale: Divide To convert to a higher temporal resolution, the values need to be apportioned across the new time scale. In the above example, the 120 GWh of electricity would be divided over the days of February to produce a daily time series of generation. For example:

```
1st Feb: 17 GWh
2nd Feb: 17 GWh
3rd Feb: 17 GWh
...
```

Downscale: Sum To resample weekly values to a lower temporal resolution, the values would need to be accumulated. A monthly total would be:

```
Feb: 480 GWh
```

Remapping Remapping quantities, as is required in the conversion from energy demand (hourly values over a year) to energy supply (hourly values for one week for each of four seasons) requires additional averaging operations. The quantities are averaged over the many-to-one relationship of hours to time-slices, so that the seasonal-hourly timeslices in the model approximate the hourly profiles found across the particular seasons in the year. For example:

```
hour 1: 20 GWh
hour 2: 15 GWh
hour 3: 10 GWh
...
hour 8592: 16 GWh
hour 8593: 12 GWh
hour 8594: 21 GWh
...
hour 8760: 43 GWh
```

To:

```
season 1 hour 1: 20+16+.../4 GWh # Denominator number hours in sample
season 1 hour 2: 15+12+.../4 GWh
season 1 hour 3: 10+21+.../4 GWh
...
```

Prices Unlike quantities, prices are associated with a point in time. For example a spot price of £870/GWh. An average price can be associated with a duration, but even then, we are just assigning a price to any point in time within a range of times.

Upscale: Fill Given a timeseries of monthly spot prices, converting these to a daily price can be done by a fill operation. E.g. copying the monthly price to each day.

From:

```
Feb: £870/GWh
```

To:

```
1st Feb: £870/GWh
2nd Feb: £870/GWh
...
```

Downscale: Average On the other hand, going down scale, such as from daily prices to a monthly price requires use of an averaging function. From:

```
1st Feb: £870/GWh
2nd Feb: £870/GWh
...
```

To:

```
Feb: £870/GWh
```

Development Notes

- We could use `numpy.convolve()` to compare time intervals as hourly arrays before adding them to the set of intervals

class `smif.convert.interval.`**Interval**(*name*, *list_of_intervals*, *base_year=2010*)
Bases: `object`

A time interval

Parameters

- **id** (*str*) – The unique name of the Interval
- **list_of_intervals** (*str*) – A list of tuples of valid ISO8601 duration definition string denoting the time elapsed from the beginning of the year to the (beginning, end) of the interval
- **base_year** (*int, default=2010*) – The reference year used for conversion to a datetime tuple

Example

```
>>> a = Interval('id', ('PT0H', 'PT1H'))
>>> a.interval = ('PT1H', 'PT2H')
>>> repr(a)
"Interval('id', [('PT0H', 'PT1H'), ('PT1H', 'PT2H')], base_year=2010)"
>>> str(a)
"Interval 'id' starts at hour 0 and ends at hour 1"
```

baseyear
The reference year

end
The end hour of the interval(s)

Returns A list of integers, representing the hour from the beginning of the year associated with the end of each of the intervals

Return type list

interval
The list of intervals

Setter appends a tuple or list of intervals to the list of intervals

name

start
The start hour of the interval(s)

Returns A list of integers, representing the hour from the beginning of the year associated with the start of each of the intervals

Return type list

to_hourly_array()
Converts a list of intervals to a boolean array of hours

to_hours()
Return a list of tuples of the intervals in terms of hours

Returns A list of tuples of the start and end hours of the year of the interval

Return type list

class smif.convert.interval.**TimeIntervalRegister**(*base_year=2010*)
Bases: object

Holds the set of time-intervals used by the SectorModels

Parameters **base_year** (*int, default=2010*) – Set the year which is used as a reference by all time interval sets and repeated for each future year

convert(*timeseries*, *from_interval*, *to_interval*)
Convert some data to a time_interval type

Parameters

- **timeseries** (*TimeSeries*) – The timeseries to convert from *from_interval* to *to_interval*
- **from_interval** (*str*) – The unique identifier of a interval type which matches the timeseries
- **to_interval** (*str*) – The unique identifier of a registered interval type

Returns A dictionary with keys *name* and *value*, where the entries for *key* are the name of the target time interval, and the values are the resampled timeseries values.

Return type dict

get_intervals_in_set(*set_name*)

Parameters **set_name** (*str*) – The unique identifying name of the interval definitions

Returns Returns a collection of the intervals in the order in which they were defined

Return type collections.OrderedDict

interval_set_names
A list of the interval set names contained in the register

Returns

Return type list

register(*intervals*, *set_name*)
Add a time-interval definition to the set of intervals types

Detects duplicate references to the same annual-hours by performing a convolution of the two one-dimensional arrays of time-intervals.

Parameters

- **intervals** (*list*) – Time intervals required as a list of dicts, with required keys `start`, `end` and `name`
- **set_name** (*str*) – A unique identifier for the set of time intervals

class smif.convert.interval.**TimeSeries**(*data*)

Bases: object

A series of values associated with an interval definition

Parameters **data** (*list*) – A list of dicts, each entry containing 'name' and 'value' keys

hourly_values

The timeseries resampled to hourly values

Module contents In this module, we implement the conversion across space and time

The *SpaceTimeConvertor* is instantiated with data to convert, and the names of the four source and destination spatio-temporal resolutions.

The *convert()* method returns a new list of *smif.SpaceTimeValue* namedtuples for passing to a sector model.

class smif.convert.**SpaceTimeConvertor**(*data*, *from_spatial*, *to_spatial*, *from_temporal*, *to_temporal*, *region_register*, *interval_register*)

Bases: object

Handles the conversion of time and space for a list of values

Parameters

- **data** (*list*) – A list of *smif.SpaceTimeValue*
- **from_spatial** (*str*) – The name of the spatial resolution of the data
- **to_spatial** (*str*) – The name of the required spatial resolution
- **from_temporal** (*str*) – The name of the temporal resolution of the data
- **to_temporal** (*str*) – The name of the required temproal resolution
- **region_register** (*smif.convert.area.RegionRegister*) – A fully populated register of the models' regions
- **interval_register** (*smif.convert.interval.TimeIntervalRegister*) – A fully populated register of the models' intervals

Notes

Future development requires using a data object which allows multiple views upon the values across the three dimensions of time, space and units. This will then allow more efficient conversion across any one of these dimensions while holding the others constant. One option could be collections.ChainMap.

convert()

Convert the data according to the parameters passed to the SpaceTimeConvertor

Returns A list of *smif.SpaceTimeValue*

Return type list

data_by_region

data_regions

smif.data_layer package

Submodules

smif.data_layer.load module Parse yaml config files, to construct sector models

smif.data_layer.load.**dump**(*data*, *file_path*)
Write plain data to a file as yaml

Parameters

- **file_path** (*str*) – The path of the configuration file to write
- **data** – Data to write (should be lists, dicts and simple values)

smif.data_layer.load.**load**(*file_path*)
Parse yaml config file into plain data (lists, dicts and simple values)

Parameters **file_path** (*str*) – The path of the configuration file to parse

smif.data_layer.load.**space_time_value_constructor**(*loader*, *node*)
Load ustom yaml representation of SpaceTimeValue

smif.data_layer.load.**space_time_value_representer**(*dumper*, *data*)
Dump custom yaml representation of SpaceTimeValue

smif.data_layer.sector_model_config module Read and parse the config for sector models

class smif.data_layer.sector_model_config.**SectorModelReader**(*initial_config=None*)
Bases: object

Parses the configuration and input data for a sector model

Parameters **initial_config** (*dict*) – Sector model details, sufficient to read the full config from a set of files. Must contain the following fields:

"model_name" The name of the sector model, for reference within the system-of-systems model

"model_path" The path to the python module file that contains an implementation of SectorModel

"model_classname" The name of the class that implements SectorModel

"model_config_dir" The root path of model config/data to use, which must contain inputs.yaml, outputs.yaml, time_intervals.yaml and regions.shp/regions.geojson

"initial_conditions" List of files containing initial conditions

"interventions" List of files containing interventions

data
Expose all loaded config data

Returns

data – Model configuration data, with the following fields:

"name": The name of the sector model, for reference within the system-of-systems model

"path": The path to the python module file that contains an implementation of SectorModel

"classname": The name of the class that implements SectorModel

"inputs": A list of the inputs that this model requires

"outputs": A list of the outputs that this model provides

"time_intervals": A list of time intervals within a year that are represented by the model, each with reference to the model's internal identifier for timesteps

"regions": A list of geographical regions used within the model, as objects with both geography and attributes

"initial_conditions": A list of initial conditions required to set up the modelled system in the base year

"interventions": A list of possible interventions that could be made in the modelled system

Return type dict

load()
Load and check all config

load_initial_conditions()
Inital conditions are located in yaml files specified in sector model blocks in the sos model config

load_inputs()
Input spec is located in the `data/<sectormodel>/inputs.yaml` file

load_interventions()
Interventions are located in yaml files specified in sector model blocks in the sos model config

load_io_metadata(*inputs_or_outputs*)
Load inputs or outputs, allowing missing or empty file

load_outputs()
Output spec is located in `data/<sectormodel>/output.yaml` file

smif.data_layer.sos_model_config module Read and parse the config files for the system-of-systems model

class smif.data_layer.sos_model_config.**SosModelReader**(*config_file_path*)
Bases: object

Encapsulates the parsing of the system-of-systems configuration

Parameters **config_file_path** (*str*) – A path to the master config file

data
Expose all model configuration data

Returns

Returns a dictionary with the following keys:

timesteps the sequence of years

max_iterations limit iterations for solving interdependencies

sector_model_config: list The list of sector model configuration data

scenario_data: dict A dictionary of scenario data, with the parameter name as the key and the data as the value

planning: list A list of dicts of planning instructions

region_sets: dict A dictionary of region set data, with the name as the key and the data as the value

interval_sets: dict A dictionary of interval set data, with the name as the key and the data as the value

resolution_mapping: dict The mapping of spatial and temporal resolutions to inputs, outputs and scenarios

Return type dict

load()
Load and check all config

load_max_iterations()
Parse max_iterations setting

load_planning()
Loads the set of build instructions for planning

Returns A list of planning instructions loaded from the planning file

Return type list

load_regions()
Model regions are specified in `data/<sectormodel>/regions.*`

The file format must be possible to parse with GDAL, and must contain an attribute "name" to use as an identifier for the region.

load_scenario_data()
Load scenario data from list in sos model config

Working assumptions:

•scenario data is list of dicts, each like:

```
{
    'parameter': 'parameter_name',
    'file': 'relative file path',
    'spatial_resolution': 'national'
    'temporal_resolution': 'annual'
}
```

•data in file is list of dicts, each like:

```
{
    'value': 100,
    'units': 'kg',
    # optional, depending on parameter type:
    'region': 'UK',
    'year': 2015
}
```

Returns A dictionary where keys are parameters names and values are the file contents, so a list of dicts

Return type dict

load_sector_model_data()
Parse list of sector models to run

Model details include: - model name - model config directory - SectorModel class name to call

load_sos_config()
Parse model master config

•configures run mode

•sets max iterations for solving interdependencies

•points to timesteps file

•points to shared data files

•points to sector models and sector model data files

load_time_intervals()
Within-year time intervals are specified in `data/<sectormodel>/time_intervals.yaml`

These specify the mapping of model timesteps to durations within a year (assume modelling 365 days: no extra day in leap years, no leap seconds)

Each time interval must have - start (period since beginning of year) - end (period since beginning of year) - id (label to use when passing between integration layer and sector model)

use ISO 8601[1]_ duration format to specify periods:

```
P[n]Y[n]M[n]DT[n]H[n]M[n]S
```

For example:

```
P1Y == 1 year
P3M == 3 months
PT168H == 168 hours
```

So to specify a period from the beginning of March to the end of May:

```
start: P2M
end: P5M
id: spring
```

References

load_timesteps()
Parse model timesteps

smif.data_layer.validate module Validate the correct format and presence of the config data for the system-of-systems model

exception smif.data_layer.validate.**ValidationError**
Bases: Exception

Custom exception to use for parsing validation.

smif.data_layer.validate.**validate_dependency**(*dep*)
Check a dependency specification

smif.data_layer.validate.**validate_initial_condition**(*datum*, *file_path*)
Check a single initial condition datum

smif.data_layer.validate.**validate_initial_conditions**(*data*, *file_path*)
Check a list of initial condition observations

smif.data_layer.validate.**validate_input_spec**(*input_spec*, *model_name*)
Check the input specification for a single sector model

smif.data_layer.validate.**validate_interval_sets_config**(*interval_sets*)
Check interval sets

smif.data_layer.validate.**validate_interventions**(*data*, *path*)
Validate the loaded data as required for model interventions

smif.data_layer.validate.**validate_output**(*dep*)
Check an output specification

smif.data_layer.validate.**validate_output_spec**(*output_spec*, *model_name*)
Check the output specification for a single sector model

smif.data_layer.validate.**validate_path_to_timesteps**(*timesteps*)
Check timesteps is a path to timesteps file

smif.data_layer.validate.**validate_planning_config**(*planning*)
Check planning options

smif.data_layer.validate.**validate_region_sets_config**(*region_sets*)
Check regions sets

smif.data_layer.validate.**validate_scenario**(*scenario*)
Check a single scenario specification

smif.data_layer.validate.**validate_scenario_data**(*data*, *file_path*)
Check a list of scenario observations

smif.data_layer.validate.**validate_scenario_data_config**(*scenario_data*)
Check scenario data

smif.data_layer.validate.**validate_scenario_datum**(*datum*, *file_path*)
Check a single scenario datum

smif.data_layer.validate.**validate_sector_model_initial_config**(*sector_model_config*)
Check a single sector model initial configuration

smif.data_layer.validate.**validate_sector_models_initial_config**(*sector_models*)
Check list of sector models initial configuration

smif.data_layer.validate.**validate_sos_model_config**(*data*)
Check expected values for data loaded from master config file

smif.data_layer.validate.**validate_time_interval**(*interval*)
Check a single time interval

smif.data_layer.validate.**validate_time_intervals**(*intervals*, *file_path*)
Check time intervals

smif.data_layer.validate.**validate_timesteps**(*timesteps*, *file_path*)
Check timesteps is a list of integers

Module contents Data access modules for loading system-of-systems model configuration

Submodules

smif.decision module

The decision module handles the three planning levels

Currently, only pre-specified planning is implemented.

class smif.decision.**Planning**(*planned_interventions=None*)
Bases: object

Holds the list of planned interventions, where a single planned intervention is an intervention with a build date after which it will be included in the modelled systems.

For example, a small pumping station might be built in Oxford in 2045:

```
{
    'name': 'small_pumping_station',
    'build_date': 2045
}
```

planned_interventions
list

A list of pre-specified planned interventions

names
Returns the set of assets defined in the planned interventions

timeperiods
Returns the set of build dates defined in the planned interventions

smif.intervention module

This module handles the collection of interventions and assets in a sector. The set of interventions describes the targets of possible physical (or non-physical) decisions which can be made in the sector.

An Asset is the general term for an existing component of an infrastructure system.

The difference between an Intervention and an Asset, is that the latter exists (it has been "built"), whereas the former describes the potential to build an Asset.

The set of assets defines the 'state' of the infrastructure system.

Notes

This module implements:

- initialisation of the set of assets from model config (either a collection of yaml text files, or a database)
  - hold generic list of key/values
  - creation of new assets by decision logic (rule-based/optimisation solver)
  - maintain or derive set of possible assets
  - makes the distinction between known-ahead values and build-time values. Location and date are specified at build time, while cost and capacity are a function of time and location.
- serialisation for passing to models
  - ease of access to full generic data structure
- output list of assets for reporting
  - write out with legible or traceable keys and units for verification and understanding

Terminology

name: A category of infrastructure intervention (e.g. power station, policy) which holds default attribute/value pairs. These names can be inherited by asset/intervention definitions to reduce the degree of duplicate data entry.

asset: An instance of an intervention, which represents a single investment decisions which will take place, or has taken place. Historical interventions are defined as initial conditions, while future interventions are listed as pre-specified planning. Both historical and future interventions can make use of names to ease data entry. Assets must have `location`, `build_date` and `name` attributes defined.

intervention: A potential asset or investment. Interventions are defined in the same way as for assets, cannot have a `build_date` defined.

class `smif.intervention.`**`Asset`**(*name=''*, *data=None*, *sector=''*)
Bases: *smif.intervention.Intervention*

An instance of an intervention with a build date.

Used to represent pre-specified planning and existing infrastructure assets and interventions

Parameters

- **`name`** (*`str, default=""`*) – The type of asset, which should be unique across all sectors
- **`data`** (*`dict, default=None`*) – The dictionary of asset attributes
- **`sector`** (*`str, default=""`*) – The sector associated with the asset

`build_date`
The build date of this asset instance (if specified - asset types will not have build dates)

`get_attributes`()
Ensures location is present and no build date is specified

class smif.intervention.**AssetRegister**
Bases: *smif.intervention.Register*

Register each asset type

register(*asset*)

class smif.intervention.**Intervention**(*name=''*, *data=None*, *sector=''*)
Bases: *smif.intervention.InterventionContainer*

An potential investment to send to the logic-layer

An Intervention, is an investment which has a name (or name), other attributes (such as capital cost and economic lifetime), and location, but no build date.

The set of interventions are defined within each sector, and these are collected into an *InterventionRegister* when a smif.controller.SosModel is instantiated by the controller at runtime.

Parameters

- **name** (*str, default=""*) – The type of asset, which should be unique across all sectors
- **data** (*dict, default=None*) – The dictionary of asset attributes
- **sector** (*str, default=""*) – The sector associated with the asset

get_attributes()
Ensures location is present and no build date is specified

location
The location of this asset instance (if specified - asset types may not have explicit locations)

class smif.intervention.**InterventionContainer**(*name=''*, *data=None*, *sector=''*)
Bases: object

An container for asset types, interventions and assets.

An asset's data is set up to be a flexible, plain data structure.

Parameters

- **name** (*str, default=""*) – The type of asset, which should be unique across all sectors
- **data** (*dict, default=None*) – The dictionary of asset attributes
- **sector** (*str, default=""*) – The sector associated with the asset

static deterministic_dict_to_str(*data*)
Return a reproducible string representation of any dict

Parameters **data** (*dict*) – An intervention attributes dictionary

Returns A reproducible string representation

Return type str

get_attributes()
Override to return two lists, one containing required attributes, the other containing omitted attributes

Returns Tuple of lists, one contained required attributes, the other which must be omitted

Return type tuple

sector
The name of the sector model this asset is used in.

sha1sum()
Compute the SHA1 hash of this asset's data

Returns

Return type str

class smif.intervention.**InterventionRegister**
Bases: *smif.intervention.Register*

The collection of Intervention objects

An InterventionRegister contains an immutable collection of sector specific assets and decision points which can be decided on by the Logic Layer

- Reads in a collection of interventions defined in each sector model
- Builds an ordered and immutable collection of interventions
- Provides interfaces to
  - optimisation/rule-based planning logic
  - SectorModel class model wrappers

Key functions:

- outputs a complete list of asset build possibilities (asset type at location) which are (potentially) constrained by the pre-specified planning instructions and existing infrastructure.
- translate a binary vector of build instructions (e.g. from optimisation routine) into Asset objects with human-readable key-value pairs
- translates an immutable collection of Asset objects into a binary vector to pass to the logic-layer.

Notes

Internal data structures

Intervention_types is a 2D array of integers: each entry is an array representing an Intervention type, each integer indexes attribute_possible_values

attribute_keys is a 1D array of strings

attribute_possible_values is a 2D array of simple values, possibly (boolean, integer, float, string, tuple). Each entry is a list of possible values for the attribute at that index.

Invariants

- there must be one name and one list of possible values per attribute
- each Intervention type must list one value for each attribute, and that value must be a valid index into the possible_values array
- each possible_values array should be all of a single type

__iter__()
Iterate over the list of asset types held in the register

__len__()
Returns the number of asset types stored in the register

attribute_index(*key*)
Get the index of an attribute name

attribute_value_index(*attr_idx*, *value*)
Get the index of a possible value for a given attribute index

get_intervention(*name*)
Returns the named asset data

numeric_to_intervention(*numeric_asset*)
Convert the numeric representation of an asset back to Asset (with legible key/value data)

Parameters **numeric_asset** (*list*) – A list of integers of length *self._attribute_keys*

Returns An *Intervention* object

Return type *Intervention*

Examples

Given a (very minimal) possible state of a register:

```
>>> register = AssetRegister()
>>> register._names = [[1,1,1]]
>>> register._attribute_keys = ["name", "capacity", "sector"]
>>> register._attribute_possible_values = [
...     [None, "water_treatment_plant"],
...     [None, {"value": 5, "units": "ML/day"}],
...     [None, "water_supply"]
... ]
```

Calling this function would piece together the asset:

```
>>> asset = register.numeric_to_asset([1,1,1])
>>> print(asset)
Asset("water_treatment_plant", {"name": "water_treatment_plant",
"capacity": {"units": "ML/day", "value": 5}, "sector": "water_supply"})
```

register(*intervention*)
Add a new intervention to the register

Parameters **intervention** (*Intervention*) –

class smif.intervention.**Register**
Bases: object

Holds interventions, pre-spec'd planning instructions & existing assets

- register each asset type/intervention name
- translate a set of assets representing an initial system into numeric representation

register(*asset*)
Adds a new asset to the collection

smif.optimisation module

Solve the optimal planning problem for a system of systems model

smif.optimisation.**define_basic_model**(*assets*, *availability_constraint*, *asset_costs*)
Define the binary integer planning problem

Parameters

- **assets** (*list*) – The list of assets
- **availability_constraint** (*dict*) – A dictionary of binary constraints on whether you can build each asset in *assets*
- **asset_costs** (*dict*) – The investment cost of each asset
- **asset_value** (*dict*) – The value function approximation of each asset

Returns **model** – An abstract instance of the incomplete model with no objective function

Return type pyomo.environ.ConcreteModel

smif.optimisation.**feature_vfa_model**(*assets*, *availability_constraint*, *asset_costs*, *feature_coefficients*, *asset_features*)
Define the value function approximation

Here we assume that the value function approximation is a function of the features of the state, rather than individual assets, or enumeration of all possible states

Parameters

- **assets** (*list*) – The list of assets
- **availability_constraint** (*dict*) – A dictionary of binary constraints on whether you can build each asset in *assets*
- **asset_costs** (*dict*) – The investment cost of each asset
- **features** (*list*) – The set of features
- **feature_coefficients** (*dict*) – The regression coefficients for each feature
- **asset_features** (*dict*) – The mapping of features to assets

Returns **model** – A concrete instance of the model

Return type pyomo.environ.ConcreteModel

Notes

The use of features decomposes the optimisation problem into several sub-components. The first is to find the clusters of 'bits' in the state, which correctly predict minimal cost investments and operation of infrastructure assets.

These clusters can be used to define a feature, adding an entry to the *feature* set of features and a column to the *asset_features* matrix, where 1 indicates that the feature includes that asset.

Initially, feature selection could begin by regressing the results from a random sample of investment policies. This could highlight which patterns of investments seem to result in least cost systems. In addition, the attributes of the assets could provide a set of features which at least help categorise the assets (e.g. according to sector, size, location, and asset type).

The binary integer problem is posed as follows:

$\min \sum_i^I c_i x_i + x_i \sum_f^F (e_{if} b_f)$

where

- i is an element in the set of assets I
- f is an element in the set of features F
- c_i is the cost of asset i
- x_i is the decision to invest in asset i
- e_{if} is the mapping of feature f to asset i
- b_f is the basis coefficient of asset i and feature f

smif.optimisation.**formulate_model**(*asset_register*, *availability_constraint*, *feature_coefficients*, *asset_features*)

smif.optimisation.**linear_vfa_model**(*assets*, *availability_constraint*, *asset_costs*, *asset_value*)

Define the value function approximation

Here we assume a linear relationship (in the asset)

Parameters

- **assets** (*list*) – The list of assets
- **availability_constraint** (*dict*) – A dictionary of binary constraints on whether you can build each asset in *assets*
- **asset_costs** (*dict*) – The investment cost of each asset

- **asset_value** (*dict*) – The value function approximation of each asset

Returns **model** – A concrete instance of the model

Return type pyomo.environ.ConcreteModel

smif.optimisation.**solve_model**(*model*, *state=None*)

Solves the model using glpk

Passing in a *state* as a list of asset names initialises the state of the model, fixing those decision variables.

Parameters

- **model** (*pyomo.environ.ConcreteModel*) – A concrete instance of the model
- **state** (*dict, optional, default=None*) – A list of assets which were installed in a previous iteration or time period
- **returns** (*pyomo.core.Model.Instance*) – An instance of the model populated with results

smif.optimisation.**state_vfa_model**(*assets*, *availability_constraint*, *asset_costs*, *asset_value*, *states*)

Define the value function approximation

Here we assume that the value function approximation is a function of the state, rather than individual assets

Unfortunately, the number of states becomes very large, growing exponentially in the number of assets, and so representing the approximate value function like this is very inefficient as soon as the number of assets increases above 32 (about 4 GB).

Parameters

- **assets** (*list*) – The list of assets
- **availability_constraint** (*dict*) – A dictionary of binary constraints on whether you can build each asset in *assets*
- **asset_costs** (*dict*) – The investment cost of each asset
- **asset_value** (*dict*) – The value function approximation of each asset
- **states** (*dict*) – A dictionary where the keys are tuples of entries in *assets* and an index of states (`2 ** len(assets)`) and the value is a binary indicator showing the possible combinations

Returns **model** – A concrete instance of the model

Return type pyomo.environ.ConcreteModel

Notes

$\hat{v}_t^n = \min_{a_t \in A_t^n} C_t^{INV}(S_t^n, a_t^n) + V_t^n(S_t^n, a_t^n)$

smif.parameters module

Encapsulates the input or output parameters of a sector model, for example:

```
- name: petrol_price
  spatial_resolution: GB
  temporal_resolution: annual
- name: diesel_price
  spatial_resolution: GB
  temporal_resolution: annual
- name: LPG_price
  spatial_resolution: GB
  temporal_resolution: annual
```

class smif.parameters.**ModelParameters**(*parameters*)
Bases: object

A container for all the model inputs

Parameters **inputs** (*list*) – A list of dicts of model parameter name, spatial resolution and temporal resolution

get_spatial_res(*name*)
The spatial resolution for parameter *name*

Parameters **name** (*str*) – The name of a model parameter

get_temporal_res(*name*)
The temporal resolution for parameter *name*

Parameters **name** (*str*) – The name of a model parameter

names

parameters
A list of the model parameters

Returns

Return type smif.parameters.ParameterList

spatial_resolutions
A list of the spatial resolutions

Returns A list of the spatial resolutions associated with the model parameters

Return type list

temporal_resolutions
A list of the temporal resolutions

Returns A list of the temporal resolutions associated with the model parameters

Return type list

class smif.parameters.**Parameter**(*name*, *spatial_resolution*, *temporal_resolution*)
Bases: tuple

__getnewargs__()
Return self as a plain tuple. Used by copy and pickle.

static __new__(*_cls*, *name*, *spatial_resolution*, *temporal_resolution*)
Create new instance of Parameter(name, spatial_resolution, temporal_resolution)

__repr__()
Return a nicely formatted representation string

name
Alias for field number 0

spatial_resolution
Alias for field number 1

temporal_resolution
Alias for field number 2

smif.sector_model module

This module acts as a bridge to the sector models from the controller

The *SectorModel* exposes several key methods for running wrapped sector models. To add a sector model to an instance of the framework, first implement *SectorModel*.

Key Functions

This class performs several key functions which ease the integration of sector models into the system-of-systems framework.

The user must implement the various abstract functions throughout the class to provide an interface to the sector model, which can be called upon by the framework. From the model's perspective, *SectorModel* provides a bridge from the sector-specific problem representation to the general representation which allows reasoning across infrastructure systems.

The key functions include

- converting input/outputs to/from geographies/temporal resolutions
- converting control vectors from the decision layer of the framework, to asset Interventions specific to the sector model
- returning scaler/vector values to the framework to enable measurements of performance, particularly for the purposes of optimisation and rule-based approaches

class smif.sector_model.**SectorModel**
Bases: abc.ABC

A representation of the sector model with inputs and outputs

extract_obj(*results*)
Implement this method to return a scalar value objective function

This method should take the results from the output of the *simulate()* method, process the results, and return a scalar value which can be used as a component of the objective function by the decision layer

Parameters **results** (*dict*) – A nested dict of the results from the *simulate()* method

Returns A scalar component generated from the simulation model results

Return type float

initialise(*initial_conditions*)
Implement this method to set up the model system

Parameters **initial_conditions** (*list*) – A list of past Interventions, with build dates and locations as necessary to specify the infrastructure system to be modelled.

inputs
The inputs to the model

The inputs should be specified in a list. For example:

```
- name: eletricity_price
  spatial_resolution: GB
  temporal_resolution: annual
```

Parameters **value** (*list*) – A list of dicts of inputs to the model. These includes parameters, assets and exogenous data

Returns

Return type smif.parameters.ModelInputs

intervention_names
The names of the interventions

Returns A list of the names of the interventions

Return type list

name
The name of the sector model

Returns The name of the sector model

Return type str

outputs
The outputs from the model

Parameters **value** (*list*) – A list of dicts of outputs from the model. This may include results and metrics

Returns

Return type *smif.parameters.ModelParameters*

simulate(*decisions*, *state*, *data*)
Implement this method to run the model

Parameters

- **decisions** (*list*) – A list of :py:class:Intervention to apply to the modelled system
- **state** (*list*) – A list of :py:class:StateData to update the state of the modelled system
- **data** (*dict*) – A dictionary of the format: `data[parameter] = [SpaceTimeValue(region, interval, value, units), ...]`

Returns A dictionary of the format: `results[parameter] = [SpaceTimeValue(region, interval, value, units), ...]`

Return type dict

Notes

In the results returned from the *simulate()* method:

interval should reference an id from the interval set corresponding to the output parameter, as specified in model configuration

region should reference a region name from the region set corresponding to the output parameter, as specified in model configuration

validate()
Validate that this SectorModel has been set up with sufficient data to run

class smif.sector_model.**SectorModelBuilder**(*name*)
Bases: object

Build the components that make up a sectormodel from the configuration

Parameters **name** (*str*) – The name of the sector model

add_inputs(*input_dict*)
Add inputs to the sector model

add_interventions(*intervention_list*)
Add interventions to the sector model

Parameters **intervention_list** (*list*) – A list of dicts of interventions

add_outputs(*output_dict*)
Add outputs to the sector model

create_initial_system(*initial_conditions*)
Set up model with initial system

finish()
Validate and return the sector model

load_model(*model_path*, *classname*)
Dynamically load model module

validate()
Check and/or assert that the sector model is correctly set up - should raise errors if invalid

smif.sos_model module

This module coordinates the software components that make up the integration framework.

class smif.sos_model.**ModelSet**(*models*, *sos_model*)
Bases: object

Wraps a set of interdependent models

Given a directed graph of dependencies between models, any cyclic dependencies are contained within the strongly-connected components of the graph.

A ModelSet corresponds to the set of models within a single strongly- connected component. If this is a set of one model, it can simply be run deterministically. Otherwise, this class provides the machinery necessary to find a solution to each of the interdependent models.

The current implementation first estimates the outputs for each model in the set, guaranteeing that each model will then be able to run, then begins iterating, running every model in the set at each iteration, monitoring the model outputs over the iterations, and stopping at timeout, divergence or convergence.

Notes

This calls back into *SosModel* quite extensively for state, data, decisions, regions and intervals.

converged(*timestep*)
Check whether the results of a set of models have converged.

Returns **converged** – True if the results have converged to within a tolerance

Return type bool

Raises DivergenceError – If the results appear to be diverging

guess_results(*model*, *timestep*)
Dependency-free guess at a model's result set.

Initially, guess zeroes, or the previous timestep's results.

run(*timestep*)
Runs a set of one or more models

class smif.sos_model.**RunMode**
Bases: enum.Enum

Enumerates the operating modes of a SoS model

dynamic_optimisation = <RunMode.dynamic_optimisation: 3>

sequential_simulation = <RunMode.sequential_simulation: 1>

static_optimisation = <RunMode.static_optimisation: 2>

static_simulation = <RunMode.static_simulation: 0>

class smif.sos_model.**SosModel**
Bases: object

Consists of the collection of timesteps and sector models

This is NISMOD - i.e. the system of system model which brings all of the sector models together. Sector models may be joined through dependencies.

This class is populated at runtime by the *SosModelBuilder* and called from *smif.cli.run_model()*.

models
dict

This is a dictionary of `smif.SectorModel`

initial_conditions
list

List of interventions required to set up the initial system, with any state attributes provided here too

static add_data_series(*list_a*, *list_b*)
Given two lists of SpaceTimeValues of identical spatial and temporal resolution, return a single list with matching values added together.

Notes

Assumes a data series is not sparse, i.e. has a value for every region/interval combination

determine_running_mode()
Determines from the config in what mode to run the model

Returns The mode in which to run the model

Return type *RunMode*

get_data(*model*, *timestep*)
Gets the data in the required format to pass to the simulate method

Returns A nested dictionary of the format: `data[parameter][region][time_interval] = {value, units}`

Return type dict

Notes

Note that the timestep is *not* passed to the SectorModel in the nested data dictionary. The current timestep is available in `data['timestep']`.

get_decisions(*model*, *timestep*)
Gets the interventions that correspond to the decisions

Parameters

- **model** (*smif.sector_model.SectorModel*) – The instance of the sector model wrapper to run
- **timestep** (*int*) – The current model year

get_state(*model*, *timestep*)
Gets the state to pass to SectorModel.simulate

inputs
A dictionary of model names associated with an inputs

Returns Keys are parameter names, value is a list of sector model names

Return type dict

intervention_names
Names (id-like keys) of all known asset type

outputs
Model names associated with model outputs & scenarios

Returns Keys are parameter names, value is a list of sector model names

Return type dict

resolution_mapping
Returns the temporal and spatial mapping to an input, output or scenario parameter

Example

The data structure follows `source->parameter->{temporal, spatial}`:

```
{
    'scenario': {
        'raininess': {
            'temporal_resolution': 'annual',
            'spatial_resolution': 'LSOA'
        }
    }
}
```

results
Get nested dict of model results

Returns Nested dictionary in the format results[int:year][str:model][str:parameter] => list of SpaceTimeValues

Return type dict

run()
Runs the system-of-system model

0.Determine run mode

1.Determine running order

2.Run each sector model

3.Return success or failure

run_sector_model(*model_name*)
Runs the sector model

Parameters **model_name** (*str*) – The name of the model, corresponding to the folder name in the models subfolder of the project folder

run_sector_model_timestep(*model*, *timestep*)
Run the sector model for a specific timestep

Parameters

- **model** (*smif.sector_model.SectorModel*) – The instance of the sector model wrapper to run
- **timestep** (*int*) – The year for which to run the model

scenario_data
Get nested dict of scenario data

Returns Nested dictionary in the format `data[year][param] = SpaceTimeValue(region, interval, value, unit)`

Return type dict

sector_models
The list of sector model names

Returns A list of sector model names

Return type list

set_data (*model*, *timestep*, *results*)
Sets results output from model as data available to other/future models

Stores only latest estimated results (i.e. not holding on to iterations here while trying to solve interdependencies)

set_state (*model*, *from_timestep*, *state*)
Sets state output from model ready for next timestep

timestep_after (*timestep*)
Returns the timestep after a given timestep, or None

timestep_before (*timestep*)
Returns the timestep previous to a given timestep, or None

timesteps
Returns the list of timesteps

Returns A list of timesteps, distinct and sorted in ascending order

Return type list

class smif.sos_model.**SosModelBuilder**
Bases: object

Constructs a system-of-systems model

Builds a *SosModel*.

Examples

Call *SosModelBuilder.construct()* to populate a *SosModel* object and *SosModelBuilder.finish()* to return the validated and dependency-checked system-of-systems model.

```
>>> builder = SosModelBuilder()
>>> builder.construct(config_data)
>>> sos_model = builder.finish()
```

add_initial_conditions (*model_name*, *initial_conditions*)
Adds initial conditions (state) for a model

add_interventions (*model_name*, *interventions*)
Adds interventions for a model

add_model (*model*)
Adds a sector model to the system-of-systems model

Parameters **model** (*smif.sector_model.SectorModel*) – A sector model wrapper

add_model_data (*model*, *model_data*)
Adds sector model data to the system-of-systems model which is convenient to have available at the higher level.

add_planning (*planning*)
Loads the planning logic into the system of systems model

Pre-specified planning interventions are defined at the sector-model level, read in through the SectorModel class, but populate the intervention register in the controller.

Parameters **planning** (*list*) – A list of planning instructions

add_resolution_mapping (*resolution_mapping*)

Parameters **resolution_mapping** (*dict*) – A dictionary containing information on the spatial and temporal resoultion of scenario data

Example

The data structure follows `source->parameter->{temporal, spatial}`:

```
{'scenario': {
 'raininess': {'temporal_resolution': 'annual',
               'spatial_resolution': 'LSOA'}}}
```

add_scenario_data(*data*)
Load the scenario data into the system of systems model

Expect a dictionary, where each key maps a parameter name to a list of data, each observation with:

- timestep
- value
- units
- region (must use a region id from scenario regions)
- interval (must use an id from scenario time intervals)

Add a dictionary of list of *smif.SpaceTimeValue* named tuples, for ease of iteration:

```
data[year][param] = SpaceTimeValue(region, interval, value, units)
```

Default region: "national" Default interval: "annual"

add_timesteps(*timesteps*)
Set the timesteps of the system-of-systems model

Parameters **timesteps** (*list*) – A list of timesteps

construct(*config_data*)
Set up the whole SosModel

Parameters **config_data** (*dict*) – A valid system-of-systems model configuration dictionary

finish()
Returns a configured system-of-systems model ready for operation

Includes validation steps, e.g. to check dependencies

static intervention_state_from_data(*intervention_data*)
Unpack an intervention from the initial system to extract StateData

load_interval_sets(*interval_sets*)
Loads the time-interval sets into the system-of-system model

Parameters **interval_sets** (*list*) – A dict, where key is the name of the interval set, and the value the data

load_models(*model_data_list*)
Loads the sector models into the system-of-systems model

Parameters

- **model_data_list** (*list*) – A list of sector model config/data
- **assets** (*list*) – A list of assets to pass to the sector model

load_region_sets(*region_sets*)
Loads the region sets into the system-of-system model

Parameters **region_sets** (*list*) – A dict, where key is the name of the region set, and the value the data

set_max_iterations(*config_data*)

Module contents

smif

class smif.**SpaceTimeValue**(*region*, *interval*, *value*, *units*)
Bases: object

A piece of model input/output data

Parameters

- **region** (*str*) – A valid (unique) region name which is registered in the region register
- **interval** (*str*) – A valid (unique) interval name which is registered in the interval register
- **value** (*float*) – The value
- **units** (*str*) – The units associated with the *value*

class smif.**StateData**(*target*, *data*)
Bases: object

A piece of state data

Parameters

- **target** – The id or name of the object described by this state
- **data** – The state attribute/data to apply - could typically be a dict of attributes

CHAPTER 6

Indices and tables

- genindex
- modindex
- search

S

Symbols

A

B

C

D

N

O

P

R

S

T

V