resqpy

Release 0.0.0

BP

Nov 15, 2023

ABOUT

resqpy is a pure Python package which provides a programming interface (API) for reading, writing, and modifying reservoir models in the RESQML format. It gives you the ability to work with reservoir models programmatically, without having to know the details of the RESQML standard.

resqpy is written and maintained by bp, and is made available under the MIT license as a contribution to the open-source community.

The repository is hosted on GitHub.

CHAPTER

ONE

INSTALLATION

resqpy is written for Python 3.

It is recommended to use a conda environment for each new project.

```
$ conda create -n my_env python=3
$ conda activate my_env
```

Install using pip:

```
$ pip install resqpy
```

To install a development version on your local machine, use:

```
$ pip install -e /path/to/working/copy
```

To run unit tests (requires pytest):

```
$ python -m pytest tests/
```

To build the documentation locally (requires sphinx):

```
$ sphinx-build docs docs/html
```

CHAPTER

TWO

CONTRIBUTING TO RESQPY

Resqpy is an open source project released under the MIT license. Contributions of all forms are most welcome!

Resqpy was created by Andy Beer.

All contributors (alphabetical order):

- Andrew Ediriscoriya
- Andy Beer
- Casey Warshauer
- Chris Flynn
- Connor Tann
- Duncan Hunter
- Emma Nesbit
- Jeremy Tillay
- Kadija Hassanali
- Max Maunder
- Nathan Lane
- Nirjhor Chakraborty

2.1 Ways of contributing

- Submitting bug reports and feature requests (using the GitHub issue tracker)
- Contributing code (in the form of Pull requests)
- Documentation or test improvements
- Publicity and support

2.2 Checklist for pull requests

1. Changes or additions should have appropriate unit tests (see below)
2. Follow the PEP8 style guide as far as possible (with caveats below).
3. All public functions and classes should have Google-style docstrings
4. Code should be formatted with `yapf`
5. All GitHub checks should pass

2.3 Development environment setup

1. Clone the repo

   Create a fork of the repository using the GitHub website. Note: this step can be skipped if you already have write access to the main repo. Then, clone your fork locally to your computer to your working area:

   ```
   git clone <url from GitHub>
   cd resqpy
   ```

2. Set up a Python environment **Note: due to a requirement of one of the dependencies, you will need to use a 64-bit installation of Python when working with RESQPY.** The RESQPY project uses Poetry for dependency management and environment setup. Please install Poetry first if you have not already done so. With Poetry installed, please then install the Poetry Dynamic Versioning Plugin.

   Once both those packages are installed, the environment can then be setup automatically with all dependencies installed using the following command in the base directory (the directory with the pyproject.toml file):

   ```
   poetry install
   ```

   You can then switch to the virtual environment that you have just created using:

   ```
   poetry shell
   ```

   Whilst inside the virtual environment, you can run all of the unit tests to confirm that the setup was successful using the command:

   ```
   pytest
   ```

   If at a later date you wish to ensure that the dependencies in your dev environment are up to date with the latest supported versions, you can again run *poetry install* and your libraries will automatically be updated.

3. Make a Pull Request

   Create a new branch from master:

   ```
   git checkout master
   git pull
   git checkout -b <your-branch-name>
   ```

   You can then commit and push your changes as usual. Open a Pull Request on GitHub to submit your code to be merged into master.

2.4 Code Style

We use the yapf auto-formatter with the style configured in the repository. Most IDEs allow you to configure a formatter to run automatically when you save a file. Alternatively, you can run the following command before commiting any changes:

```
# Reformat all python files in the repository
yapf -ir .
```

Please try to write code according to the PEP8 Python style guide, which defines conventions such as variable naming and capitalisation. A consistent style makes it much easier for other developers to read and understand your code.

See *Static analysis* for how to check your code for conformance to PEP8 style.

2.5 Tests

2.5.1 Why write tests?

Automated tests are used to check that code does what it is supposed to do. This is absolutely key to maintaining quality: for example, automated tests enable maintainers to check whether anything breaks when new versions of 3rd party libraries are released.

As a rule of thumb: if you want your code to still work in 6 months' time, ensure it has some unit tests!

2.5.2 Writing tests

pytest is a framework for running automated tests in Python. It is a high-level framework, so very little code is required to write a test.

Tests are written in the form of functions with the prefix *test_*. Look in the tests directory for examples of existing tests. A typical pattern is "Arrange-Act-Assert":

```
def test_a_thing():
    """ Test to check that MyClass behaves as expected """

    # Arrange
    my_obj = resqml.MyClass()

    # Act
    result = my_obj.do_calculation()

    # Assert
    expected = [1,2,3]
    assert result == expected
```

2.5.3 Running tests

The easiest way to run the tests is simply to open a Pull Request on GitHub. This automatically triggers the unit tests, run in several different Python environments. Note that if your MR references an outside fork of the repo, then a maintainer may need to manually approve the CI suite to run.

Alternatively, you can run the tests against your local clone of the code base from the command line:

```
pytest
```

There are several command line options that can be appended, for example:

```
pytest -k foobar  # selects just tests with "foobar" in the name
pytest -rA        # prints summary of all executed tests at end
```

2.5.4 Static analysis

We use flake8 to scan for obvious code errors. This is automatically run part as part of the CI tests, and can also be run locally with:

```
flake8 .
```

The configuration of which error codes are checked by default is configured in the repo in setup.cfg.

By default in resqpy:

- `F`- Logical errors (i.e. bugs) are enabled
- `E`- Style checks (i.e. PEP8 compliance) are disabled

You can test for PEP8 compliance by running flake8 with further error codes:

```
flake8 . -select=F,E2,E3,E4,E7
```

2.6 Documentation

The docs are built automatically when code is merged into master, and are hosted at readthedocs.

There a few different versions of the documentation available, tied to different versions of the code:

URL	Version	Hidden
https://resqpy.readthedocs.io/en/latest/	The *master* branch, default	No
https://resqpy.readthedocs.io/en/stable/	The most recent git tag	No
https://resqpy.readthedocs.io/en/docs/	The *docs* branch	Yes

These automatically re-build when the relevant branch is updated, or when a new tag is pushed.

The documentation is also automatically built in a temporary staging area for all open Pull Requests. Check the "Checks" section of your Pull Request to see how the docs will look.

You may find it helpful to run a linter to check that the syntax of your ReStructured text is correct: the python package *restructuredtext-lint* is pretty good for this purpose. Similarly, many IDEs or plugins have a "rewrap" function that inserts line endings for uniform line lengths, which can make text more readable and visually pleasing.

You can also build the docs locally, providing you have installed all required dependencies as described above:

```
sphinx-build docs docs/html
```

The autoclasstoc extension is used to group some of the most commonly-used methods together at the top of the class summary tables. To make a method appear in this list, add *:meta common:* to the bottom of the method docstring.

2.7 Making a release

To make a release at a given commit, simply make a git tag:

```
# Make a tag
git tag -a v0.0.1 -m "Incremental release with some bugfixes"

# Push tag to github
git push origin v0.0.1
```

The tag must have the prefix `v` and have the form `MAJOR.MINOR.PATCH`.

Following [semantic versioning](https://semver.org/), increment the:

- `MAJOR` version when you make incompatible API changes,
- `MINOR` version when you add functionality in a backwards compatible manner, and
- `PATCH` version when you make backwards compatible bug fixes.

2.7.1 Interpreting version numbers

The version number is made available to users as an attribute of the module:

```
>>> import resqpy
>>> print(resqpy.__version__)
'1.6.1'
```

When working with a development version of the code that does not correspond to a tagged release, the version number will look a little different, for example `1.6.2.dev301+gddfbf6c`.

This can be interpreted as:

- `1.6.2` : is the *next* expected release. The previous release would be `1.6.1`.
- `dev301` : 301 commits added since the previous release.
- `+gddfbf6c` : a `+g` prefix followed by current commit ID: `ddfbf6c`.

2.7.2 How the version is retrieved

The git history defines the version, and consequently the version number cannot be written in a file that is itself under source control.

The Poetry plugin [poetry-dynamic-versioning](https://pypi.org/project/poetry-dynamic-versioning/) is used to extract the version number from the git history.

2.8 Get in touch

For bug reports and feature requests, please use the GitHub issue page.

For other queries about resqpy please feel free to get in touch at Nathan.Lane@bp.com

2.9 Code of Conduct

We abide by the Contributor-covenant standard:

https://www.contributor-covenant.org/version/1/4/code-of-conduct/code_of_conduct.md

CHAPTER

THREE

RELEASE NOTES

For details of changes in each release, see the releases page on GitHub.

Resqpy releases follow [semantic versioning](https://semver.org/).

CHAPTER

FOUR

TRADEMARKS

The following trademarks appear in resqpy source code and/or diagnostic messages.

- **RESQML** is a trademark of the Energistics Consortium
- **Nexus** is a registered trademark of the Halliburton Company
- **RMS** and **ROXAR** are registered trademarks of Roxar Software Solutions AS, an Emerson company
- **GOCAD** is also a trademark of Emerson

CHAPTER

FIVE

GETTING STARTED

```
>>> import resqpy
```

A first step is typically to instantiate a *resqpy.model.Model* object from your *.epc* file:

```
>>> from resqpy.model import Model
>>> model = Model(epc_file='my_file.epc')
<resqpy.model.Model at 0x7fdcd14e4700>
```

Models can be conveniently opened with the *resqpy.model.ModelContext* context manager, to ensure file handles are closed properly upon exit:

```
>>> from resqpy.model import ModelContext
>>> with ModelContext("my_model.epc") as model:
>>>     print(model.uuids())
```

If you don't have any RESQML datasets, you can use the tiny datasets included in the example_data directory of the resqpy repository.

To list all the parts (high level objects) in the model:

```
for part in model.parts():

   part_type = model.type_of_part(part)
   title = model.citation_title_for_part(part)
   uuid = str(model.uuid_for_part(part))

   print(f'{title:<30s} {part_type:<35s} {uuid}')
```

CHAPTER

SIX

TUTORIALS

These pages are currently under development. However, the material so far in place should be usable.

This page contains tutorials for programmers wanting to use the **resqpy** RESQML application programming interface (API). RESQML is the standard format for storing subsurface models. The resqpy API is a Python library providing classes and functions allowing application code to work with RESQML datasets at a high level.

Whilst reading through a tutorial, it is recommended that you have a Python interpreter (or Jupyter notebook) open in another window, so that code snippets can be cut, pasted and executed whilst progressing through the material.

6.1 Reservoir Modelling with RESQML

6.1.1 What is RESQML?

RESQML is a standard format for storing and exchanging reservoir models. It is 'software vendor neutral' meaning that the standard is not defined or owned by a single commercial company. Instead, RESQML is a public domain standard defined by Energistics, which is a consortium of oil companies and reservoir software companies.

The RESQML standard aims to be:

- Comprehensive: covering all the main elements of subsurface models for the entire modelling workflow
- Flexible: parts of a model can themselves constitute a valid package of data
- Efficient: array data is stored in binary form
- Rigorous: units and coordinate reference systems are thorough, and unique identifiers ensure correct identification of parts

RESQML is *the* standard for subsurface modelling, gradually being adopted across the industry.

Energistics also defines two other, related, standards: PRODML, which covers production data and WITSML, which handles well data.

Current Version

The current recommended version (as of March 2021) of the RESQML standard is 2.0.1. Version 2 marked a major development from earlier versions.

6.1.2 The Structure of a RESQML Model

Physically, a resqml model is stored in a compressed file with the extension .epc together (usually) with one or more hdf5 format files holding array data, with the extension .h5. Neither of these file types can be viewed or edited using a simple text editor – other more specialised tools must be used.

The *epc* File

The *epc* (Energistics Package Convention) file is the 'main' file holding the metadata for the model, along with scalar data. It also contains links to any hdf5 (.h5) files holding array data for the model. The *epc* file itself conforms to another standard file structure (not specific to Energistics) known as opc. It is basically a zipped file containing a set of xml files. An individual xml file might be one of the following (amongst others):

- The main contents file holding a list of the other xml files.
- A primary model part, holding data for an object such as a coordinate reference system (crs), a surface, a grid, a grid property etc.
- A relationship file, holding the relationships between a model part and other parts, for example which crs applies to the geometry of a grid object. (These relationship files are grouped together in a subdirectory named `_rels` within the zipped *epc* file structure.)
- A reference to an *hdf5* file holding array data for one or more of the model parts.
- A documentation folder holding non-data documents that are not defined by the standard.

The RESQML standard does not specify a minimum set of objects, or parts, for a model to be valid. So, for example, a coordinate reference system on its own would be a valid (if minimal) RESQML model (or package, to use the Energistics term). For maximum flexibility, there is not even a requirement that all the objects referred to in the relationships must be present. So, for example, if a grid object refers to a crs, then that crs may be absent. This allows a partial model to be transferred, for example, when only a small part of a model has changed.

Because array data is not stored in the epc file, file sizes are small – perhaps a few tens of kilobytes for a sizeable model.

The bulk of the RESQML standard is published in the form of xml schema definition files (with extension xsd). These files specify the required xml contents for each of the types of object.

hdf5 (.h5) Files

Any array data that forms part of the model must be stored in an hdf5 file, and not within one of the xml files within the epc (though those xml files will contain references to the hdf5 files). The hdf5 format – hierarchical data format – is not specific to Energistics. It is widely used in high performance technical computing. It stores array data in binary format and can handle extremely large data sets. A single hdf5 file can hold multiple arrays, organised within the file in a hierarchical structure rather like a directory structure. Random access to an array, or part of an array, within an hdf5 file is fast. The detailed format of the array storage is highly compatible with Python numpy arrays.

All the arrays for a RESQML model may be stored in a single hdf5 file, or they may be spread amongst multiple files. Furthermore, an hdf5 file may be referred to by more than one *epc* file, potentially reducing the duplication of data. A particular *epc* file does not need to refer to all the arrays within an hdf5 file. Despite all this flexibility, the recommendation is to keep a simple one-to-one correspondence between *epc* and hdf5 files wherever possible (and this is the default behaviour of the resqpy code base).

The hdf5 files can be large: typically several gigabytes.

6.1.3 RESQML Objects and Universally Unique Identifiers

From the discussion above, it can be seen that a RESQML model is a collection of parts, most of which are RESQML objects. The relationships between these objects also forms part of the model. The RESQML standard defines many classes of object, such as a fault interpretation or an IJK grid. There is no limit on how many objects of any given type are included in a single RESQML model. For example, several different coordinate reference systems could be included in one model.

To help keep track of these objects in a rigorous way, whenever a new object is created, or modified, it is given a new universally unique identifier (UUID), also known as a globally unique identifier (GUID). The format of the UUIDs is not specific to Energistics – it is the subject of an ISO standard (ISO/IEC 9824-8). As the name suggests, a UUID is completely unique. If you see 2 UUIDs that are the same, they are referring to the same thing. It is this feature that allows partial RESQML models to be moved around and joined up again correctly.

A UUID is actually a 128 bit integer. However, it is usually displayed in hexadecimal form with hyphens at key points, for example: decd627f-c91e-47e1-946e-8a6a4d91617f

All the links between objects within a RESQML model use the UUIDs as the way to ensure that the correct objects are identified. They also make it possible to create a rigorous audit trail of the development of a model, though such an audit trail is not currently covered by the RESQML standard.

6.1.4 Units of Measure

The Energistics standards include a rigorous handling of units of measure (uom). This aspect of the standards is common to RESQML, PRODML and WITSML. The uom definitions include things such as:

- A (very long) list of physical units in use around the world.
- Which units are convertible to each other, together with conversion factors.
- A (long) list of quantity classes, such as rock permeability.
- Which units are applicable to which quantity classes.
- The fundamental dimensions of a quantity class (in terms of Mass, Length, Time etc.)

Although the Energistics UoM definitions are primarily intended to support the Energistics standards, they are general purpose and could form the basis of any technical unit handling and conversion system.

6.2 Getting started with the Model class

This tutorial covers opening an existing RESQML dataset and identifying the high level objects contained within it.

6.2.1 Prerequisites

You will need to have resqpy installed in your Python environment, along with its dependencies, before proceeding.

You will also need an example RESQML dataset (some are available within the resqpy repository). The RESQML dataset will consist of two files, one with extension .epc and the other .h5. This pair of files should have the same name prior to the extension and be located in the same directory (folder). You can use any dataset for this exercise – the detailed output from each step will vary depending on the data.

Note: Example file names shown here and in other resqpy tutorials are for a Unix environment. If you are working in a Windows environment, the file paths would be in the corresponding format.

6.2.2 Importing the model module

In this tutorial, we will be using the *resqpy.model.Model* class which is contained in *resqpy.model*. This can be imported with:

```
import resqpy.model as rq
```

The rest of this tutorial assumes the import statement shown above. However, you can vary it according to your preferred style. Other examples are:

```
import resqpy.model
from resqpy.model import Model
```

6.2.3 Opening an existing RESQML dataset

The dataset is accessed via the epc file. It is opened with:

```
model = rq.Model('/path/to/my_file.epc')
```

Tip: the `Model` initialiser method has some optional arguments which are needed when creating a new dataset or copying an existing dataset before opening the duplicate.

As a convenient shorthand, models can be opened using the *resqpy.model.ModelContext* context manager:

```
with rq.ModelContext("my_model.epc", mode="read/write") as model:
    print(model.uuids())
```

When a RESQML dataset is opened in this way, file handles are safely closed when the "with" clause exits and optionally changes can be written to disk.

A `Model` object is almost always the starting point for code using resqpy. The other resqpy object classes require a Model object which is treated as a 'parent'. The resqpy Model class is not equivalent to any of the RESQML classes, rather it should be thought of as equivalent to a whole epc file.

The `Model` class includes many methods. In this tutorial we will focus on some of the more essential ones when reading a model.

6.2.4 Keys to the RESQML high level objects

A RESQML dataset is a collection of high level objects, also called parts. There are four primary data items that code is likely to work with when handling these parts:

- A *uuid* (universally unique identifier), which is an object of class uuid.UUID. The uuid module is a standard Python module. A uuid is sometimes referred to as a guid (globally unique identifier). The resqpy code base sticks with the term uuid as preferred by Energistics and the underlying ISO standard which these identifiers adhere to. As the uuids are often presented as a hexadecimal string, the resqpy code generally allows uuids to be passed around either as UUID objects or as strings.
- A *part name*, which is a string representing an internal 'file name' within the epc package. A part name usually consists of a high level object class followed by a uuid (see next point) in hexadecimal form and a .xml extension. Where a resqpy argument is named part or part_name, it refers to such a part name.
- An *xml root node*. The metadata for each part is held within the epc in xml format. The lowest level of resqpy code reads this xml into an internal tree structure using the lxml standard Python module, which is compatible with elementTree. Where a resqpy argument name contains root or root_node, it is referring to the root node in

the internal tree representation of the xml for the part. Such a root is an object of type lxml._Element and does not have a meaningful human readable form.

- A *citation title*, which is a human readable string held within the citation block of the xml for the part. This is what a human would consider to be the name of the high level object. However, there is no requirement for the citation titles to be unique within a RESQML dataset, so they should generally not be used as a primary key. Where a resqpy argument is named citation_title, or simply title, it is referring to this item of data.

Within a `Model` object, there is a one-to-one correspondence between a part name and a uuid, and between a part name and a root node. There are methods for moving from one of these to another and also for finding the (possibly non-unique) citation title.

The `Model` class contains four similar methods each of which returns a list of items, corresponding to the four points above. The methods have the names:

- *resqpy.model.Model.uuids()*
- *resqpy.model.Model.parts()*
- *resqpy.model.Model.roots()*
- *resqpy.model.Model.titles()*

If applied to a `Model` object without any arguments, a full list is returned, i.e. with one item per high level object.

6.2.5 Selectively listing high level objects

The four methods mentioned above have similar lists of optional arguments, some of which allow for filtering of the list:

- **`obj_type` (string): only objects of this RESQML high level object class are included in the returned list. The leading obj underscore may be omitted from the class name. Examples:**

```
model.parts(obj_type = "obj_LocalDepth3dCrs")
model.titles(obj_type = "DeviationSurveyRepresentation")
```

- **`uuid` (UUID object or string): the list will contain the one high level object which matches this uuid, eg.:**

```
model.roots(uuid = '27e11404-231b-11ea-8971-80e650222718')
```

- **`related_uuid` (UUID object or string): the list will only contain those high level objects which have a relationship with the object identified by this uuid, e.g.:**

```
model.parts(related_uuid = '27e11404-231b-11ea-8971-80e650222718')
```

- **`extra` (dictionary of key:value pairs): if a non-empty dictionary is provided, only those high level objects with extra metadata including all the key:value pairs in this dictionary will be in the returned list, eg.:**

```
model.roots(obj_type = 'WellboreTrajectoryRepresentation',
            extra = {'development_phase': 2, 'planned_use': 'injection'})
```

- **`title` (string): the list will only contain high level objects whose citation title matches this string, e.g.:**

```
model.uuids(title = 'WELL_A')
```

By default, the `title` argument results in a case insensitive string comparison with the objects' citation titles. However, other optional arguments may be used to modify this behaviour:

- `title_case_sensitive` (boolean, default `False`): if set `True`, the comparison will be case sensitive

- `title_mode` (string, default 'is'): one of 'is', 'starts', 'ends', 'contains', 'is not', 'does not start', 'does not end', 'does not contain'

If multiple filtering arguments are supplied, then only those high level objects meeting all the criteria will be included ('and' logic).

Rather than starting from the full list of high level objects present in the model, it is also possible to pass in a starting list to apply other filters to:

- **`parts_list` (list of strings): if present, a list of 'input' part names to which any other filtering arguments are applied, eg:**
  - roots(parts_list = ["obj_IjkGridRepresentation_27e10fc2-231b-11ea-8971-80e650222718.xml", "obj_IjkGridRepresentation_319154f4-5f3e-11eb-9d8d-80e650222718.xml"], title = 'ROOT')

The return list will not be in any particular order unless a further argument is supplied:

- `sort_by` (string): if not None then one of 'newest', 'oldest', 'title', 'uuid', 'type'

6.2.6 Finding a single high level object

Each of the above four methods has a corresponding method which can be used if it is expected that at most one high level object will meet the criteria:

- *resqpy.model.Model.uuid()*
- *resqpy.model.Model.part()*
- *resqpy.model.Model.root()*
- *resqpy.model.Model.title()*

For example:

- `model.title(uuid = '27e11404-231b-11ea-8971-80e650222718')`

The filtering arguments for these singleton methods are the same as for the list methods. If no objects match the criteria then None is returned. There is a further argument which controls the behaviour when more than one object matches the criteria:

- multiple_handling (string, default 'exception'): one of 'exception', 'none', 'first', 'oldest', 'newest'

6.2.7 Other methods in the Model class

Although the Model class contains many other methods, the eight listed above are the crucial ones when reading a RESQML dataset. Most of the other methods are involved with writing or modifying datasets, which are more complicated operations and will be covered by other tutorials.

There are three other methods worth mentioning in passing here, which are involved with accessing the hdf5 file:

- *resqpy.model.Model.h5_file_name()*
- *resqpy.model.Model.h5_uuid()*
- *resqpy.model.Model.h5_release()*

The first of these, `h5_file_name()`, returns the full path of the hdf5 file for the model. By default, any hdf5 filename(s) stored within the xml in the epc file are ignored and a path for a single hdf5 file is returned, based on the epc filename supplied when initialising the model. This protocol makes it much easier to move RESQML datasets around and rename them but it assumes a simple one-to-one pairing of epc and h5 files. Optional arguments allow for other ways of working.

The `h5_uuid()` method returns the uuid for the hdf5 'external part'. Although not a normal RESQML high level object, the hdf5 file(s) associated with the epc are treated as special parts and each gets its own uuid. Calling code does not usually need to be concerned with this if the simple file naming protocol is being used.

The last of the three methods, `h5_release()`, ensures that the hdf5 file is closed, assuming that it has been accessed by other resqpy operations. This is more important when writing a dataset, to ensure the hdf5 file is released ready for other code to access.

The model.py module also contains a tiny convenience function for creating a new, empty RESQML dataset (overwriting any existing files with the same name):

- `resqpy.model.new_model('new_file.epc')()`

6.2.8 Summary

In this tutorial, we have seen how to open an existing RESQML dataset and discover what high level objects it contains.

6.3 Coordinate Reference Systems

In this resqpy tutorial, we will take a look at a RESQML coordinate reference system object.

6.3.1 Opening a model

We can open a model in the usual way, as shown in previous tutorials:

```
import resqpy.model as rq
model = rq.Model('s_bend.epc')
```

6.3.2 About RESQML coordinate reference systems

The RESQML standard requires all objects that involve a geometry (in 3D space) to have a related coordinate reference system (CRS). There are actually two classes of CRS:

- `obj_LocalTime3dCrs` which has time based z values, for seismic data
- `obj_LocalDepth3dCrs` which has length based z values, for everything else

(In these tutorials RESQML object classes will sometimes be shown for brevity without the leading `obj_`. The resqpy code also usually accepts these class names with or without the `obj_`)

Both these classes of CRS consist only of xml metadata – there is no associated array data, so no group in the hdf5 file. The metadata includes the units of measure (uom) for the x & y values, and an independent uom for the z values. It also indicates whether the z values are increasing upwards (away from the centre of the Earth), or downwards, and how the x & y axes relate to the compass directions.

The local coordinate reference system may also be placed within a parent CRS, with an xyz origin which locates the local point (0.0, 0.0, 0.0) within another frame of reference. A rotation in the projected (plan) view may also be specified. The parent CRS may optionally be identified as another RESQML CRS or by specifying an EPSG code. (For more information on EPSG codes visit https://epsg.org) The parent may also be left unspecified, in which case the implication is that all CRS objects within the RESQML dataset share the same parent frame of reference.

The rest of this tutorial will focus on a depth based CRS (LocalDepth3dCrs).

6.3.3 Identifying a CRS object

Usually when reading a CRS, it has been referenced in some other object such as a grid, a surface or a well trajectory. The reference contains the universally unique identifier (uuid) of the CRS and the uuid can be thought of as a primary key for the object. A later tutorial will look more at object references.

Alternatively, if we are not following a reference, we can list the uuids of depth based CRS objects with the `uuids()` method of the Model class, which we encountered in an earlier tutorial:

```
crs_uuid_list = model.uuids(obj_type = 'LocalDepth3dCrs')
```

The S-bend example dataset only has one CRS object, so this list should only contain one uuid. If the calling code knows that will be the case, it can instead use the singular method:

```
crs_uuid = model.uuid(obj_type = 'LocalDepth3dCrs')
```

or, of course, we could pick a single item out of the list, for example with:

```
crs_uuid = crs_uuid_list[0]
```

6.3.4 Instantiating a resqpy Crs object

Many of the RESQML object classes have corresponding resqpy Python classes available, and that includes the CRS classes. Note that there is not always a one-to-one correspondence between RESQML and resqpy classes though. (The next tutorial discusses this in more detail.) The resqpy Crs class caters for both the RESQML CRS classes: `LocalTime3dCrs` and `LocalDepth3dCrs`

Having found the uuid, we can instantiate a resqpy Crs object:

```
import resqpy.crs as rqc
crs = rqc.Crs(model, uuid = crs_uuid)
```

A similar approach is used to instantiate objects for all the resqpy classes, when reading an existing dataset.

Older releases of resqpy used the xml root instead of the uuid, when instantiating resqpy objects for existing RESQML objects. This is now deprecated (from v0.3.0).

6.3.5 Inspecting the resqpy Crs object

The resqpy API allows calling code to make direct use of attributes within high level objects. Three commonly accessed attributes in a Crs object are:

```
crs.xy_units
crs.z_units
crs.z_inc_down
```

Note that these attribute names are not generally identical to the RESQML schema definition field names. In this case, for example, resqpy uses `xy_units` where the RESQML xsd uses `ProjectedUom`

6.3.6 Using resqpy Crs methods

Of course the resqpy classes provide methods for working with the objects. An example from the Crs class is a method which checks whether one Crs is equivalent to another. The following should always return `True` !:

```
crs.is_equivalent(other_crs = crs)
```

Another Crs method determines the handedness of the xyz axes:

```
crs.is_right_handed_xyz()
```

The S-bend dataset only has one CRS. If it had more, the following Crs methods could be used to convert xyz data from one to another:

```
crs.convert_to(another_crs, xyz)  # returns a new tuple for a single xyz point
crs.convert_array_to(another_crs, xyz_array)  # converts in situ a numpy float array of␣
↪shape (..., 3)
```

The two conversion methods above assume that the xyz data is starting in the space of this `crs` and being converted to `another_crs`. There are an equivalent pair of methods for converting from the other crs (ie. the one passed as an argument), so the following two lines would have exactly the same effect as the two above:

```
another_crs.convert_from(crs, xyz)
another_crs.convert_array_from(crs, xyz_array)
```

Along with some other simple resqpy classes, Crs includes a definition for __eq__() and __ne__(), so that the == and != operators can be used to test for equivalence between two coordinate reference system objects (behind the scenes this is calling the *is_equivalent()* method):

```
if crs == another_crs:
    print('no coordinate transformation needed')
```

The Crs class includes other methods but those mentioned above are the most commonly used ones.

6.3.7 RESQML Units of Measure (uom)

The RESQML standard includes a comprehensive set of data for handling physical units, which is shared with the sister standards PRODML and WITSML. Some components of this data include:

- a comprehensive list of quantity classes, such as volume flow rate
- the physical dimensionality of each quantity class (in terms of Mass, Length, Time etc.), e.g. L3/T
- a reference unit of measure for each quantity class (called the base unit), e.g. m3/s
- a comprehensive list of units of measure
- unit prefixes, e.g. *nano*
- conversion factors for compatible units of measure to and from the base unit, and for the prefixes

There is also a list of standard *property kinds* of relevance to reservoir modelling, such as *porosity*.

The resqpy library does not yet make full use of the RESQML units data. So, for example, the Crs conversion methods currently only recognize the following length units: m, ft, ft[US]. However, a release coming soon will include support for the full RESQML uom system.

6.4 High Level Objects

This tutorial discusses some concepts that are important when working with high level objects in resqpy.

6.4.1 RESQML and resqpy classes

The RESQML standard defines many classes of high level objects and specifies precisely how they are to be represented in persistent storage (files). However, application code making use of resqpy will not usually interact directly with the RESQML objects but rather with the closely related resqpy classes of object. Whilst there is a degree of correspondence between RESQML high level classes and resqpy classes, there are some differences which should be borne in mind:

- Class names are usually different
- Some resqpy classes cater for more than one RESQML class
- There are a few circumstances where a RESQML object can be represented by more than one resqpy class
- RESQML is purely concerned with what data is stored for a class, whilst a resqpy class also contains methods to provide different ways of viewing or processing the data
- Whereas a RESQML class is defined in a hierarchical way, and makes use of inheritance (xsd extension base) and abstract classes, the comparable resqpy class is flattened with data elements held as simple attributes
- Some resqpy classes use class inheritance to allow common functionality to be implemented in a base class – this is a different hierarchy to that used in the RESQML schema definition
- Not all RESQML classes are yet catered for (except in the lowest level generic layer of code)
- Some RESQML objects have optional attributes or multiple possible representations of an attribute – some of the options might not yet be implemented in resqpy

Apart from the last two of these points, the differences are due to the different aims of RESQML and resqpy: RESQML aims to provide a comprehensive and unambiguous standard for efficient storage of reservoir models, whereas resqpy aims to provide high level functionality to facilitate processing of the models.

The table at the end of this page shows which resqpy class implements each RESQML class.

6.4.2 Reading and writing objects

From the discussion above, it is evident that the same information can exist in two different representations: in a file in RESQML format, or in memory as resqpy objects. When reading a dataset, the transformation is from RESQML to resqpy. When writing, the transformation is from resqpy to RESQML. However, for efficiency of processing, things are more complicated than that and the representation of a conceptual object can exist in one of a number of states.

Firstly, the resqpy code differentiates between RESQML classes depending on how much array data they involve:

- Classes with no array data, for example measured depth datum (*obj_MdDatum* in RESQML, *MdDatum* in resqpy)
- Classes with modest amounts of array data, eg. wellbore trajectory (*obj_WellboreTrajectoryRepresentation* in RESQML, *Trajectory* in resqpy)
- Classes with large amounts of array data, eg. ijk cellular grid (*obj_IjkGridRepresentation* in RESQML, *Grid* in resqpy)

The rest of this tutorial will refer to these volumes of array data as none, small or large respectively. Note that the behaviour of the resqpy code is based on the typical amounts of array data for a given class, not the actual size of the arrays for a specific object.

When **reading**, the representation of an object passes through these states:

1. Only in files: metadata in xml compressed into the epc file; any array data in the hdf5 file
2. Metadata loaded into equivalent data structure in memory; any array data still only in the hdf5 file
3. In memory resqpy object instantiated; metadata in object attributes; if small array(s), array data also in memory as attributes
4. For classes with large arrays, Individual arrays are cached as attributes on demand

Step 2 in this sequence occurs with the instantiation of a Model object for an existing epc. The metadata for each part is loaded into a Python lxml tree (which is compatible with elementTree). Application code does not usually interact directly with this representation, though the root node of the lxml tree for an object is sometimes used as an argument to resqpy function calls. Here is an example of code that moves all objects in the s_bend dataset into state 2:

```
import resqpy.model as rq
model = rq.Model('s_bend.epc')
```

Step 3 occurs when the application code instantiates a resqpy object for one of the parts in the model. At this point, the lxml metadata is interrogated to set the values of the class-specific attributes. The naming and definition of these attributes is often very similar to the equivalent metadata fields in the RESQML class. If the class has a small volume of array data, then it is also loaded at this point into numpy array attributes. The resqpy class might also have derived attributes which are not stored in the RESQML object but are set for the convenience of application code. The following lines will create a resqpy Grid object in state 3 for one of the IjkGridRepresentation parts in the s_bend model:

```
import resqpy.grid as grr
faulted_grid_uuid = model.uuid(obj_type = 'IjkGridRepresentation', title = 'FAULTED GRID
↪')
faulted_grid = grr.Grid(model, uuid = faulted_grid_uuid)
```

Step 4 only pertains to classes with large amounts of array data. To minimize memory and time usage, these arrays are not loaded until application code requests them using specific methods in the class. The names of these methods usually contain terms like *cached* and/or *array_ref*. There is often another method allowing for the uncaching of such arrays, which has the effect of deleting the associated attribute from the resqpy object. The following example loads a numpy boolean array from the hdf5 file (unless it has already been cached), indicating which cells in a resqpy Grid object have geometry defined; the array is stored as an attribute of the object (cached) and also returned by the method:

```
faulted_grid.cell_geometry_is_defined_ref()
```

The Grid class also has a method which ensures that all arrays are cached:

```
faulted_grid.cache_all_geometry_arrays()
```

Note that these steps are triggered by application code calling resqpy methods. Apart from step 4, the calling code needs to keep track of which state the information for a particular object is in – resqpy itself is not generally keeping a handle on high level objects as they are instantiated.

When **writing**, the representation of an object typically passes through these states:

1. Only in memory, as a resqpy object, with metadata and any array data held as attributes
2. Metadata and any array data held as attributes of resqpy object; any array data also written to the hdf5 file
3. The metadata is also stored in an lxml tree, in memory, in a form ready to be written to the epc file
4. When all parts have been through the steps above, the metadata for all parts is written to the epc file from the lxml trees

Step 1 in this sequence is achieved by calling the initialization method of the resqpy class with arguments set to indicate import from a different format. Or an empty resqpy object can be instantiated and all the attributes set by the calling code. Only when the object's attributes are fully populated can the representation proceed with the rest of the steps.

The s_bend dataset, unrealistically, uses a single measured depth datum for 4 wells. Here is some example code for creating a new resqpy MdDatum object in state 1, located 5 metres to the east of the existing datum:

```
import resqpy.well as rqw
existing_md_uuid = model.uuid(obj_type = 'obj_MdDatum')  # we happen to know there is␣
↪only one MdDatum object
existing_md_datum = rqw.MdDatum(model, uuid = existing_md_uuid)
x, y, z = existing_md_datum.location
x += 5.0
new_md_datum = rqw.MdDatum(model,
                           crs_root = existing_md_datrum.crs_root,
                           location = (x, y, z))
```

Step 2 is achieved by the application code calling a method, usually named `write_hdf5()`, for the resqpy object. As the obj_MdDatum class does not involve any array data, this step does not apply to our example.

Step 3 Each resqpy class has a method named `create_xml()` which generates the lxml tree representation of the metadata, in memory, and adds the part to the parent resqpy Model object, also creating relationship data structures. Here is the line for the newly created MdDatum object instantiated above:

```
new_md_datum.create_xml()
```

Step 4 is achieved by the application code calling the `store_epc()` method of the Model object when all objects have been prepared as far as step 3. So in the example above, when the application code has generated all the required objects, the call is simply:

```
model.store_epc()
```

At this point the data is stored persistently in the epc file (and hdf5 file) and the application can exit, or delete the model and other objects.

6.4.3 Temporary object states

The two situations discussed above – reading and writing – are the most common ways of working with resqpy objects. However, resqpy has been designed to support processing of models and for this a third situation can arise: the need for temporary objects. Such objects are not written to the epc file (nor their arrays to the hdf5 file) but exist only in memory as resqpy objects.

As an example of working with temporary objects, imagine an application that generates many undrilled well trajectories and then tests them against a reservoir model to select the best trajectory. The trajectories could all be saved, using the sequence for writing resqpy objects outlined above. However, perhaps there is only the need to keep the trajectory that has been selected as best. The other trajectories would be temporary.

The simplest way to work with a temporary object is simply to instantiate it. This is equivalent to step 1 of the writing sequence above. Such an object can be used for most processing purposes. Note, however, that it has not been added as a part to the nominal parent Model object, nor does any xml exist for it. Some of the resqpy method and function calls require these other steps to have been taken.

Another approach for working with temporary objects is to create a separate, temporary, Model object and to instantiate the temporary high level objects with the temporary model as the parent. The `create_xml()` methods of the high level objects can be called without calling the `write_hdf5()` methods. If the temporary model's `store_epc()` method is not called, nothing will be written to the persistent file system. This is equivalent to steps 1 and 3 of the writing sequence discussed above.

6.4.4 Managing resqpy objects

Although a resqpy high level object is associated with a Model object (and contains a reference to the Model as an argument), the Model does not maintain a list of resqpy objects which have been instantiated for it. The Model does contain the list of RESQML parts, each of which can be used to instantiate a resqpy object (at least for the classes catered for in resqpy).

The exception is the resqpy Grid class (RESQML obj_IjkGridRepresentation), for which the Model class includes methods for optionally managing a list of resqpy Grid objects. This exception is made because grids can be memory and time intensive to instantiate, and are fundamental to all processing when working with a cellular model.

In general, though, it is up to the application code to manage the lifecycle of the resqpy objects.

6.4.5 RESQML to resqpy class mapping

The table below shows which high level resqpy class is used to represent each RESQML class. The blank rows indicate that a high level resqpy class has not yet been implemented for the RESQML class. (The lowest level resqpy code is generic, so steps 1 & 2 of the reading sequence above will function for all RESQML classes, as will step 4 of the writing sequence.)

RESQML class	array data	primary resqpy class
obj_Activity		
obj_ActivityTemplate		
obj_BlockedWellboreRepresentation	small	`resqpy.well.BlockedWell`
obj_BoundaryFeature	none	`resqpy.organize.BoundaryFeature`
obj_BoundaryFeatureInterpretation	none	`resqpy.organize.BoundaryFeatureInterpretati`
obj_CategoricalProperty	large	*resqpy.property.PropertyCollection*
obj_CategoricalPropertySeries		
obj_CommentProperty		
obj_CommentPropertySeries		
obj_ContinuousProperty	large	*resqpy.property.PropertyCollection*
obj_ContinuousPropertySeries		
obj_DeviationSurveyRepresentation	small	`resqpy.well.DeviationSurvey`
obj_DiscreteProperty	large	*resqpy.property.PropertyCollection*
obj_DiscretePropertySeries		
obj_DoubleTableLookup		
obj_EarthModelInterpretation	none	`resqpy.organize.EarthModelInterpretation`
obj_EpcExternalPartReference		
obj_FaultInterpretation	none	`resqpy.organize.FaultInterpretation`
obj_FluidBoundaryFeature	none	`resqpy.organize.FluidBoundaryFeature`
obj_FrontierFeature	none	`resqpy.organize.FrontierFeature`
obj_GenericFeatureInterpretation		
obj_GeneticBoundaryFeature	none	`resqpy.organize.GeneticBoundaryFeature`
obj_GeobodyBoundaryInterpretation	none	`resqpy.organize.GeobodyBoundaryInterpretati`
obj_GeobodyFeature	none	`resqpy.organize.GeobodyFeature`
obj_GeobodyInterpretation	none	`resqpy.organize.GeobodyInterpretation`
obj_GeologicUnitFeature	none	`resqpy.organize.GeologicUnitFeature`
obj_GeologicUnitInterpretation	none	`resqpy.strata.GeologicUnitInterpretation`
obj_GlobalChronostratigraphicColumn		
obj_GpGridRepresentation		
obj_Grid2dRepresentation	large	`resqpy.surface.Mesh`
obj_Grid2dSetRepresentation		

continues on next pa

Table 1 – continued from previous page

RESQML class	array data	primary resqpy class
obj_GridConnectionSetRepresentation	large	resqpy.fault.GridConnectionSet
obj_HorizonInterpretation	none	resqpy.organize.HorizonInterpretation
obj_IjkGridRepresentation	large	resqpy.grid.Grid
obj_LocalDepth3dCrs	none	resqpy.crs.Crs
obj_LocalGridSet		
obj_LocalTime3dCrs	none	resqpy.crs.Crs
obj_MdDatum	none	resqpy.well.MdDatum
obj_NonSealedSurfaceFrameworkRepresentation		
obj_OrganizationFeature	none	resqpy.organize.OrganizationFeature
obj_PlaneSetRepresentation		
obj_PointSetRepresentation	large	resqpy.surface.PointSet
obj_PointsProperty	large	*resqpy.property.PropertyCollection*
obj_PolylineRepresentation	small	resqpy.lines.Polyline
obj_PolylineSetRepresentation	small	resqpy.lines.PolylineSet
obj_PropertyKind	none	resqpy.property.PropertyKind
obj_PropertySet	none	*resqpy.property.PropertyCollection*
obj_RedefinedGeometryRepresentation		
obj_RepresentationIdentitySet		
obj_RepresentationSetRepresentation		
obj_RockFluidOrganizationInterpretation		
obj_RockFluidUnitFeature	none	resqpy.organize.RockFluidUnitFeature
obj_RockFluidUnitInterpretation		
obj_SealedSurfaceFrameworkRepresentation		
obj_SealedVolumeFrameworkRepresentation		
obj_SeismicLatticeFeature		
obj_SeismicLineFeature		
obj_SeismicLineSetFeature		
obj_StratigraphicColumn	none	resqpy.strata.StratigraphicColumn
obj_StratigraphicColumnRankInterpretation	none	resqpy.strata.StratigraphicColumnRank
obj_StratigraphicOccurrenceInterpretation		
obj_StratigraphicUnitFeature	none	resqpy.strata.StratigraphicUnitFeature
obj_StratigraphicUnitInterpretation	none	resqpy.strata.StratigraphicUnitInterpretati
obj_StreamlinesFeature		
obj_StreamlinesRepresentation		
obj_StringTableLookup	none	resqpy.property.StringLookup
obj_StructuralOrganizationInterpretation		
obj_SubRepresentation		
obj_TectonicBoundaryFeature	none	resqpy.organize.TectonicBoundaryFeature
obj_TimeSeries	none	resqpy.time_series.TimeSeries
obj_TriangulatedSetRepresentation	large	resqpy.surface.Surface
obj_TruncatedIjkGridRepresentation		
obj_TruncatedUnstructuredColumnLayerGridRepresentation		
obj_UnstructuredColumnLayerGridRepresentation		
obj_UnstructuredGridRepresentation	large	resqpy.unstructured.UnstructuredGrid
obj_WellboreFeature	none	resqpy.organize.WellboreFeature
obj_WellboreFrameRepresentation	small	resqpy.well.WellboreFrame
obj_WellboreInterpretation	none	resqpy.organize.WellboreInterpretation
obj_WellboreMarkerFrameRepresentation	small	resqpy.well.WellboreMarkerFrame
obj_WellboreTrajectoryRepresentation	small	resqpy.well.Trajectory

6.5 Attributes of High Level Objects

The previous tutorial discussed some of the differences between RESQML and resqpy objects and included a table showing the equivalent class names. This page discusses attributes of these high level objects and includes a table of the RESQML attribute names and structure for each RESQML class. (A later release will include the equivalent resqpy attribute names.)

Each Energistics schema is defined by a set of *xsd* (xml schema definition) files. To avoid duplication, the content of these files represents a complex structure with a lot of references to intermediate sub-types. The tables included here have been generated from the xsd files for RESQML v2.0.1 by expanding all the inherited items for each class, making a (nearly) complete list of attributes for each.

Each leaf node in the tree structures shown in these tables is one of the following:

- a scalar integer or floating point value
- a string, either unrestricted or from a set of valid values
- a reference to another high level object
- a numerical array, usually defined in an abstract way allowing for different physical representations
- some other abstract complex type, meaning there is a choice of data substructure not listed here

The classes are in alphabetical order. The first column shows the RESQML xml tag, indented to show the hierarchical structure. The second column shows the data type, including the xml namespace. The third column indicates whether the field is *required* (exactly one instance); *optional* (0 or 1); *0 or more*; or *1 or more*.

NB: The tables below are auto-generated and have not been extensively checked. Needless to say, Energistics documentation is authoritative in the case of any inconsistencies.

6.5.1 obj_Activity

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Parent	eml:DataObjectReference	optional
ActivityDescriptor	eml:DataObjectReference	required
Parameter	resqml:AbstractActivityParameter	1 or more
···Title	xs:string	required
···Index	xs:long	optional
···Selection	xs:string	optional
···Key	resqml:AbstractParameterKey	0 or more

6.5.2 obj_ActivityTemplate

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Parameter	resqml:ParameterTemplate	1 or more
···KeyConstraint	xs:string	0 or more
···IsInput	xs:boolean	required
···AllowedKind	resqml:ParameterKind	0 or more
···IsOutput	xs:boolean	required
···Title	xs:string	required
···DataObjectContentType	xs:string	optional
···MaxOccurs	xs:long	required
···MinOccurs	xs:long	required
···Constraint	xs:string	optional
···DefaultValue	resqml:AbstractActivityParameter	0 or more
·····Title	xs:string	required
·····Index	xs:long	optional
·····Selection	xs:string	optional
·····Key	resqml:AbstractParameterKey	0 or more

6.5.3 obj_BlockedWellboreRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
NodeCount	xs:positiveInteger	required
NodeMd	resqml:AbstractDoubleArray	required
WitsmlLogReference	eml:DataObjectReference	optional
IntervalStratigraphiUnits	resqml:IntervalStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
CellFluidPhaseUnits	resqml:CellFluidPhaseUnits	optional
···PhaseUnitIndices	resqml:AbstractIntegerArray	required
···FluidOrganization	eml:DataObjectReference	required
Trajectory	eml:DataObjectReference	required
CellCount	xs:nonNegativeInteger	required
CellIndices	resqml:AbstractIntegerArray	required
GridIndices	resqml:AbstractIntegerArray	required
LocalFacePairPerCellIndices	resqml:AbstractIntegerArray	required
Grid	eml:DataObjectReference	1 or more

6.5.4 obj_BoundaryFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required

6.5.5 obj_BoundaryFeatureInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required

6.5.6 obj_CategoricalProperty

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Count	xs:positiveInteger	required
IndexableElement	resqml:IndexableElements	required
RealizationIndex	xs:nonNegativeInteger	optional
TimeStep	xs:nonNegativeInteger	optional
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required
LocalCrs	eml:DataObjectReference	optional
PropertyKind	resqml:AbstractPropertyKind	required
PatchOfValues	resqml:PatchOfValues	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Values	resqml:AbstractValueArray	required
Facet	resqml:PropertyKindFacet	0 or more
···Facet	resqml:Facet	required
···Value	xs:string	required
Lookup	eml:DataObjectReference	required

6.5.7 obj_CategoricalPropertySeries

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Count	xs:positiveInteger	required
IndexableElement	resqml:IndexableElements	required
RealizationIndex	xs:nonNegativeInteger	optional
TimeStep	xs:nonNegativeInteger	optional
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required
LocalCrs	eml:DataObjectReference	optional
PropertyKind	resqml:AbstractPropertyKind	required
PatchOfValues	resqml:PatchOfValues	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Values	resqml:AbstractValueArray	required
Facet	resqml:PropertyKindFacet	0 or more
···Facet	resqml:Facet	required
···Value	xs:string	required
Lookup	eml:DataObjectReference	required
RealizationIndices	resqml:AbstractIntegerArray	optional
SeriesTimeIndices	resqml:TimeIndices	optional
···TimeIndexCount	xs:positiveInteger	required
···TimeIndexStart	xs:nonNegativeInteger	optional
···SimulatorTimeStep	resqml:AbstractIntegerArray	optional
···UseInterval	xs:boolean	required
···TimeSeries	eml:DataObjectReference	required

6.5.8 obj_CommentProperty

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Count	xs:positiveInteger	required
IndexableElement	resqml:IndexableElements	required
RealizationIndex	xs:nonNegativeInteger	optional
TimeStep	xs:nonNegativeInteger	optional
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required
LocalCrs	eml:DataObjectReference	optional
PropertyKind	resqml:AbstractPropertyKind	required
PatchOfValues	resqml:PatchOfValues	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Values	resqml:AbstractValueArray	required
Facet	resqml:PropertyKindFacet	0 or more
···Facet	resqml:Facet	required
···Value	xs:string	required
Language	xs:string	optional

6.5.9 obj_CommentPropertySeries

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Count	xs:positiveInteger	required
IndexableElement	resqml:IndexableElements	required
RealizationIndex	xs:nonNegativeInteger	optional
TimeStep	xs:nonNegativeInteger	optional
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required
LocalCrs	eml:DataObjectReference	optional
PropertyKind	resqml:AbstractPropertyKind	required
PatchOfValues	resqml:PatchOfValues	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Values	resqml:AbstractValueArray	required
Facet	resqml:PropertyKindFacet	0 or more
···Facet	resqml:Facet	required
···Value	xs:string	required
Language	xs:string	optional
RealizationIndices	resqml:AbstractIntegerArray	optional
SeriesTimeIndices	resqml:TimeIndices	optional
···TimeIndexCount	xs:positiveInteger	required
···TimeIndexStart	xs:nonNegativeInteger	optional
···SimulatorTimeStep	resqml:AbstractIntegerArray	optional
···UseInterval	xs:boolean	required
···TimeSeries	eml:DataObjectReference	required

6.5.10 obj_ContinuousProperty

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Count	xs:positiveInteger	required
IndexableElement	resqml:IndexableElements	required
RealizationIndex	xs:nonNegativeInteger	optional
TimeStep	xs:nonNegativeInteger	optional
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required
LocalCrs	eml:DataObjectReference	optional
PropertyKind	resqml:AbstractPropertyKind	required
PatchOfValues	resqml:PatchOfValues	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Values	resqml:AbstractValueArray	required
Facet	resqml:PropertyKindFacet	0 or more
···Facet	resqml:Facet	required
···Value	xs:string	required
MinimumValue	xs:double	0 or more
MaximumValue	xs:double	0 or more
UOM	resqml:ResqmlUom	required

6.5.11 obj_ContinuousPropertySeries

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Count	xs:positiveInteger	required
IndexableElement	resqml:IndexableElements	required
RealizationIndex	xs:nonNegativeInteger	optional
TimeStep	xs:nonNegativeInteger	optional
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required
LocalCrs	eml:DataObjectReference	optional
PropertyKind	resqml:AbstractPropertyKind	required
PatchOfValues	resqml:PatchOfValues	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Values	resqml:AbstractValueArray	required
Facet	resqml:PropertyKindFacet	0 or more
···Facet	resqml:Facet	required
···Value	xs:string	required
MinimumValue	xs:double	0 or more
MaximumValue	xs:double	0 or more
UOM	resqml:ResqmlUom	required
RealizationIndices	resqml:AbstractIntegerArray	optional
SeriesTimeIndices	resqml:TimeIndices	optional
···TimeIndexCount	xs:positiveInteger	required
···TimeIndexStart	xs:nonNegativeInteger	optional
···SimulatorTimeStep	resqml:AbstractIntegerArray	optional
···UseInterval	xs:boolean	required
···TimeSeries	eml:DataObjectReference	required

6.5.12 obj_DeviationSurveyRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
WitsmlDeviationSurvey	eml:DataObjectReference	optional
IsFinal	xs:boolean	required
StationCount	xs:positiveInteger	required
MdUom	eml:LengthUom	required
Mds	resqml:AbstractDoubleArray	required
FirstStationLocation	resqml:Point3d	required
···Coordinate1	xs:double	required
···Coordinate2	xs:double	required
···Coordinate3	xs:double	required
AngleUom	eml:PlaneAngleUom	required
Azimuths	resqml:AbstractDoubleArray	required
Inclinations	resqml:AbstractDoubleArray	required
MdDatum	eml:DataObjectReference	required
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required

6.5.13 obj_DiscreteProperty

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Count	xs:positiveInteger	required
IndexableElement	resqml:IndexableElements	required
RealizationIndex	xs:nonNegativeInteger	optional
TimeStep	xs:nonNegativeInteger	optional
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required
LocalCrs	eml:DataObjectReference	optional
PropertyKind	resqml:AbstractPropertyKind	required
PatchOfValues	resqml:PatchOfValues	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Values	resqml:AbstractValueArray	required
Facet	resqml:PropertyKindFacet	0 or more
···Facet	resqml:Facet	required
···Value	xs:string	required
MinimumValue	xs:integer	0 or more
MaximumValue	xs:integer	0 or more

6.5.14 obj_DiscretePropertySeries

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Count	xs:positiveInteger	required
IndexableElement	resqml:IndexableElements	required
RealizationIndex	xs:nonNegativeInteger	optional
TimeStep	xs:nonNegativeInteger	optional
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required
LocalCrs	eml:DataObjectReference	optional
PropertyKind	resqml:AbstractPropertyKind	required
PatchOfValues	resqml:PatchOfValues	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Values	resqml:AbstractValueArray	required
Facet	resqml:PropertyKindFacet	0 or more
···Facet	resqml:Facet	required
···Value	xs:string	required
MinimumValue	xs:integer	0 or more
MaximumValue	xs:integer	0 or more
RealizationIndices	resqml:AbstractIntegerArray	optional
SeriesTimeIndices	resqml:TimeIndices	optional
···TimeIndexCount	xs:positiveInteger	required
···TimeIndexStart	xs:nonNegativeInteger	optional
···SimulatorTimeStep	resqml:AbstractIntegerArray	optional
···UseInterval	xs:boolean	required
···TimeSeries	eml:DataObjectReference	required

6.5.15 obj_DoubleTableLookup

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Value	resqml:DoubleLookup	1 or more
···Key	xs:double	required
···Value	xs:double	required

6.5.16 obj_EarthModelInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
StratigraphicOccurrences	eml:DataObjectReference	0 or more
StratigraphicColumn	eml:DataObjectReference	optional
Structure	eml:DataObjectReference	optional
Fluid	eml:DataObjectReference	optional

6.5.17 obj_EpcExternalPartReference

MimeType	xs:string	required

6.5.18 obj_FaultInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
IsListric	xs:boolean	optional
MaximumThrow	eml:LengthMeasure	optional
MeanAzimuth	eml:PlaneAngleMeasure	optional
MeanDip	eml:PlaneAngleMeasure	optional
ThrowInterpretation	resqml:FaultThrow	0 or more
···Throw	resqml:ThrowKind	1 or more
···HasOccuredDuring	resqml:TimeInterval	optional
·····ChronoBottom	eml:DataObjectReference	required
·····ChronoTop	eml:DataObjectReference	required

6.5.19 obj_FluidBoundaryFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
FluidContact	resqml:FluidContact	required

6.5.20 obj_FrontierFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required

6.5.21 obj_GenericFeatureInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required

6.5.22 obj_GeneticBoundaryFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
GeneticBoundaryKind	resqml:GeneticBoundaryKind	required
AbsoluteAge	resqml:Timestamp	optional
···DateTime	xs:dateTime	required
···YearOffset	xs:long	optional

6.5.23 obj_GeobodyBoundaryInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
BoundaryRelation	resqml:BoundaryRelation	0 or more

6.5.24 obj_GeobodyFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required

6.5.25 obj_GeobodyInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
GeologicUnitComposition	resqml:GeologicUnitComposition	optional
GeologicUnitMaterialImplacement	resqml:GeologicUnitMaterialImplacement	optional
Geobody3dShape	resqml:Geobody3dShape	optional

6.5.26 obj_GeologicUnitFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required

6.5.27 obj_GeologicUnitInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
GeologicUnitComposition	resqml:GeologicUnitComposition	optional
GeologicUnitMaterialImplacement	resqml:GeologicUnitMaterialImplacement	optional

6.5.28 obj_GlobalChronostratigraphicColumn

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
ChronostratigraphicColumnComponent	resqml:ChronostratigraphicRank	1 or more
···Name	eml:NameString	required
···Contains	eml:DataObjectReference	1 or more

6.5.29 obj_GpGridRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
CellFluidPhaseUnits	resqml:CellFluidPhaseUnits	optional
···PhaseUnitIndices	resqml:AbstractIntegerArray	required
···FluidOrganization	eml:DataObjectReference	required
ParentWindow	resqml:AbstractParentWindow	optional
···CellOverlap	resqml:CellOverlap	optional
·····Count	xs:positiveInteger	required
·····ParentChildCellPairs	resqml:AbstractIntegerArray	required
·····OverlapVolume	resqml:OverlapVolume	optional
·····VolumeUom	eml:VolumeUom	required
·····OverlapVolumes	resqml:AbstractDoubleArray	required
CellStratigraphicUnits	resqml:CellStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
ColumnLayerGrid	resqml:GpGridColumnLayerGrid	0 or more
···Nk	xs:nonNegativeInteger	required
···KGaps	resqml:KGaps	optional
·····Count	xs:positiveInteger	required
·····GapAfterLayer	resqml:AbstractBooleanArray	required
···IjkGridPatch	resqml:GpGridIjkGridPatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····Ni	xs:nonNegativeInteger	required
·····Nj	xs:nonNegativeInteger	required
·····RadialGridIsComplete	xs:boolean	optional
·····Geometry	resqml:IjkGridGeometry	optional
·····TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····LocalCrs	eml:DataObjectReference	required
·····Points	resqml:AbstractPoint3dArray	required
·····SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required
·····AdditionalGridPoints	resqml:AdditionalGridPoints	0 or more
·····RepresentationPatchIndex	xs:nonNegativeInteger	optional
·····Attachment	resqml:GridGeometryAttachment	required

continues on next page

Table 2 – continued from previous page

····Points	resqml:AbstractPoint3dArray	required
····KDirection	resqml:KDirection	required
····PillarGeometryIsDefined	resqml:AbstractBooleanArray	required
····PillarShape	resqml:PillarShape	required
····CellGeometryIsDefined	resqml:AbstractBooleanArray	optional
····NodeIsColocatedInKDirection	resqml:AbstractBooleanArray	optional
····NodeIsColocatedOnKEdge	resqml:AbstractBooleanArray	optional
····SubnodeTopology	resqml:ColumnLayerSubnodeTopology	optional
····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
····PatchIndex	xs:nonNegativeInteger	required
····SubnodeNodeObject	resqml:SubnodeNodeObject	required
····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
····ObjectIndices	resqml:AbstractIntegerArray	required
····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
····PatchIndex	xs:nonNegativeInteger	required
····SubnodeNodeObject	resqml:SubnodeNodeObject	required
····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
····SubnodeCountPerObject	xs:positiveInteger	required
····ColumnSubnodes	resqml:ColumnSubnodePatch	0 or more
····PatchIndex	xs:nonNegativeInteger	required
····SubnodeNodeObject	resqml:SubnodeNodeObject	required
····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
····SubnodeCountPerObject	resqml:AbstractIntegerArray	required
····SplitCoordinateLines	resqml:ColumnLayerSplitCoordinateLines	optional
····Count	xs:positiveInteger	required
····PillarIndices	resqml:AbstractIntegerArray	required
····ColumnsPerSplitCoordinateLine	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
····SplitColumnEdges	resqml:ColumnLayerSplitColumnEdges	optional
····Count	xs:positiveInteger	required
····ParentColumnEdgeIndices	resqml:AbstractIntegerArray	required
····ColumnPerSplitColumnEdge	resqml:AbstractIntegerArray	required
····SplitNodes	resqml:SplitNodePatch	optional
····PatchIndex	xs:nonNegativeInteger	required
····Count	xs:positiveInteger	required
····ParentNodeIndices	resqml:AbstractIntegerArray	required
····CellsPerSplitNode	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
····SplitFaces	resqml:SplitFaces	optional
····Count	xs:positiveInteger	required
····ParentFaceIndices	resqml:AbstractIntegerArray	required
····CellIndices	resqml:AbstractIntegerArray	required
····SplitEdges	resqml:SplitEdges	optional
····Count	xs:positiveInteger	required
····ParentEdgeIndices	resqml:AbstractIntegerArray	required
····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required

continues on next page

Table 2 – continued from previous page

····CumulativeLength	resqml:AbstractIntegerArray	required
···GridIsRighthanded	xs:boolean	required
···IjGaps	resqml:IjGaps	optional
···SplitPillarCount	xs:positiveInteger	required
···ParentPillarIndices	resqml:AbstractIntegerArray	required
···ColumnsPerSplitPillar	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
···IjSplitColumnEdges	resqml:IjSplitColumnEdges	optional
····Count	xs:positiveInteger	required
····PillarsPerSplitColumnEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
··TruncationCells	resqml:TruncationCellPatch	optional
···PatchIndex	xs:nonNegativeInteger	required
···TruncationNodeCount	xs:positiveInteger	required
···TruncationFaceCount	xs:positiveInteger	required
···TruncationCellCount	xs:positiveInteger	required
···NodesPerTruncationFace	resqml:ResqmlJaggedArray	required
···Elements	resqml:AbstractValueArray	required
···CumulativeLength	resqml:AbstractIntegerArray	required
···ParentCellIndices	resqml:AbstractIntegerArray	required
···LocalFacesPerCell	resqml:ResqmlJaggedArray	required
···Elements	resqml:AbstractValueArray	required
···CumulativeLength	resqml:AbstractIntegerArray	required
···TruncationFacesPerCell	resqml:ResqmlJaggedArray	required
···Elements	resqml:AbstractValueArray	required
···CumulativeLength	resqml:AbstractIntegerArray	required
···TruncationCellFaceIsRightHanded	resqml:AbstractBooleanArray	required
·UnstructuredColumnLayerGridPatch	resqml:GpGridUnstructuredColumnLayerGridPatch	0 or more
··PatchIndex	xs:nonNegativeInteger	required
··UnstructuredColumnCount	xs:nonNegativeInteger	required
··Geometry	resqml:UnstructuredColumnLayerGridGeometry	optional
···TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
···LocalCrs	eml:DataObjectReference	required
···Points	resqml:AbstractPoint3dArray	required
···SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
···SeismicSupport	eml:DataObjectReference	required
···AdditionalGridPoints	resqml:AdditionalGridPoints	0 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Attachment	resqml:GridGeometryAttachment	required
···Points	resqml:AbstractPoint3dArray	required
···KDirection	resqml:KDirection	required
···PillarGeometryIsDefined	resqml:AbstractBooleanArray	required
···PillarShape	resqml:PillarShape	required
···CellGeometryIsDefined	resqml:AbstractBooleanArray	optional
···NodeIsColocatedInKDirection	resqml:AbstractBooleanArray	optional
···NodeIsColocatedOnKEdge	resqml:AbstractBooleanArray	optional

continues on next page

Table 2 – continued from previous page

·····SubnodeTopology	resqml:ColumnLayerSubnodeTopology	optional
·····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····ObjectIndices	resqml:AbstractIntegerArray	required
·····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
·····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	xs:positiveInteger	required
·····ColumnSubnodes	resqml:ColumnSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	resqml:AbstractIntegerArray	required
·····SplitCoordinateLines	resqml:ColumnLayerSplitCoordinateLines	optional
·····Count	xs:positiveInteger	required
·····PillarIndices	resqml:AbstractIntegerArray	required
·····ColumnsPerSplitCoordinateLine	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitColumnEdges	resqml:ColumnLayerSplitColumnEdges	optional
·····Count	xs:positiveInteger	required
·····ParentColumnEdgeIndices	resqml:AbstractIntegerArray	required
·····ColumnPerSplitColumnEdge	resqml:AbstractIntegerArray	required
·····SplitNodes	resqml:SplitNodePatch	optional
·····PatchIndex	xs:nonNegativeInteger	required
·····Count	xs:positiveInteger	required
·····ParentNodeIndices	resqml:AbstractIntegerArray	required
·····CellsPerSplitNode	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitFaces	resqml:SplitFaces	optional
·····Count	xs:positiveInteger	required
·····ParentFaceIndices	resqml:AbstractIntegerArray	required
·····CellIndices	resqml:AbstractIntegerArray	required
·····SplitEdges	resqml:SplitEdges	optional
·····Count	xs:positiveInteger	required
·····ParentEdgeIndices	resqml:AbstractIntegerArray	required
·····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····ColumnShape	resqml:ColumnShape	required
·····PillarCount	xs:positiveInteger	required
·····PillarsPerColumn	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····ColumnIsRightHanded	resqml:AbstractBooleanArray	required

continues on next page

Table 2 – continued from previous page

·····ColumnEdges	resqml:UnstructuredColumnEdges	optional
·····Count	xs:positiveInteger	required
·····PillarsPerColumnEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····TruncationCells	resqml:TruncationCellPatch	optional
·····PatchIndex	xs:nonNegativeInteger	required
·····TruncationNodeCount	xs:positiveInteger	required
·····TruncationFaceCount	xs:positiveInteger	required
·····TruncationCellCount	xs:positiveInteger	required
·····NodesPerTruncationFace	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····ParentCellIndices	resqml:AbstractIntegerArray	required
·····LocalFacesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····TruncationFacesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····TruncationCellFaceIsRightHanded	resqml:AbstractBooleanArray	required
UnstructuredGridPatch	resqml:GpGridUnstructuredGridPatch	0 or more
···PatchIndex	xs:nonNegativeInteger	required
···UnstructuredCellCount	xs:nonNegativeInteger	required
···Geometry	resqml:UnstructuredGridGeometry	optional
·····TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····LocalCrs	eml:DataObjectReference	required
·····Points	resqml:AbstractPoint3dArray	required
·····SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required
·····AdditionalGridPoints	resqml:AdditionalGridPoints	0 or more
·····RepresentationPatchIndex	xs:nonNegativeInteger	optional
·····Attachment	resqml:GridGeometryAttachment	required
·····Points	resqml:AbstractPoint3dArray	required
·····CellShape	resqml:CellShape	required
·····NodeCount	xs:positiveInteger	required
·····FaceCount	xs:positiveInteger	required
·····NodesPerFace	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····FacesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····CellFaceIsRightHanded	resqml:AbstractBooleanArray	required
·····HingeNodeFaces	resqml:UnstructuredGridHingeNodeFaces	optional
·····Count	xs:positiveInteger	required
·····FaceIndices	resqml:AbstractIntegerArray	required
·····SubnodeTopology	resqml:UnstructuredSubnodeTopology	optional

continues on next page

Table 2 – continued from previous page

·····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····ObjectIndices	resqml:AbstractIntegerArray	required
·····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
·····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	xs:positiveInteger	required
·····Edges	resqml:Edges	optional
·····Count	xs:positiveInteger	required
·····NodesPerEdge	resqml:AbstractIntegerArray	required
·····NodesPerCell	resqml:NodesPerCell	optional
·····NodesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required

6.5.30 obj_Grid2dRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
SurfaceRole	resqml:SurfaceRole	required
Boundaries	resqml:PatchBoundaries	0 or more
···InnerRing	eml:DataObjectReference	0 or more
···OuterRing	eml:DataObjectReference	optional
···ReferencedPatch	xs:nonNegativeInteger	required
Grid2dPatch	resqml:Grid2dPatch	required
···PatchIndex	xs:nonNegativeInteger	required
···FastestAxisCount	xs:positiveInteger	required
···SlowestAxisCount	xs:positiveInteger	required
···Geometry	resqml:PointGeometry	required
·····TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····LocalCrs	eml:DataObjectReference	required
·····Points	resqml:AbstractPoint3dArray	required
·····SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required

6.5.31 obj_Grid2dSetRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
SurfaceRole	resqml:SurfaceRole	required
Boundaries	resqml:PatchBoundaries	0 or more
···InnerRing	eml:DataObjectReference	0 or more
···OuterRing	eml:DataObjectReference	optional
···ReferencedPatch	xs:nonNegativeInteger	required
Grid2dPatch	resqml:Grid2dPatch	2 or more
···PatchIndex	xs:nonNegativeInteger	required
···FastestAxisCount	xs:positiveInteger	required
···SlowestAxisCount	xs:positiveInteger	required
···Geometry	resqml:PointGeometry	required
·····TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····LocalCrs	eml:DataObjectReference	required
·····Points	resqml:AbstractPoint3dArray	required
·····SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required

6.5.32 obj_GridConnectionSetRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
Count	xs:positiveInteger	required
CellIndexPairs	resqml:AbstractIntegerArray	required
GridIndexPairs	resqml:AbstractIntegerArray	optional
LocalFacePerCellIndexPairs	resqml:AbstractIntegerArray	optional
ConnectionInterpretations	resqml:ConnectionInterpretations	optional
···InterpretationIndices	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···FeatureInterpretation	eml:DataObjectReference	1 or more
Grid	eml:DataObjectReference	1 or more

6.5.33 obj_HorizonInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
BoundaryRelation	resqml:BoundaryRelation	0 or more
SequenceStratigraphySurface	resqml:SequenceStratigraphySurface	optional

6.5.34 obj_IjkGridRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
CellFluidPhaseUnits	resqml:CellFluidPhaseUnits	optional
···PhaseUnitIndices	resqml:AbstractIntegerArray	required
···FluidOrganization	eml:DataObjectReference	required
ParentWindow	resqml:AbstractParentWindow	optional
···CellOverlap	resqml:CellOverlap	optional
·····Count	xs:positiveInteger	required
·····ParentChildCellPairs	resqml:AbstractIntegerArray	required
·····OverlapVolume	resqml:OverlapVolume	optional
·····VolumeUom	eml:VolumeUom	required
·····OverlapVolumes	resqml:AbstractDoubleArray	required
CellStratigraphicUnits	resqml:CellStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
Nk	xs:positiveInteger	required
IntervalStratigraphicUnits	resqml:IntervalStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
Ni	xs:positiveInteger	required
Nj	xs:positiveInteger	required
RadialGridIsComplete	xs:boolean	optional
KGaps	resqml:KGaps	optional
···Count	xs:positiveInteger	required
···GapAfterLayer	resqml:AbstractBooleanArray	required
Geometry	resqml:IjkGridGeometry	optional
···TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
···LocalCrs	eml:DataObjectReference	required
···Points	resqml:AbstractPoint3dArray	required
···SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional

continues on next page

Table 3 – continued from previous page

·····SeismicSupport	eml:DataObjectReference	required
···AdditionalGridPoints	resqml:AdditionalGridPoints	0 or more
·····RepresentationPatchIndex	xs:nonNegativeInteger	optional
·····Attachment	resqml:GridGeometryAttachment	required
·····Points	resqml:AbstractPoint3dArray	required
···KDirection	resqml:KDirection	required
···PillarGeometryIsDefined	resqml:AbstractBooleanArray	required
···PillarShape	resqml:PillarShape	required
···CellGeometryIsDefined	resqml:AbstractBooleanArray	optional
···NodeIsColocatedInKDirection	resqml:AbstractBooleanArray	optional
···NodeIsColocatedOnKEdge	resqml:AbstractBooleanArray	optional
···SubnodeTopology	resqml:ColumnLayerSubnodeTopology	optional
·····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····ObjectIndices	resqml:AbstractIntegerArray	required
·····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
·····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	xs:positiveInteger	required
·····ColumnSubnodes	resqml:ColumnSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	resqml:AbstractIntegerArray	required
···SplitCoordinateLines	resqml:ColumnLayerSplitCoordinateLines	optional
·····Count	xs:positiveInteger	required
·····PillarIndices	resqml:AbstractIntegerArray	required
·····ColumnsPerSplitCoordinateLine	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitColumnEdges	resqml:ColumnLayerSplitColumnEdges	optional
·····Count	xs:positiveInteger	required
·····ParentColumnEdgeIndices	resqml:AbstractIntegerArray	required
·····ColumnPerSplitColumnEdge	resqml:AbstractIntegerArray	required
···SplitNodes	resqml:SplitNodePatch	optional
·····PatchIndex	xs:nonNegativeInteger	required
·····Count	xs:positiveInteger	required
·····ParentNodeIndices	resqml:AbstractIntegerArray	required
·····CellsPerSplitNode	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitFaces	resqml:SplitFaces	optional
·····Count	xs:positiveInteger	required
·····ParentFaceIndices	resqml:AbstractIntegerArray	required
·····CellIndices	resqml:AbstractIntegerArray	required
·····SplitEdges	resqml:SplitEdges	optional

continues on next page

Table 3 – continued from previous page

····Count	xs:positiveInteger	required
····ParentEdgeIndices	resqml:AbstractIntegerArray	required
····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
···GridIsRighthanded	xs:boolean	required
···IjGaps	resqml:IjGaps	optional
····SplitPillarCount	xs:positiveInteger	required
····ParentPillarIndices	resqml:AbstractIntegerArray	required
····ColumnsPerSplitPillar	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
····IjSplitColumnEdges	resqml:IjSplitColumnEdges	optional
·····Count	xs:positiveInteger	required
·····PillarsPerSplitColumnEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required

6.5.35 obj_LocalDepth3dCrs

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
YOffset	xs:double	required
ZOffset	xs:double	required
ArealRotation	eml:PlaneAngleMeasure	required
ProjectedAxisOrder	eml:AxisOrder2d	required
ProjectedUom	eml:LengthUom	required
VerticalUom	eml:LengthUom	required
XOffset	xs:double	required
ZIncreasingDownward	xs:boolean	required
VerticalCrs	eml:AbstractVerticalCrs	required
ProjectedCrs	eml:AbstractProjectedCrs	required

6.5.36 obj_LocalGridSet

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Activation	resqml:Activation	optional
···ActivationToggleIndices	resqml:AbstractIntegerArray	required
···TimeSeries	eml:DataObjectReference	required
ChildGrid	eml:DataObjectReference	1 or more

6.5.37 obj_LocalTime3dCrs

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
YOffset	xs:double	required
ZOffset	xs:double	required
ArealRotation	eml:PlaneAngleMeasure	required
ProjectedAxisOrder	eml:AxisOrder2d	required
ProjectedUom	eml:LengthUom	required
VerticalUom	eml:LengthUom	required
XOffset	xs:double	required
ZIncreasingDownward	xs:boolean	required
VerticalCrs	eml:AbstractVerticalCrs	required
ProjectedCrs	eml:AbstractProjectedCrs	required
TimeUom	eml:TimeUom	required

6.5.38 obj_MdDatum

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Location	resqml:Point3d	required
···Coordinate1	xs:double	required
···Coordinate2	xs:double	required
···Coordinate3	xs:double	required
MdReference	resqml:MdReference	required
LocalCrs	eml:DataObjectReference	required

6.5.39 obj_NonSealedSurfaceFrameworkRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
IsHomogeneous	xs:boolean	required
Representation	eml:DataObjectReference	1 or more
ContactIdentity	resqml:ContactIdentity	0 or more
···IdentityKind	resqml:IdentityKind	required
···ListOfContactRepresentations	resqml:AbstractIntegerArray	required
···ListOfIdenticalNodes	resqml:AbstractIntegerArray	optional
NonSealedContactRepresentation	resqml:AbstractContactRepresentationPart	0 or more
···Index	xs:nonNegativeInteger	required

6.5.40 obj_OrganizationFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
OrganizationKind	resqml:OrganizationKind	required

6.5.41 obj_PlaneSetRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
SurfaceRole	resqml:SurfaceRole	required
Boundaries	resqml:PatchBoundaries	0 or more
···InnerRing	eml:DataObjectReference	0 or more
···OuterRing	eml:DataObjectReference	optional
···ReferencedPatch	xs:nonNegativeInteger	required
Planes	resqml:AbstractPlaneGeometry	1 or more
···TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
···LocalCrs	eml:DataObjectReference	required

6.5.42 obj_PointSetRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
NodePatch	resqml:NodePatch	1 or more
···PatchIndex	xs:nonNegativeInteger	required
···Count	xs:positiveInteger	required
···Geometry	resqml:PointGeometry	required
·····TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····LocalCrs	eml:DataObjectReference	required
·····Points	resqml:AbstractPoint3dArray	required
·····SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required

6.5.43 obj_PointsProperty

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Count	xs:positiveInteger	required
IndexableElement	resqml:IndexableElements	required
RealizationIndex	xs:nonNegativeInteger	optional
TimeStep	xs:nonNegativeInteger	optional
TimeIndex	resqml:TimeIndex	optional
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required
LocalCrs	eml:DataObjectReference	optional
PropertyKind	resqml:AbstractPropertyKind	required
PatchOfPoints	resqml:PatchOfPoints	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Points	resqml:AbstractPoint3dArray	required

6.5.44 obj_PolylineRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
LineRole	resqml:LineRole	optional
IsClosed	xs:boolean	required
NodePatch	resqml:NodePatch	required
···PatchIndex	xs:nonNegativeInteger	required
···Count	xs:positiveInteger	required
···Geometry	resqml:PointGeometry	required
·····TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····LocalCrs	eml:DataObjectReference	required
·····Points	resqml:AbstractPoint3dArray	required
·····SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required

6.5.45 obj_PolylineSetRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
LineRole	resqml:LineRole	optional
LinePatch	resqml:PolylineSetPatch	1 or more
···PatchIndex	xs:nonNegativeInteger	required
···ClosedPolylines	resqml:AbstractBooleanArray	required
···NodeCountPerPolyline	resqml:AbstractIntegerArray	required
···Geometry	resqml:PointGeometry	required
·····TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····LocalCrs	eml:DataObjectReference	required
·····Points	resqml:AbstractPoint3dArray	required
·····SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required

6.5.46 obj_PropertyKind

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
NamingSystem	xs:anyURI	required
IsAbstract	xs:boolean	required
RepresentativeUom	resqml:ResqmlUom	required
ParentPropertyKind	resqml:AbstractPropertyKind	required

6.5.47 obj_PropertySet

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
TimeSetKind	resqml:TimeSetKind	required
HasSinglePropertyKind	xs:boolean	required
HasMultipleRealizations	xs:boolean	required
ParentSet	eml:DataObjectReference	0 or more
Properties	eml:DataObjectReference	1 or more

6.5.48 obj_RedefinedGeometryRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
PatchOfGeometry	resqml:PatchOfGeometry	1 or more
···RepresentationPatchIndex	xs:nonNegativeInteger	optional
···Geometry	resqml:AbstractGeometry	required
·····TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····LocalCrs	eml:DataObjectReference	required
SupportingRepresentation	eml:DataObjectReference	required

6.5.49 obj_RepresentationIdentitySet

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentationIdentity	resqml:RepresentationIdentity	1 or more
···IdenticalElementCount	xs:positiveInteger	required
···ElementIdentity	resqml:ElementIdentity	2 or more
·····ElementIndices	resqml:AbstractIntegerArray	optional
·····IdentityKind	resqml:IdentityKind	required
·····IndexableElement	resqml:IndexableElements	required
·····Representation	eml:DataObjectReference	required
·····FromTimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····ToTimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
···AdditionalGridTopology	resqml:AdditionalGridTopology	optional
·····SplitEdges	resqml:SplitEdges	optional
·····Count	xs:positiveInteger	required
·····ParentEdgeIndices	resqml:AbstractIntegerArray	required
·····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitNodes	resqml:SplitNodePatch	optional
·····PatchIndex	xs:nonNegativeInteger	required
·····Count	xs:positiveInteger	required
·····ParentNodeIndices	resqml:AbstractIntegerArray	required
·····CellsPerSplitNode	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitFaces	resqml:SplitFaces	optional
·····Count	xs:positiveInteger	required

continues on next page

Table 4 – continued from previous page

····ParentFaceIndices	resqml:AbstractIntegerArray	required
····CellIndices	resqml:AbstractIntegerArray	required
····SplitEdges	resqml:SplitEdges	optional
····Count	xs:positiveInteger	required
····ParentEdgeIndices	resqml:AbstractIntegerArray	required
····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
··SplitColumnEdges	resqml:ColumnLayerSplitColumnEdges	optional
··Count	xs:positiveInteger	required
··ParentColumnEdgeIndices	resqml:AbstractIntegerArray	required
··ColumnPerSplitColumnEdge	resqml:AbstractIntegerArray	required
··UnstructuredColumnEdges	resqml:UnstructuredColumnEdges	optional
··Count	xs:positiveInteger	required
··PillarsPerColumnEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
··SplitFaces	resqml:SplitFaces	optional
··Count	xs:positiveInteger	required
··ParentFaceIndices	resqml:AbstractIntegerArray	required
··CellIndices	resqml:AbstractIntegerArray	required
··SplitEdges	resqml:SplitEdges	optional
····Count	xs:positiveInteger	required
····ParentEdgeIndices	resqml:AbstractIntegerArray	required
····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
··IjSplitColumnEdges	resqml:IjSplitColumnEdges	optional
··Count	xs:positiveInteger	required
··PillarsPerSplitColumnEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
··UnstructuredSubnodeTopology	resqml:UnstructuredSubnodeTopology	optional
··VariableSubnodes	resqml:VariableSubnodePatch	0 or more
····PatchIndex	xs:nonNegativeInteger	required
····SubnodeNodeObject	resqml:SubnodeNodeObject	required
····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
····ObjectIndices	resqml:AbstractIntegerArray	required
····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
··UniformSubnodes	resqml:UniformSubnodePatch	0 or more
····PatchIndex	xs:nonNegativeInteger	required
····SubnodeNodeObject	resqml:SubnodeNodeObject	required
····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
····SubnodeCountPerObject	xs:positiveInteger	required
··Edges	resqml:Edges	optional
····Count	xs:positiveInteger	required
····NodesPerEdge	resqml:AbstractIntegerArray	required
··NodesPerCell	resqml:NodesPerCell	optional
····NodesPerCell	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required

continues on next page

Table 4 – continued from previous page

··············CumulativeLength	resqml:AbstractIntegerArray	required
·····ColumnLayerSubnodeTopology	resqml:ColumnLayerSubnodeTopology	optional
·····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
···········PatchIndex	xs:nonNegativeInteger	required
···········SubnodeNodeObject	resqml:SubnodeNodeObject	required
···········NodeWeightsPerSubnode	resqml:AbstractValueArray	required
···········ObjectIndices	resqml:AbstractIntegerArray	required
···········SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
·····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
···········PatchIndex	xs:nonNegativeInteger	required
···········SubnodeNodeObject	resqml:SubnodeNodeObject	required
···········NodeWeightsPerSubnode	resqml:AbstractValueArray	required
···········SubnodeCountPerObject	xs:positiveInteger	required
·····ColumnSubnodes	resqml:ColumnSubnodePatch	0 or more
···········PatchIndex	xs:nonNegativeInteger	required
···········SubnodeNodeObject	resqml:SubnodeNodeObject	required
···········NodeWeightsPerSubnode	resqml:AbstractValueArray	required
···········SubnodeCountPerObject	resqml:AbstractIntegerArray	required

6.5.50 obj_RepresentationSetRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
IsHomogeneous	xs:boolean	required
Representation	eml:DataObjectReference	1 or more

6.5.51 obj_RockFluidOrganizationInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
ContactInterpretation	resqml:AbstractContactInterpretationPart	0 or more
···ContactRelationship	resqml:ContactRelationship	required
···Index	xs:nonNegativeInteger	required
···PartOf	eml:DataObjectReference	optional
RockFluidUnitIndex	resqml:RockFluidUnitInterpretationIndex	required
···Index	xs:nonNegativeInteger	required
···RockFluidUnit	eml:DataObjectReference	required

6.5.52 obj_RockFluidUnitFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Phase	resqml:Phase	required
FluidBoundaryBottom	eml:DataObjectReference	required
FluidBoundaryTop	eml:DataObjectReference	required

6.5.53 obj_RockFluidUnitInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
GeologicUnitComposition	resqml:GeologicUnitComposition	optional
GeologicUnitMaterialImplacement	resqml:GeologicUnitMaterialImplacement	optional
Phase	resqml:Phase	optional

6.5.54 obj_SealedSurfaceFrameworkRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
IsHomogeneous	xs:boolean	required
Representation	eml:DataObjectReference	1 or more
ContactIdentity	resqml:ContactIdentity	0 or more
···IdentityKind	resqml:IdentityKind	required
···ListOfContactRepresentations	resqml:AbstractIntegerArray	required
···ListOfIdenticalNodes	resqml:AbstractIntegerArray	optional
SealedContactRepresentation	resqml:SealedContactRepresentationPart	0 or more
···Index	xs:nonNegativeInteger	required
···IdenticalNodeIndices	resqml:AbstractIntegerArray	optional
···IdentityKind	resqml:IdentityKind	required
···Contact	resqml:ContactPatch	2 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····Count	xs:positiveInteger	required
·····RepresentationIndex	xs:nonNegativeInteger	required
·····SupportingRepresentationNodes	resqml:AbstractIntegerArray	required

6.5.55 obj_SealedVolumeFrameworkRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
IsHomogeneous	xs:boolean	required
Representation	eml:DataObjectReference	1 or more
BasedOn	eml:DataObjectReference	required
Shells	resqml:VolumeShell	1 or more
···ShellUid	xs:string	required
···MacroFaces	resqml:OrientedMacroFace	1 or more
·····PatchIndexOfRepresentation	xs:nonNegativeInteger	required
·····RepresentationIndex	xs:nonNegativeInteger	required
·····SideIsPlus	xs:boolean	required
Regions	resqml:VolumeRegion	1 or more
···PatchIndex	xs:nonNegativeInteger	required
···InternalShells	resqml:VolumeShell	0 or more
·····ShellUid	xs:string	required
·····MacroFaces	resqml:OrientedMacroFace	1 or more
·····PatchIndexOfRepresentation	xs:nonNegativeInteger	required
·····RepresentationIndex	xs:nonNegativeInteger	required
·····SideIsPlus	xs:boolean	required
···Represents	eml:DataObjectReference	required
···ExternalShell	resqml:VolumeShell	required
·····ShellUid	xs:string	required
·····MacroFaces	resqml:OrientedMacroFace	1 or more
·····PatchIndexOfRepresentation	xs:nonNegativeInteger	required
·····RepresentationIndex	xs:nonNegativeInteger	required
·····SideIsPlus	xs:boolean	required

6.5.56 obj_SeismicLatticeFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
CrosslineCount	xs:positiveInteger	required
CrosslineIndexIncrement	xs:integer	required
FirstCrosslineIndex	xs:integer	required
FirstInlineIndex	xs:integer	required
InlineCount	xs:positiveInteger	required
InlineIndexIncrement	xs:integer	required
IsPartOf	resqml:SeismicLatticeSetFeature	optional
···ExtraMetadata	resqml:NameValuePair	0 or more
·····Name	xs:string	required
·····Value	xs:string	required

6.5.57 obj_SeismicLineFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
FirstTraceIndex	xs:integer	required
TraceCount	xs:positiveInteger	required
TraceIndexIncrement	xs:integer	required
IsPartOf	eml:DataObjectReference	optional

6.5.58 obj_SeismicLineSetFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required

6.5.59 obj_StratigraphicColumn

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Ranks	eml:DataObjectReference	1 or more

6.5.60 obj_StratigraphicColumnRankInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
ContactInterpretation	resqml:AbstractContactInterpretationPart	0 or more
···ContactRelationship	resqml:ContactRelationship	required
···Index	xs:nonNegativeInteger	required
···PartOf	eml:DataObjectReference	optional
OrderingCriteria	resqml:OrderingCriteria	required
Index	xs:nonNegativeInteger	required
StratigraphicUnits	resqml:StratigraphicUnitInterpretationIndex	1 or more
···Index	xs:nonNegativeInteger	required
···Unit	eml:DataObjectReference	required

6.5.61 obj_StratigraphicOccurrenceInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
ContactInterpretation	resqml:AbstractContactInterpretationPart	0 or more
···ContactRelationship	resqml:ContactRelationship	required
···Index	xs:nonNegativeInteger	required
···PartOf	eml:DataObjectReference	optional
OrderingCriteria	resqml:OrderingCriteria	required
IsOccurrenceOf	eml:DataObjectReference	optional
GeologicUnitIndex	resqml:GeologicUnitInterpretationIndex	0 or more
···Index	xs:nonNegativeInteger	required
···Unit	eml:DataObjectReference	required

6.5.62 obj_StratigraphicUnitFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
ChronostratigraphicBottom	eml:DataObjectReference	optional
ChronostratigraphicTop	eml:DataObjectReference	optional

6.5.63 obj_StratigraphicUnitInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
GeologicUnitComposition	resqml:GeologicUnitComposition	optional
GeologicUnitMaterialImplacement	resqml:GeologicUnitMaterialImplacement	optional
DepositionMode	resqml:DepositionMode	optional
MaxThickness	eml:LengthMeasure	optional
MinThickness	eml:LengthMeasure	optional

6.5.64 obj_StreamlinesFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Flux	resqml:StreamlineFlux	required
OtherFlux	xs:string	optional
TimeIndex	resqml:TimeIndex	required
···Index	xs:nonNegativeInteger	required
···TimeSeries	eml:DataObjectReference	required

6.5.65 obj_StreamlinesRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
LineCount	xs:positiveInteger	required
StreamlineWellbores	resqml:StreamlineWellbores	optional
···InjectorPerLine	resqml:AbstractIntegerArray	required
···ProducerPerLine	resqml:AbstractIntegerArray	required
···WellboreTrajectoryRepresentation	eml:DataObjectReference	1 or more
Geometry	resqml:StreamlinePolylineSetPatch	optional
···PatchIndex	xs:nonNegativeInteger	required
···NodeCount	xs:positiveInteger	required
···IntervalCount	xs:positiveInteger	required
···ClosedPolylines	resqml:AbstractBooleanArray	required
···NodeCountPerPolyline	resqml:AbstractIntegerArray	required
···IntervalGridCells	resqml:IntervalGridCells	optional
·····CellCount	xs:positiveInteger	required
·····GridIndices	resqml:AbstractIntegerArray	required
·····CellIndices	resqml:AbstractIntegerArray	required
·····LocalFacePairPerCellIndices	resqml:AbstractIntegerArray	required
·····Grids	eml:DataObjectReference	1 or more

6.5.66 obj_StringTableLookup

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Value	resqml:StringLookup	1 or more
···Key	xs:integer	required
···Value	xs:string	required

6.5.67 obj_StructuralOrganizationInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
ContactInterpretation	resqml:AbstractContactInterpretationPart	0 or more
···ContactRelationship	resqml:ContactRelationship	required
···Index	xs:nonNegativeInteger	required
···PartOf	eml:DataObjectReference	optional
OrderingCriteria	resqml:OrderingCriteria	required
Faults	eml:DataObjectReference	0 or more
Horizons	resqml:HorizonInterpretationIndex	0 or more
···Index	xs:nonNegativeInteger	required
···StratigraphicRank	xs:nonNegativeInteger	optional
···Horizon	eml:DataObjectReference	required
Sides	eml:DataObjectReference	0 or more
TopFrontier	eml:DataObjectReference	0 or more
BottomFrontier	eml:DataObjectReference	0 or more

6.5.68 obj_SubRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
AdditionalGridTopology	resqml:AdditionalGridTopology	optional
···SplitEdges	resqml:SplitEdges	optional
·····Count	xs:positiveInteger	required
·····ParentEdgeIndices	resqml:AbstractIntegerArray	required
·····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···SplitNodes	resqml:SplitNodePatch	optional
·····PatchIndex	xs:nonNegativeInteger	required
·····Count	xs:positiveInteger	required
·····ParentNodeIndices	resqml:AbstractIntegerArray	required
·····CellsPerSplitNode	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitFaces	resqml:SplitFaces	optional
·····Count	xs:positiveInteger	required
·····ParentFaceIndices	resqml:AbstractIntegerArray	required
·····CellIndices	resqml:AbstractIntegerArray	required
·····SplitEdges	resqml:SplitEdges	optional

continues on next page

Table 5 – continued from previous page

····Count	xs:positiveInteger	required
····ParentEdgeIndices	resqml:AbstractIntegerArray	required
····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
···SplitColumnEdges	resqml:ColumnLayerSplitColumnEdges	optional
····Count	xs:positiveInteger	required
····ParentColumnEdgeIndices	resqml:AbstractIntegerArray	required
····ColumnPerSplitColumnEdge	resqml:AbstractIntegerArray	required
···UnstructuredColumnEdges	resqml:UnstructuredColumnEdges	optional
····Count	xs:positiveInteger	required
····PillarsPerColumnEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···SplitFaces	resqml:SplitFaces	optional
····Count	xs:positiveInteger	required
····ParentFaceIndices	resqml:AbstractIntegerArray	required
····CellIndices	resqml:AbstractIntegerArray	required
····SplitEdges	resqml:SplitEdges	optional
·····Count	xs:positiveInteger	required
·····ParentEdgeIndices	resqml:AbstractIntegerArray	required
·····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
···IjSplitColumnEdges	resqml:IjSplitColumnEdges	optional
····Count	xs:positiveInteger	required
····PillarsPerSplitColumnEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···UnstructuredSubnodeTopology	resqml:UnstructuredSubnodeTopology	optional
····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····ObjectIndices	resqml:AbstractIntegerArray	required
·····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	xs:positiveInteger	required
····Edges	resqml:Edges	optional
·····Count	xs:positiveInteger	required
·····NodesPerEdge	resqml:AbstractIntegerArray	required
····NodesPerCell	resqml:NodesPerCell	optional
·····NodesPerCell	resqml:ResqmlJaggedArray	required
····Elements	resqml:AbstractValueArray	required
····CumulativeLength	resqml:AbstractIntegerArray	required
···ColumnLayerSubnodeTopology	resqml:ColumnLayerSubnodeTopology	optional
····VariableSubnodes	resqml:VariableSubnodePatch	0 or more

continues on next page

Table 5 – continued from previous page

·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····ObjectIndices	resqml:AbstractIntegerArray	required
·····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	xs:positiveInteger	required
····ColumnSubnodes	resqml:ColumnSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	resqml:AbstractIntegerArray	required
SupportingRepresentation	eml:DataObjectReference	required
SubRepresentationPatch	resqml:SubRepresentationPatch	1 or more
···PatchIndex	xs:nonNegativeInteger	required
···Count	xs:positiveInteger	required
···ElementIndices	resqml:ElementIndices	1 to 2
····IndexableElement	resqml:IndexableElements	required
····Indices	resqml:AbstractIntegerArray	required

6.5.69 obj_TectonicBoundaryFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
TectonicBoundaryKind	resqml:TectonicBoundaryKind	required

6.5.70 obj_TimeSeries

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Time	resqml:Timestamp	1 or more
···DateTime	xs:dateTime	required
···YearOffset	xs:long	optional
TimeSeriesParentage	resqml:TimeSeriesParentage	optional
···HasOverlap	xs:boolean	required
···ParentTimeIndex	resqml:TimeIndex	required
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required

6.5.71 obj_TriangulatedSetRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
SurfaceRole	resqml:SurfaceRole	required
Boundaries	resqml:PatchBoundaries	0 or more
···InnerRing	eml:DataObjectReference	0 or more
···OuterRing	eml:DataObjectReference	optional
···ReferencedPatch	xs:nonNegativeInteger	required
TrianglePatch	resqml:TrianglePatch	1 or more
···PatchIndex	xs:nonNegativeInteger	required
···Count	xs:positiveInteger	required
···NodeCount	xs:nonNegativeInteger	required
···Triangles	resqml:AbstractIntegerArray	required
···Geometry	resqml:PointGeometry	required
·····TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
·····LocalCrs	eml:DataObjectReference	required
·····Points	resqml:AbstractPoint3dArray	required
·····SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required
···SplitEdgePatch	resqml:EdgePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····Count	xs:positiveInteger	required
·····SplitEdges	resqml:AbstractIntegerArray	optional

6.5.72 obj_TruncatedIjkGridRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
CellFluidPhaseUnits	resqml:CellFluidPhaseUnits	optional
···PhaseUnitIndices	resqml:AbstractIntegerArray	required
···FluidOrganization	eml:DataObjectReference	required
ParentWindow	resqml:AbstractParentWindow	optional
···CellOverlap	resqml:CellOverlap	optional
·····Count	xs:positiveInteger	required
·····ParentChildCellPairs	resqml:AbstractIntegerArray	required
·····OverlapVolume	resqml:OverlapVolume	optional
·····VolumeUom	eml:VolumeUom	required
·····OverlapVolumes	resqml:AbstractDoubleArray	required
CellStratigraphicUnits	resqml:CellStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
Nk	xs:positiveInteger	required

continues on next page

Table 6 – continued from previous page

TruncationCells	resqml:TruncationCellPatch	required
···PatchIndex	xs:nonNegativeInteger	required
···TruncationNodeCount	xs:positiveInteger	required
···TruncationFaceCount	xs:positiveInteger	required
···TruncationCellCount	xs:positiveInteger	required
···NodesPerTruncationFace	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···ParentCellIndices	resqml:AbstractIntegerArray	required
···LocalFacesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···TruncationFacesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···TruncationCellFaceIsRightHanded	resqml:AbstractBooleanArray	required
Ni	xs:positiveInteger	required
Nj	xs:positiveInteger	required
Geometry	resqml:IjkGridGeometry	required
···TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
···LocalCrs	eml:DataObjectReference	required
···Points	resqml:AbstractPoint3dArray	required
···SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required
···AdditionalGridPoints	resqml:AdditionalGridPoints	0 or more
·····RepresentationPatchIndex	xs:nonNegativeInteger	optional
·····Attachment	resqml:GridGeometryAttachment	required
·····Points	resqml:AbstractPoint3dArray	required
···KDirection	resqml:KDirection	required
···PillarGeometryIsDefined	resqml:AbstractBooleanArray	required
···PillarShape	resqml:PillarShape	required
···CellGeometryIsDefined	resqml:AbstractBooleanArray	optional
···NodeIsColocatedInKDirection	resqml:AbstractBooleanArray	optional
···NodeIsColocatedOnKEdge	resqml:AbstractBooleanArray	optional
···SubnodeTopology	resqml:ColumnLayerSubnodeTopology	optional
·····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····ObjectIndices	resqml:AbstractIntegerArray	required
·····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
·····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	xs:positiveInteger	required
·····ColumnSubnodes	resqml:ColumnSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required

continues on next page

Table 6 – continued from previous page

·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	resqml:AbstractIntegerArray	required
···SplitCoordinateLines	resqml:ColumnLayerSplitCoordinateLines	optional
·····Count	xs:positiveInteger	required
·····PillarIndices	resqml:AbstractIntegerArray	required
·····ColumnsPerSplitCoordinateLine	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitColumnEdges	resqml:ColumnLayerSplitColumnEdges	optional
·····Count	xs:positiveInteger	required
·····ParentColumnEdgeIndices	resqml:AbstractIntegerArray	required
·····ColumnPerSplitColumnEdge	resqml:AbstractIntegerArray	required
···SplitNodes	resqml:SplitNodePatch	optional
·····PatchIndex	xs:nonNegativeInteger	required
·····Count	xs:positiveInteger	required
·····ParentNodeIndices	resqml:AbstractIntegerArray	required
·····CellsPerSplitNode	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitFaces	resqml:SplitFaces	optional
·····Count	xs:positiveInteger	required
·····ParentFaceIndices	resqml:AbstractIntegerArray	required
·····CellIndices	resqml:AbstractIntegerArray	required
·····SplitEdges	resqml:SplitEdges	optional
·····Count	xs:positiveInteger	required
·····ParentEdgeIndices	resqml:AbstractIntegerArray	required
·····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···GridIsRighthanded	xs:boolean	required
···IjGaps	resqml:IjGaps	optional
·····SplitPillarCount	xs:positiveInteger	required
·····ParentPillarIndices	resqml:AbstractIntegerArray	required
·····ColumnsPerSplitPillar	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····IjSplitColumnEdges	resqml:IjSplitColumnEdges	optional
·····Count	xs:positiveInteger	required
·····PillarsPerSplitColumnEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required

6.5.73 obj_TruncatedUnstructuredColumnLayerGridRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
CellFluidPhaseUnits	resqml:CellFluidPhaseUnits	optional
···PhaseUnitIndices	resqml:AbstractIntegerArray	required
···FluidOrganization	eml:DataObjectReference	required
ParentWindow	resqml:AbstractParentWindow	optional
···CellOverlap	resqml:CellOverlap	optional
·····Count	xs:positiveInteger	required
·····ParentChildCellPairs	resqml:AbstractIntegerArray	required
·····OverlapVolume	resqml:OverlapVolume	optional
·····VolumeUom	eml:VolumeUom	required
·····OverlapVolumes	resqml:AbstractDoubleArray	required
CellStratigraphicUnits	resqml:CellStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
Nk	xs:positiveInteger	required
TruncationCells	resqml:TruncationCellPatch	required
···PatchIndex	xs:nonNegativeInteger	required
···TruncationNodeCount	xs:positiveInteger	required
···TruncationFaceCount	xs:positiveInteger	required
···TruncationCellCount	xs:positiveInteger	required
···NodesPerTruncationFace	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···ParentCellIndices	resqml:AbstractIntegerArray	required
···LocalFacesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···TruncationFacesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···TruncationCellFaceIsRightHanded	resqml:AbstractBooleanArray	required
ColumnCount	xs:positiveInteger	required
Geometry	resqml:UnstructuredColumnLayerGridGeometry	required
···TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
···LocalCrs	eml:DataObjectReference	required
···Points	resqml:AbstractPoint3dArray	required
···SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required
···AdditionalGridPoints	resqml:AdditionalGridPoints	0 or more
·····RepresentationPatchIndex	xs:nonNegativeInteger	optional
·····Attachment	resqml:GridGeometryAttachment	required
·····Points	resqml:AbstractPoint3dArray	required
···KDirection	resqml:KDirection	required

continues on next page

Table 7 – continued from previous page

···PillarGeometryIsDefined	resqml:AbstractBooleanArray	required
···PillarShape	resqml:PillarShape	required
···CellGeometryIsDefined	resqml:AbstractBooleanArray	optional
···NodeIsColocatedInKDirection	resqml:AbstractBooleanArray	optional
···NodeIsColocatedOnKEdge	resqml:AbstractBooleanArray	optional
···SubnodeTopology	resqml:ColumnLayerSubnodeTopology	optional
·····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····ObjectIndices	resqml:AbstractIntegerArray	required
·····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
·····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	xs:positiveInteger	required
·····ColumnSubnodes	resqml:ColumnSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	resqml:AbstractIntegerArray	required
···SplitCoordinateLines	resqml:ColumnLayerSplitCoordinateLines	optional
·····Count	xs:positiveInteger	required
·····PillarIndices	resqml:AbstractIntegerArray	required
·····ColumnsPerSplitCoordinateLine	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitColumnEdges	resqml:ColumnLayerSplitColumnEdges	optional
·····Count	xs:positiveInteger	required
·····ParentColumnEdgeIndices	resqml:AbstractIntegerArray	required
·····ColumnPerSplitColumnEdge	resqml:AbstractIntegerArray	required
···SplitNodes	resqml:SplitNodePatch	optional
·····PatchIndex	xs:nonNegativeInteger	required
·····Count	xs:positiveInteger	required
·····ParentNodeIndices	resqml:AbstractIntegerArray	required
·····CellsPerSplitNode	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitFaces	resqml:SplitFaces	optional
·····Count	xs:positiveInteger	required
·····ParentFaceIndices	resqml:AbstractIntegerArray	required
·····CellIndices	resqml:AbstractIntegerArray	required
·····SplitEdges	resqml:SplitEdges	optional
·····Count	xs:positiveInteger	required
·····ParentEdgeIndices	resqml:AbstractIntegerArray	required
·····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···ColumnShape	resqml:ColumnShape	required

continues on next page

Table 7 – continued from previous page

···PillarCount	xs:positiveInteger	required
···PillarsPerColumn	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···ColumnIsRightHanded	resqml:AbstractBooleanArray	required
···ColumnEdges	resqml:UnstructuredColumnEdges	optional
·····Count	xs:positiveInteger	required
·····PillarsPerColumnEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required

6.5.74 obj_UnstructuredColumnLayerGridRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
CellFluidPhaseUnits	resqml:CellFluidPhaseUnits	optional
···PhaseUnitIndices	resqml:AbstractIntegerArray	required
···FluidOrganization	eml:DataObjectReference	required
ParentWindow	resqml:AbstractParentWindow	optional
···CellOverlap	resqml:CellOverlap	optional
·····Count	xs:positiveInteger	required
·····ParentChildCellPairs	resqml:AbstractIntegerArray	required
·····OverlapVolume	resqml:OverlapVolume	optional
·····VolumeUom	eml:VolumeUom	required
·····OverlapVolumes	resqml:AbstractDoubleArray	required
CellStratigraphicUnits	resqml:CellStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
Nk	xs:positiveInteger	required
IntervalStratigraphicUnits	resqml:IntervalStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
ColumnCount	xs:positiveInteger	required
Geometry	resqml:UnstructuredColumnLayerGridGeometry	optional
···TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
···LocalCrs	eml:DataObjectReference	required
···Points	resqml:AbstractPoint3dArray	required
···SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required
···AdditionalGridPoints	resqml:AdditionalGridPoints	0 or more
·····RepresentationPatchIndex	xs:nonNegativeInteger	optional
·····Attachment	resqml:GridGeometryAttachment	required
·····Points	resqml:AbstractPoint3dArray	required
···KDirection	resqml:KDirection	required

continues on next page

Table 8 – continued from previous page

···PillarGeometryIsDefined	resqml:AbstractBooleanArray	required
···PillarShape	resqml:PillarShape	required
···CellGeometryIsDefined	resqml:AbstractBooleanArray	optional
···NodeIsColocatedInKDirection	resqml:AbstractBooleanArray	optional
···NodeIsColocatedOnKEdge	resqml:AbstractBooleanArray	optional
···SubnodeTopology	resqml:ColumnLayerSubnodeTopology	optional
·····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····ObjectIndices	resqml:AbstractIntegerArray	required
·····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
·····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	xs:positiveInteger	required
·····ColumnSubnodes	resqml:ColumnSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	resqml:AbstractIntegerArray	required
···SplitCoordinateLines	resqml:ColumnLayerSplitCoordinateLines	optional
·····Count	xs:positiveInteger	required
·····PillarIndices	resqml:AbstractIntegerArray	required
·····ColumnsPerSplitCoordinateLine	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitColumnEdges	resqml:ColumnLayerSplitColumnEdges	optional
·····Count	xs:positiveInteger	required
·····ParentColumnEdgeIndices	resqml:AbstractIntegerArray	required
·····ColumnPerSplitColumnEdge	resqml:AbstractIntegerArray	required
···SplitNodes	resqml:SplitNodePatch	optional
·····PatchIndex	xs:nonNegativeInteger	required
·····Count	xs:positiveInteger	required
·····ParentNodeIndices	resqml:AbstractIntegerArray	required
·····CellsPerSplitNode	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
·····SplitFaces	resqml:SplitFaces	optional
·····Count	xs:positiveInteger	required
·····ParentFaceIndices	resqml:AbstractIntegerArray	required
·····CellIndices	resqml:AbstractIntegerArray	required
·····SplitEdges	resqml:SplitEdges	optional
·····Count	xs:positiveInteger	required
·····ParentEdgeIndices	resqml:AbstractIntegerArray	required
·····FacesPerSplitEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···ColumnShape	resqml:ColumnShape	required

continues on next page

Table 8 – continued from previous page

···PillarCount	xs:positiveInteger	required
···PillarsPerColumn	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···ColumnIsRightHanded	resqml:AbstractBooleanArray	required
···ColumnEdges	resqml:UnstructuredColumnEdges	optional
·····Count	xs:positiveInteger	required
·····PillarsPerColumnEdge	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required

6.5.75 obj_UnstructuredGridRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
CellFluidPhaseUnits	resqml:CellFluidPhaseUnits	optional
···PhaseUnitIndices	resqml:AbstractIntegerArray	required
···FluidOrganization	eml:DataObjectReference	required
ParentWindow	resqml:AbstractParentWindow	optional
···CellOverlap	resqml:CellOverlap	optional
·····Count	xs:positiveInteger	required
·····ParentChildCellPairs	resqml:AbstractIntegerArray	required
·····OverlapVolume	resqml:OverlapVolume	optional
·····VolumeUom	eml:VolumeUom	required
·····OverlapVolumes	resqml:AbstractDoubleArray	required
CellStratigraphicUnits	resqml:CellStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
CellCount	xs:positiveInteger	required
Geometry	resqml:UnstructuredGridGeometry	optional
···TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
···LocalCrs	eml:DataObjectReference	required
···Points	resqml:AbstractPoint3dArray	required
···SeismicCoordinates	resqml:AbstractSeismicCoordinates	optional
·····SeismicSupport	eml:DataObjectReference	required
···AdditionalGridPoints	resqml:AdditionalGridPoints	0 or more
·····RepresentationPatchIndex	xs:nonNegativeInteger	optional
·····Attachment	resqml:GridGeometryAttachment	required
·····Points	resqml:AbstractPoint3dArray	required
···CellShape	resqml:CellShape	required
···NodeCount	xs:positiveInteger	required
···FaceCount	xs:positiveInteger	required
···NodesPerFace	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required

continues on next page

Table 9 – continued from previous page

·····CumulativeLength	resqml:AbstractIntegerArray	required
···FacesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required
···CellFaceIsRightHanded	resqml:AbstractBooleanArray	required
···HingeNodeFaces	resqml:UnstructuredGridHingeNodeFaces	optional
·····Count	xs:positiveInteger	required
·····FaceIndices	resqml:AbstractIntegerArray	required
···SubnodeTopology	resqml:UnstructuredSubnodeTopology	optional
·····VariableSubnodes	resqml:VariableSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····ObjectIndices	resqml:AbstractIntegerArray	required
·····SubnodeCountPerSelectedObject	resqml:AbstractIntegerArray	required
·····UniformSubnodes	resqml:UniformSubnodePatch	0 or more
·····PatchIndex	xs:nonNegativeInteger	required
·····SubnodeNodeObject	resqml:SubnodeNodeObject	required
·····NodeWeightsPerSubnode	resqml:AbstractValueArray	required
·····SubnodeCountPerObject	xs:positiveInteger	required
·····Edges	resqml:Edges	optional
·····Count	xs:positiveInteger	required
·····NodesPerEdge	resqml:AbstractIntegerArray	required
·····NodesPerCell	resqml:NodesPerCell	optional
·····NodesPerCell	resqml:ResqmlJaggedArray	required
·····Elements	resqml:AbstractValueArray	required
·····CumulativeLength	resqml:AbstractIntegerArray	required

6.5.76 obj_WellboreFeature

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
WitsmlWellbore	resqml:WitsmlWellboreReference	optional
···WitsmlWell	eml:DataObjectReference	required
···WitsmlWellbore	eml:DataObjectReference	required

6.5.77 obj_WellboreFrameRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
NodeCount	xs:positiveInteger	required
NodeMd	resqml:AbstractDoubleArray	required
WitsmlLogReference	eml:DataObjectReference	optional
IntervalStratigraphiUnits	resqml:IntervalStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
CellFluidPhaseUnits	resqml:CellFluidPhaseUnits	optional
···PhaseUnitIndices	resqml:AbstractIntegerArray	required
···FluidOrganization	eml:DataObjectReference	required
Trajectory	eml:DataObjectReference	required

6.5.78 obj_WellboreInterpretation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
Domain	resqml:Domain	required
InterpretedFeature	eml:DataObjectReference	required
HasOccuredDuring	resqml:TimeInterval	optional
···ChronoBottom	eml:DataObjectReference	required
···ChronoTop	eml:DataObjectReference	required
IsDrilled	xs:boolean	required

6.5.79 obj_WellboreMarkerFrameRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
NodeCount	xs:positiveInteger	required
NodeMd	resqml:AbstractDoubleArray	required
WitsmlLogReference	eml:DataObjectReference	optional
IntervalStratigraphiUnits	resqml:IntervalStratigraphicUnits	optional
···UnitIndices	resqml:AbstractIntegerArray	required
···StratigraphicOrganization	eml:DataObjectReference	required
CellFluidPhaseUnits	resqml:CellFluidPhaseUnits	optional
···PhaseUnitIndices	resqml:AbstractIntegerArray	required
···FluidOrganization	eml:DataObjectReference	required
Trajectory	eml:DataObjectReference	required
WellboreMarker	resqml:WellboreMarker	1 or more
···ExtraMetadata	resqml:NameValuePair	0 or more
·····Name	xs:string	required
·····Value	xs:string	required
···FluidContact	resqml:FluidContact	optional
···FluidMarker	resqml:FluidMarker	optional
···GeologicBoundaryKind	resqml:GeologicBoundaryKind	optional
···WitsmlFormationMarker	eml:DataObjectReference	optional
···Interpretation	eml:DataObjectReference	optional

6.5.80 obj_WellboreTrajectoryRepresentation

ExtraMetadata	resqml:NameValuePair	0 or more
···Name	xs:string	required
···Value	xs:string	required
RepresentedInterpretation	eml:DataObjectReference	optional
StartMd	xs:double	required
FinishMd	xs:double	required
MdUom	eml:LengthUom	required
MdDomain	resqml:MdDomain	optional
WitsmlTrajectory	eml:DataObjectReference	optional
Geometry	resqml:AbstractParametricLineGeometry	optional
···TimeIndex	resqml:TimeIndex	optional
·····Index	xs:nonNegativeInteger	required
·····TimeSeries	eml:DataObjectReference	required
···LocalCrs	eml:DataObjectReference	required
MdDatum	eml:DataObjectReference	required
DeviationSurvey	eml:DataObjectReference	optional
ParentIntersection	resqml:WellboreTrajectoryParentIntersection	optional
···KickoffMd	xs:double	required
···ParentMd	xs:double	required
···ParentTrajectory	eml:DataObjectReference	required

6.6 Cellular Grid Basics

This tutorial page introduces some basic concepts and methods for working with cellular grids in resqpy. Cellular grids are the most complex of the RESQML object classes and further material will be presented in later tutorials. This page does not cover RESQML 2D grids, for which resqpy uses the term mesh.

6.6.1 RESQML cellular grid classes

By cellular grid, we mean a data structure that divides up a finite 3D space into cells, each of which has a defined geometry. The RESQML standard differentiates between 3 primary classes of cellular grid, depending on how the cells are indexed:

- IjkGridRepresentation, where each cell is indexed by three integers – the most commonly used form
- UnstructuredColumnLayerGridRepresentation, with cells indexed by two integers – where the column pattern in plan view is unstructured, perhaps constructed on a perpendicular bisector (pebi) basis, yet there is still a clearly defined layering
- UnstructuredGridRepresentation, with cells indexed by a single integer – where there are no constraints on the shape or arrangement of cells

Two more classes of grid are similar to the first two above but allow for some of the cells to be further split (typically by a fault plane) and therefore have locally varying number of faces and indexing:

- TruncatedIjkGridRepresentation
- TruncatedUnstructuredColumnLayerGridRepresentation

One further RESQML class is rather ill-defined and caters for research grids:

- GpGridRepresentation – the Gp stands for general purpose

Of these six RESQML classes only obj_IjkGridRepresentation and obj_UnstructuredGridRepresentation have currently been implemented in resqpy. The equivalent resqpy classes are Grid and UnstructuredGrid, though there are some derived classes. RegularGrid, can be used to construct simple block IJK grids and there are a handful of classes derived from UnstructuredGrid for constrained cell shapes.

6.6.2 IJK grid variants

The RESQML IjkGridRepresentation class includes optional attributes that allow for various grid features:

- Explicitly defined (irregular) or regular geometry
- Radial or cartesian grid geometry
- Fully defined or with missing geometry
- Unfaulted or faulted
- With or without gaps between layers
- With or without gaps between I or J slices

These are discussed in the following paragraphs.

Explicitly defined or regular geometry

The RESQML schema definition uses abstract classes to allow some conceptual objects to be represented in different ways. A grid geometry may be defined explicitly, with the xyz location of each cell corner point held in the dataset. Alternatively, if the cells have a regular cuboid shape, the size and number of these cells can be succinctly stored without saving all the corner point information. (The presentation here, of two possibilities, is actually a simplification of multiple options that are available in the standard.)

Note that the implication of the IJK grid concept is that cells, for the most part, share faces with their logical neighbours in the three logical dimensions of I, J & K even in the case of an explicitly defined geometry. For this reason, the storage of corner point locations is such that there is no duplication of data where a corner point is shared between multiple cells. When faces and corner points are not shared, for example due to faulting, data structures in the RESQML schema can represent this and these situations are discussed in later paragraphs.

The resqpy Grid class handles explicitly defined grid geometries. The RegularGrid class can be used to construct unfaulted grids with regular geometry, to be stored either in a compact form or expanded to an explicit corner point representation for persistent storage.

Radial or cartesian geometry

Given an explicit representation of grid geometry, the actual pattern of cells in physical xyz space is very flexible, subject to the condition that the geometry is predominantly continuous, ie. with cells that are neighbours in IJK space mostly sharing faces. Therefore to represent a radial grid, the only real difference in the RESQML data is the inclusion of an optional boolean flag, RadialGridIsComplete, which indicates that the last slice in the J dimension should be considered to be a neighbour of the first slice. A further data structure – radial origin polyline – is required for regular radial grids. With such radial grids, the I index varies radially away from the centre of the grid, and the J index varies in the angular (theta) direction. Of course, in the case of an explicit grid, the corner point locations will need to have been generated appropriately.

The resqpy Grid class has been written with 'cartesian' (ie. non-radial) grids in mind. In particular, methods working with the geometry of individual cells assume them to have a hexahedral geometry, which is not the intention for cells in radial grids. Therefore at present radial grids are not supported by resqpy.

Fully defined or with missing geometry

For explicitly defined grid geometries, the RESQML standard allows for some parts of the geometry to be flagged as missing (not defined). This can be indicated with a boolean array holding a value for each pillar (also known as a coordinate line) and/or another array holding a value for each cell. It is recommended that the numpy NaN (not a number) value is used in the corner point array where the geometry is missing.

The resqpy Grid class handles these missing geometry options. However, some of the higher level resqpy functionality, such as finding the intersection of a well trajectory with a grid, require the geometry to be complete. To facilitate this, one of the Grid methods – set_geometry_is_defined() – can optionally generate geometry where it is missing (and mark such cells as inactive).

Unfaulted or faulted

A grid may be unfaulted, in which case all cells share their faces with any logically neighbouring cells (with the exception of gaps, discussed in the following paragraphs). In this case the corner point data is fully shared between cells and the array has the shape (NK + 1, NJ + 1, NI + 1, 3) with the final axis covering the x,y,z values for each point.

If, however, a grid involves some faults with throw, then some of the pillars are split, with different sets of xyz data for the two sides of the fault. (And where two faults cross, a single pillar may be split four ways.) In the RESQML data, the extra pillar data is represented by flattening the middle two dimensions of the corner point data into a single 'pillar' axis, and then extending that axis with the required number of extra sets of pillar data. This means the corner point array has the shape (NK + 1, (NJ + 1) * (NI + 1) + NE, 3) where NE is the number of extra sets of pillar data due to splitting. Some extra integer arrays are also required to identify which of the original pillars are split and which columns of cells the extra pillar data pertain to.

Both the RESQML IjkGridRepresentation class and the resqpy Grid class contain a simple boolean flag indicating whether any split pillars are present or not. In general the resqpy code fully supports both faulted and unfaulted grids.

The resqpy Grid class also contains a method –unsplit_points_ref() – for returning an unsplit version of the corner point array. That method is rather simplistic and the higher level derived_model module contains functions which can modify the throw on faults in more complex ways.

With or without gaps between layers

The RESQML usage guide recommends against building cellular grids with unrepresented voids between cells. However, the schema definition does allow for this. In the case of an IJK grid, a gap can exist between layers and this is referred to as a 'K gap'. When K gaps are present, an extra boolean array indicates which layers in the model have a K gap immediately 'afterwards' (which usually means below). The first axis of the corner point data is enlarged to provide two slices of points data between layers where there is a gap (instead of the normal one, shared, slice).

The resqpy code can generally handle grids with K gaps.

With or without gaps between I or J slices

As with K gaps, the RESQML standard also allows for gaps between I or J slices of cells. However, the resqpy code does not support this. (Though the same geometry can be represented with split pillars as there is no requirement that the split pillar data lie on a single coordinate line in space.)

6.6.3 The resqpy Grid class

The resqpy grid module contains the class Grid, which handles RESQML IjkGridRepresentation objects. A Grid object has several attributes (which calling code can refer to directly) and methods, only some of which are introduced here.

Basic Grid attributes The following are just a few of the attributes which calling code is likely to access directly.

- model: the 'parent' model.Model object
- uuid
- root: the xml root node
- extent_kji: a triplet of integers containing the size of the grid (nk, nj, ni)
- ni, nj, nk: separate integer attributes, duplicating the information in extent_kji for convenience
- crs_uuid
- crs_root: the xml root node of the coordinate reference system used by the grid

- inactive: a numpy boolean array of shape extent_kji, indicating which cells are inactive
- property_collection: a property.PropertyCollection object holding the properties associated with the grid
- has_split_coordinate_lines: a boolean indicating whether the grid has any split pillars (ie. is faulted)

6.6.4 Basic Grid methods

Of the many methods provided by the Grid class, the following are the most commonly used. Refer to the docstrings for more information, including argument lists.

- cell_count(): returns the number of cells in the grid, optionally only counting those with geometry, or not pinched out, or active
- natural_cell_index(), natural_cell_indices(): converts cell index from triple index form (k, j, i) to single integer (for flattened array)
- denaturalized_cell_index(), denaturalized_cell_indices(): the opposite of the methods above
- cell_geometry_is_defined(): returns boolean indicating whether a cell has geometry
- pillar_geometry_is_defined(): returns boolean indicating whether a pillar has any geometry
- geometry_defined_for_all_cells(): returns single boolean
- geometry_defined_for_all_pillars(): returns single boolean
- cache_all_geometry_arrays(): ensures all the grid's geometry arrays are loaded as attributes
- create_column_pillar_mapping(): returns a numpy int array of shape (nj, ni, 2, 2) with pillar index for each corner of each column
- points_ref(): returns (and caches) the xyz points array, by default as a masked array based on NaN values
- xyz_box(): returns a numpy float array of shape (2, 3) holding the min and max xyz values
- split_horizon_points(): returns a numpy float array of shape (nj, ni, 2, 2, 3) being all corner points for a horizon (layer interface)
- split_x_section_points(): similar to above for a cross section (I or J interface)
- coordinate_line_end_points(): returns a numpy float array of shape (nj+1, ni+1, 2, 3) holding xyz points defining straight pillar lines
- z_corner_point_depths(): returns a numpy float array of shape (nk, nj, ni, 2, 2, 2) holding depth (z) values for cell corner points
- corner_points(): returns a numpy float array of shape (nk, nj, ni, 2, 2, 2, 3) holding the fully expanded corner points of each cell
- centre_point(): returns a numpy float array of shape (nk, nj, ni, 3) holding the centre point (mean of 8 corners) of each cell
- thickness(): returns a numpy float array of shape (nk, nj, ni) holding the thickness of each cell
- volume(): returns a numpy float array of shape (nk, nj, ni) holding the volume of each cell
- pinched_out(): returns a numpy boolean array of shape (nk, nj, ni) indicating which cells are completely pinched out
- interpolated_point(): returns the xyz location of a tri-linear interpolation of a point in a unit cube when mapped onto a cell
- face_centre(): returns the xyz location of the centre of one face of a cell

- interface_vector(): for one of the IJK axes, returns the vector from the centre of the negative face to the centre of the positive for a cell
- z_inc_down(): convenience method returning the boolean flag from the crs, indicating whether z is increasing downwards
- xy_units(): convenience method returning the units of measure of x & y, from the crs
- z_units(): convenience method returning the units of measure of z, from the crs
- off_handed(): returns True if the handedness of the IJK axes differs from that of the xyz axes, otherwise False
- find_cell_for_point_xy(): searches top of grid in 2D to find column containing a given xy point

A couple more methods are needed when writing a Grid object:

- write_hdf5()
- create_xml()

There are several other methods in the Grid class, and many of those above can be used in more than one way. The olio.grid_functions module contains some higher level functions for specialist grid operations and the derived_model module contains many functions for modifying grid geometries.

6.6.5 Reading a Grid object

In this tutorial the examples refer to the S-bend dataset.

First open a Model object in the usual way:

```
import resqpy.model as rq
import resqpy.grid as grr
model = rq.Model('s_bend.epc')
```

If your model is known to have only one grid object, or one grid titled 'ROOT', the model class convenience function grid() can be used:

```
grid = model.grid()
```

In the more general case, you will need to identify the desired RESQML object amongst potentially many grids. If the citation title for the desired grid is known and unique, the same Model method can be used, for example:

```
grid = model.grid(title = 'FAULTED GRID')
```

Alternatively, the initialiser for the Grid class can be called directly with something like:

```
grid_uuid = model.uuid(obj_type = 'IjkGridRepresentation', multiple_handling = 'newest')
grid = grr.Grid(model, uuid = grid_uuid)
```

6.6.6 Working with Regular Grids

The resqpy RegularGrid class inherits from the Grid class and can be used to create an unfaulted regular block grid. Such a regular grid can either be treated as a full grid or stored in a compact form and re-opened as a RegularGrid object. Note that opening an existing RegularGrid object will only work if the object was created using resqpy, as it relies on an item of extra metadata. Grids from other sources should be read using the Grid class directly.

To create a new regular grid where the IJK axes align with the xyz axes, use the following form:

```
new_grid = grr.RegularGrid(model, extent_kji = (10, 20, 25), dxyz = (100.0, 125.0, 10.0),
                           crs_uuid = model.uuid(obj_type = 'Local3dDepthCrs'), title =
↪'BLOCK GRID')
```

If you intend to treat the new grid as a standard grid, then use the optional *set_points_cached* argument. This causes an explicit geometry to be generated for the grid:

```
new_grid = grr.RegularGrid(model, extent_kji = (10, 20, 25), dxyz = (100.0, 125.0, 10.0),
                           crs_uuid = model.uuid(obj_type = 'LocalDepth3dCrs'), title =
↪'BLOCK GRID',
                           set_points_cached = True)
```

The same effect can be achieved after instantiation by calling the *make_regular_points_cached()* method.

The RegularGrid class includes redefinitions of several of the Grid methods, such as *volume()*, where the regularity leads to significant speed increases compared with the general case code.

When converting a resqpy RegularGrid object to a RESQ ML object, it can be stored with or without an explicit grid. If storing without an explicit grid, skip the usual call to write_hdf5(), and use the default argument settings for the create_xml() method:

```
new_grid.create_xml()
```

If, on the other hand, you want to treat the new grid as a standard Grid object, make sure that the geometry has been set up (if in doubt call the *make_regular_points_cached()* method), then call the write_hdf5() method as usual, and modify some of the arguments to create_xml():

```
new_grid.make_regular_points_cached()
new_grid.write_hdf5()
new_grid.create_xml(write_geometry = True, add_cell_length_properties = False)
```

If you want the constant cell length property arrays to be generated anyway, leave the *add_cell_length_properties* argument at its default value of True.

6.6.7 Unstructured Grids in resqpy

The unstructured grid classes are not as fully developed as the Grid and RegularGrid classes. To work with unstructured grids, include the unstructured module, for example:

```
import resqpy.unstructured as rug
```

To read an existing RESQML unstructured grid object with a known uuid, use the familiar form:

```
u_grid = rug.UnstructuredGrid(model, uuid = u_grid_uuid)
```

Alternatively, there is a convenience function in the grid module which will instantiate a suitable resqpy object for any of the supported types of grid (including IJK grids):

```
u_grid = grr.any_grid(model, uuid = u_grid_uuid)
```

Unstructured grids can be used with or without a geometry. To create a new unstructured grid without a geometry, use code along these lines:

```
new_grid = rug.UnstructuredGrid(model, find_properties = False, geometry_required =␣
↪False, title = 'abstract grid')
new_grid.set_cell_count(1500)
```

Typically, such a grid without a geometry exists primarily as a supporting representation for properties. These properties can be created and added in the same way as for IJK grid properties - see the Grid Properties tutorial. The only significant difference is the shape of the property arrays (3D for Grid, 1D for UnstructuredGrid).

If no geometry is present, the write_hdf5() method can typically be skipped (though it can still be used to process a new property collection and/or write an active cell boolean array). The xml for a new unstructured grid is created in the familiar way:

```
new_grid.create_xml(write_active = False, write_geometry = False)
```

6.6.8 Unstructured Grids for Specialised Cell Shapes

RESQML unstructured grids with a geometry include a cell shape attribute. In the general case, this is set to 'polyhedral'. However, if all cells in a grid have a similar shape, then this attribute can be set to one of 'tetrahedral', 'pyramidal', 'prism', or 'hexahedral'. The pyramidal setting implies all cells have a quadrilateral based pyramid shape, ie. one face with four edges and four triangular faces. The prism setting implies all cells are triangular prisms with two non-adjacent triangular faces and three quadrilateral faces. The hexahedral setting implies all cells have 6 quadrilateral faces (though degeneracy is allowed).

Each of these more specialised cell shapes has a corresponding resqpy class, inheriting from UnstructuredGrid. The classes are: TetraGrid, PyramidGrid, PrismGrid and HexaGrid. The intention is to include optimised methods for these classes in future.

The HexaGrid class includes a class method to create an unstructured grid from an existing unsplit IJK grid (with no K gaps): from_unsplit_grid().

6.7 Grid Properties

This tutorial is about working with RESQML grid property arrays. However, much of what is presented here is also applicable to property data attached to other classes of objects, for example well logs. Some of the notes refer to the Nexus simulator as resqpy includes import and export functions for working with Nexus. Similar considerations would apply to other simulators, though the import and export functions would need to be developed separately.

You should edit the file paths in the examples to point to your own resqml dataset.

6.7.1 Quick start for getting at property arrays in a RESQML dataset

There are different routes to tracking down a property array, in these examples we go via the host grid object.

The first step is always to open the dataset as a resqpy Model object:

```
epc_path = '/sd/sdaq_2.epc'  # an existing RESQML dataset
import resqpy.model as rq
model = rq.Model(epc_path)
```

If the dataset is for a single realisation, or has a shared grid used by all realisations, we can get a Grid object for the ROOT grid with:

```
grid = model.grid()
```

See the previous tutorial for information on finding the right grid in a multi-grid dataset.

A Grid object comes with a collection of properties as an attribute. It is a good idea to set a short variable name to this collection, as we are going to work with it intensively. Here we use pc but it has nothing to do with capillary pressure, so if you find that confusing, pick another name!

```
pc = grid.property_collection
```

The property collection is a resqpy object of class property.PropertyCollection which has many methods available. Each individual property array is referred to as a part (and also exists in the RESQML dataset as an individual object in its own right, though in these examples we always access the properties via the collection for a given grid).

The simplest PropertyCollection method simply returns the number of parts (arrays) in the collection:

```
pc.number_of_parts()
```

If we know the title of the property array we are interested in, and if the title is unique, we can get at it as a numpy array with, eg.:

```
pore_volume_array = pc.single_array_ref(citation_title = 'PVR')
```

Where a RESQML dataset has been constructed from Nexus data using the resqpy import functions, the citation title for grid properties will have been set to the Nexus keyword used in the vdb dataset or ascii files.

Note that arrays are not actually loaded into memory until they are requested with methods such as the one shown above.

Grid property arrays with one value per cell have the shape (nk, nj, ni) – **note the order of the indexing: [k, j, i]**. Also note that the indices for these numpy arrays begin at zero (the Python way), rather than 1 which is used by simulators such as Nexus (and is the default in Fortran). So, for an index of a cell in simulator format: (sim_i, sim_j, sim_k), the property value for that cell is found with:

```
pore_volume_array[sim_k - 1, sim_j - 1, sim_i - 1]
```

This arrangement means that the actual array of data is laid out on disc or in memory in exactly the same way in the two systems.

If instead of the actual array, we want to get the part name (which is often needed as an argument to other methods in the property collection class), we can use:

```
pore_volume_part = pc.singleton(citation_title = 'PVR')
```

Each property array is a high level object in its own right, and the part name is the same as that used by the Model class when managing the high level objects.

6.7.2 Using RESQML property kinds and facets

In the examples above, we are using the citation title to uniquely identify a property array. That can work if the source of the dataset is known in advance, so that the values and uniqueness of citation titles is ensured. However, to write code that will work with RESQML data that has come from other sources, it is better to use the *property kind* to find the array of interest. The resqpy Nexus vdb import code also sets the property kind, so the following should work regardless of the source of the RESQML data:

```
pore_volume_array = pc.single_array_ref(property_kind = 'pore volume')
```

There is a fixed list of standard property kinds, defined in the RESQML standard, though extra 'local property kinds' can be defined when needed. The standard property kinds that are most often used can be found as:

```
rqp.supported_property_kind_list
```

which evaluates to:

```
['code', 'index', 'depth', 'rock volume', 'pore volume', 'volume',
 'thickness', 'length', 'cell length', 'net to gross ratio', 'porosity',
 'permeability thickness', 'permeability length', 'permeability rock',
 'rock permeability', 'fluid volume', 'transmissibility', 'pressure',
 'saturation', 'solution gas-oil ratio', 'vapor oil-gas ratio',
 'property multiplier', 'thermodynamic temperature',
 'continuous', 'discrete', 'categorical']
```

That list is a small subset of the standard resqml property kinds – the subset which resqpy has some 'understanding' of. For the full list, see the definition of ResqmlPropertyKind in the RESQML schema definition file property.xsd, or find the same list in json format in the resqpy repository file: resqml/olio/data/properties.json. Using property kinds that are not in the supported_property_kind_list should usually be okay.

The following method returns a list of the distinct property kinds found within the collection:

```
property_kinds_present = pc.property_kind_list()
```

Some of the property kinds may have an associated directional indication, which is stored as a property *facet*, with a facet type of 'direction'. So to get at PERMZ using the property kind, we would need:

```
vertical_perm_array = pc.single_array_ref(property_kind = 'permeability rock', facet_
↪type = 'direction', facet = 'K')
```

or facet = 'I' or facet = 'J' for 'horizontal' permeability arrays.

Here are the facet types and facet values currently used by resqpy:

- facet_type = 'direction': facet = 'I', 'J', 'K', 'IJ', or 'IJK', used for 'permeability rock', 'transmissibility', 'property multiplier' for transmissibility
- facet_type = 'netgross': facet = 'net' or 'gross', sometimes used for property kinds 'rock volume' and 'thickness'
- facet_type = 'what': facet = 'oil', 'water' or 'gas', used for saturations

The exact use of facets is not really pinned down in the RESQML standard, so we might choose to work with the citation titles in some situations.

The RESQML standard allows for a property object to have any number of facets. However, the resqpy PropertyCollection class currently handles at most one facet per property.

6.7.3 Identifying basic static properties

The PropertyCollection class includes a convenience method for identifying 5 basic static properties: net to gross ratio, porosity, and 3 permeabilities (I, J & K). The following method returns a tuple of 5 part names:

```
ntg_part, porosity_part, perm_i_part, perm_j_part, perm_k_part = pc.basic_static_
↪property_parts(share_perm_parts = True)
```

Given a part name for a property, the numpy array can be accessed with:

```
ntg_array = pc.cached_part_array_ref(ntg_part)
```

The share_perm_parts argument allows the same part to be returned for more than one of the three permeability keys. So, for example, if only one permeability rock array is found and it doesn't have any direction facet information, then it will be returned for all three permeability dictionary entries. The array caching mechanism means that the actual array data will not be duplicated, even if 3 array variables are set up.

There is a similar method which returns the UUIDs of the same 5 basic static properties:

```
ntg_uuid, porosity_uuid, perm_i_uuid, perm_j_uuid, perm_k_uuid = pc.basic_static_
↪property_uuids(share_perm_parts = True)
```

6.7.4 Continuous, discrete and categorical properties

The RESQML standard distinguishes between three classes of property, depending on the type of an individual datum:

- **continuous**: for real (floating point) data
- **categorical**: for integer data where the set of possible values is limited and a value can be used as an index into a lookup table (e.g. facies)
- **discrete**: for other integer or boolean data

Both categorical and discrete make use of a numpy array of integers. In terms of the data structures, the difference is that a categorical property also has a reference to a string lookup table. The following example shows how to get at the lookup table. (Note that at present the resqpy code for converting from Nexus vdb to RESQML does not create any lookup tables, so the datasets only contain continuous and discrete properties, not categorical.)

```
facies_part = pc.singleton(citation_title = 'FACIES')
lookup_table = pc.string_lookup_for_part(facies_part)
```

The lookup table is an object of resqpy class StringLookup (equivalent to RESQML class StringTableLookup). It maps integer values to strings. Given an integer, the string can be looked up with:

```
facies_name = lookup_table.get_string(2)
```

To go in the opposite direction, i.e. discover the integer value for a given string, use:

```
facies_int_for_mouthbar = lookup_table.get_index_for_string('MOUTHBAR')
```

If you are not sure what class a property is, the property collection has some methods to help:

```
pc.continuous_for_part(facies_part)  # returns True if the property is continuous, False␣
↪for categorical or discrete
pc.part_is_categorical(facies_part)  # returns True it the property is categorical,␣
↪False otherwise
```

Note that the resqpy code tends to treat categorical as a special case of discrete, so some methods have a boolean argument to distinguish between continuous and discrete – in which case the argument should be set to the value for discrete data when handling a categorical property.

6.7.5 Units of measure

The RESQML standard includes a comprehensive handling of units of measure – uom. Any continuous property must have an associated uom which can be accessed, for example, with:

```
pv_part = pc.singleton(property_kind = 'pore volume')
pv_uom = pc.uom_for_part(pv_part)  # for volumes, the uom will be 'm3' or 'ft3' for our␣
↪datasets
```

The RESQML standard includes a full (very long) list of allowable units. Here are a few of the common ones we might be using:

- length: 'm', 'ft'
- area: 'm2', 'ft2'
- volume: 'm3', 'ft3', 'bbl'
- volume ratios: 'm3/m3', 'ft3/ft3', 'ft3/bbl', '1000 ft3/bbl' (the first two are used for net to gross ratio, porosity, saturation)
- volume rate: 'm3/d', 'bbl/d', 'ft3/d', '1000 ft3/d'
- permeability: 'mD'
- pressure: 'kPa', 'bar', 'psi'
- unitless: 'Euc' (but preferable to use ratio units where they exist, for dimensionless ratios such as the volume ratios above)

The RESQML units definition is shared with the other Energistic standards: PRODML & WITSML. It is very thorough and well thought out. Here we only touch on it in the most minimal way. The full list of units of measure is to be found in the RESQML common schema definition file QuantityClass.xsd, and is also available in json format in the resqpy repository file: resqml/olio/data/properties.json. The resqpy *weights_and_measures* module also has functions for retrieving such information.

Discrete and categorical properties do not have a unit of measure.

Resqpy includes support for the full Energistics uom system, including a general unit conversion capability: See *The Units of Measure system* tutorial.

6.7.6 Null values and masked arrays

RESQML continuous properties use the special floating point value Not-a-Number, or NaN (np.NaN), as the null value. This is convenient as the numpy array operations can generally handle the null values without much extra coding effort. For discrete (including categorical) properties, a null value can be explicitly identified in the metadata. It is common to use -1 as the null value unless this is a valid value for the property.

To discover the null value for a discrete (or categorical) part, use something like:

```
irock_part = pc.singleton(title = 'IROCK')
irock_null_value = pc.null_value_for_part(irock_part)
```

The null_value_for_part() method will return an integer if a null value has been defined (or None if a null value has not been defined in the metadata) for a discrete property, or np.NaN if the part is a continuous property.

The property collection methods which return an array of property data, such as single_array_ref(), return a simple numpy array by default. However, there is the option to return a numpy masked array instead. Masked arrays contain not only the data but also a boolean mask indicating which elements to exclude. When a masked array is requested, the resqpy code sets the mask to be the inactive cell mask. There is also an option to mask out elements containing the null value. Numpy operations working with a masked array as an operand will also return a masked array. Furthermore, numpy operations such as sum, mean etc. will ignore masked out values.

To get a masked version of a property array, use one of the following forms:

```
depth_masked_array = pc.single_array_ref(property_kind = 'depth', masked = True)  #␣
↪exludes inactive cells
mean_active_depth = np.mean(depth_masked_array)

# following also excludes null value cells
facies_masked_array = pc.single_array_ref(title = 'FACIES', masked = True, exclude_null␣
↪= True)
```

The cached_part_array_ref() method also has the same optional arguments.

6.7.7 Universally unique identifiers

From the earlier discussion, it is clear that sometimes we might struggle to identify a particular property object. To help with this problem, RESQML makes use of Universally Unique Identifiers (also known as GUIDs, globally unique identifiers). They are used by RESQML as a key to uniquely identify high level objects. Every part in a RESQML dataset has a UUID assigned to it, including the individual property objects.

Behind the scenes, a UUID is a 128 bit integer, but it is usually presented in ascii in a specific hexadecimal form (see example below). All of this is the subject of an ISO standard, as these UUIDs are used all over place, not just in the oil industry.

As every part of a RESQML model has a UUID, and as the name suggests it is unique, this can be thought of as a primary key for the objects or parts in the dataset. Many of the resqpy methods work with UUIDs as a way of identifying a part. Here is an example of the single_array_ref() method we saw earlier, but now using the UUID for a particular property array:

```
ntg_array = pc.single_array_ref(uuid = 'fa52e6a2-dbbb-11ea-b158-248a07af10b2')
```

These UUIDs are not very human-friendly, so the examples don't tend to focus on them. However, for scripts running as part of automated jobs, their use is to be encouraged. The basic static property parts method we saw earlier is also available in a version that returns UUIDs instead of part names:

```
ntg_uuid, porosity_uuid, perm_i_uuid, perm_j_uuid, perm_k_uuid = pc.basic_static_
↪property_uuids(share_perm_parts = True)
```

6.7.8 Working with recurrent properties

The examples above will only uniquely identify a property array if it is a static property and the grid only has property data for a single realisation. To handle recurrent properties (i.e. properties that vary over time) or multiple realisations, more is needed...

Within the property collection, each instance of a recurrent property has a time index associated with it, along with a reference to a time series object which can be used to look up an actual date for a given time index value. If the property collection has come from the import of a single Nexus case, all the time indices will relate to the same time series. The model may additionally contain other time series objects. In particular, when importing from Nexus output, the resqpy code attempts to create 2 time series: one with all the Nexus timesteps and the other limited to the steps where recurrent properties were output which will usually be the one referred to by the property collection.

To find the UUID of the time series in use in the property collection, use:

```
ts_uuid_list = pc.time_series_uuid_list()
assert len(ts_uuid_list) == 1
ts_uuid = ts_uuid_list[0]
```

Given the UUID of the time series, we can instantiate a resqpy TimeSeries object:

```
import resqml.time_series as rqts
time_series = rqts.TimeSeries(model, time_series_root = model.root(uuid = ts_uuid))
```

The TimeSeries class includes various methods, for example:

```
ti_count = time_series.number_of_timestamps()
for time_index in range(ti_count):
print(time_index, time_series.timestamp(time_index))
```

The time indices relevant to a time series are in the range zero to number_of_timestamps() - 1. The list of indices at use in a property collection can be found with:

```
time_indices_list = pc.time_index_list()
```

Note that not all the recurrent properties will necessarily exist for all the time indices. Furthermore, the time indices are not generally the same as Nexus timestep numbers, because they usually refer to the reduced time series rather than the full Nexus time series.

TheTimeSeries.timestamp() method, shown in the for loop above, returns an ascii string representation of a date, or date and time, also in a format that is specified by an ISO standard. If you want to find the time index for a given date, use one of the following:

```
time_index = time_series.index_for_timestamp('2006-10-01')  # exact match required; note␣
↪format: YYYY-MM-DD
# following includes time of day; format: YYYY-MM-DDTHH:MM:SSZ
time_index = time_series.index_for_timestamp('2006-10-01T00:00:00Z')
# an alternative method not requiring an exact match
time_index = time_series.index_for_timestamp_not_later_than('2006-10-01T18:00:00Z')
```

Given a time index, we can use it as a criterion when identifying an individual array for a recurrent property. For example:

```
final_time_index = time_series.number_of_timestamps() - 1  # time indices count up␣
↪starting at zero
```

(continues on next page)

(continued from previous page)

```
final_water_saturation_array = pc.single_array_ref(citation_title = 'SW', time_index =␣
↪final_time_index)
```

The examples shown above will work for a RESQML dataset holding data from a single Nexus case, because we know that all the recurrent arrays will refer to the same time series. In the more general case, we might need to instantiate a separate time series object for each recurrent property: the UUID of the related time series is stored for each property array and can be found with:

```
initial_pressure_part = pc.singleton(property_kind = 'pressure', time_index = 0)  # time_
↪index of zero will be earliest
pressure_specific_ts_uuid = pc.time_series_uuid_for_part(initial_pressure_part)
pressure_time_series = rqts.TimeSeries(model, time_series_root = model.root(uuid =␣
↪pressure_specific_ts_uuid))
```

The resqpy time_series.py module also includes a TimeDuration class for working with time periods, ie. the interval between two timestamps.

6.7.9 Working with groups of properties

The collection of arrays for a recurrent property, at different reporting timesteps, form a logical group of properties. The resqpy property module provides functions and methods to help with these groupings. The first approach we'll look at involves creating a new property collection object for the group. Bear in mind that the actual arrays of data are only loaded on demand, so having multiple property collections instantiated is not a problem.

Here's a general way to create a new property collection as a subset of an existing one:

```
import resqpy.property as rqp
pressure_pc = rqp.selective_version_of_collection(pc, property_kind = 'pressure')
```

The selection criteria can involve any of the items we've seen before, such as citation_title or time_index (amongst others). Eg.:

```
inital_saturations_pc = rqp.selective_version_of_collection(pc, property_kind =
↪'saturation', time_index = 0)
```

There are some convenience functions in the property module for common groupings. Here is a function which will look for a particular simulator keyword as the citation title:

```
oil_sat_pc = rqp.property_collection_for_keyword(pc, 'SO')
```

If we have identified one part for a recurrent property, we can use it as an example to group other parts that only differ by time index:

```
pressure_pc = rqp.property_over_time_series_from_collection(pc, initial_pressure_part)
```

We can also merge a second property collection into a primary one, for example:

```
hydrocarbon_saturations_pc = rqp.property_collection_for_keyword(pc, 'SG')
hydrocarbon_saturations_pc.inherit_parts_from_other_collection(oil_sat_pc)
```

Note that the example above is not calculating a hydrocarbon saturation, it is merely collecting the oil and gas saturation arrays into a single property collection.

There is another mechanism for working with groups of properties (which we won't look at in detail here), and that is via a RESQML PropertySet object. This also groups together a set of property arrays, with the grouping also being an object in the dataset. The vdb import functions support generating some PropertySet objects, if desired. For example, the import_vdb_ensemble() function has an optional boolean argument create_property_set_per_realization. And one way to instantiate a respqy PropertyCollection object is for a given RESQML PropertySet object.

6.7.10 Working with multiple realisations

A RESQML property includes an optional realisation number. These are set by the resqpy functions to match the case number, when importing an ensemble of vdbs from a TDRM/Fortuna job. The resqpy PropertyCollection methods for selecting arrays accept a realization number as an optional argument. For example:

```
case_23_pore_volume_array = pc.single_array_ref(property_kind = 'pore volume',␣
↪realization = 23)
```

The set of realisation numbers present in a PropertyCollection can be found with the following method. Note that this does not imply that all properties are present for all the realisations, though for an ensemble built from a set of successful Nexus runs, that will usually be the case.

```
realization_list = pc.realization_list()
```

Depending on how one wants to work with the properties, the methods already discussed can be used to build property collections covering different subsets of all the arrays:

- all properties, for all realisations, for all timesteps
- all properties, for all realisations, for a single timestep
- all properties, for one realisation, for all timesteps
- all properties, for one realisation, for a single timestep
- any of the above combinations for a single property

Of course, the timestep options only apply to recurrent properties.

6.7.11 Supporting representation and indexable elements

Everything discussed so far about accessing RESQML properties applies not only to grid properties but also, for example, well logs and blocked well properties, amongst other things. The same classes and methods can be used when handling all these sorts of properties. (Though for convenience resqpy also has some derived classes such as WellLogCollection.) In RESQML, the object providing the discrete geometrical frame for the properties is referred to as the supporting representation, which for our purposes here is the grid.

The dimensionality of the underlying property arrays depends on the number of dimensions used to index an indexable element of the supporting representation. In the case of Nexus grid property arrays, the indexable elements are 'cells' and the K,J,I indexing is 3D. (All references to grids here refer to the IjkGridRepresentation RESQML class – other classes of grid are available in the standard!) But the same grid object could also have some properties where the indexable element is set to 'columns' and the array is 2D, indexed by J,I. Or how about an efficient representation of zonation with a categorical property where the indexable element is 'layers' – just a single zone number would be held for each layer of the grid, indicating which zone the layer is assigned to.

Another example could be transmissibility multipliers: simulators such as Nexus rather clumsily assign I-face multipliers to the cell either on the plus side of the face, or the minus side – and different simulators have adopted opposite protocols. In RESQML, 'faces' is also a valid indexable element for a grid, which makes more explicit where the data is applicable.

For Ijk Grid properties (excluding radial grids), the full list of possible indexable elements is:

- cells
- column edges
- columns
- coordinate lines
- edges
- edges per column
- faces
- faces per cell
- hinge node faces
- interval edges
- intervals
- I0
- I0 edges
- J0
- J0 edges
- layers
- nodes
- nodes per cell
- nodes per edge
- nodes per face
- pillars
- subnodes

6.7.12 High dimensional numpy arrays

Returning to the cell based grid properties... Despite the mechanisms for grouping property arrays, the data is actually stored in the hdf5 file as individual 3D numpy arrays. The 3 dimensions cover the K, J & I axes of the grid.

There are three methods in the PropertyCollection class for presenting a group of arrays as a single 4D numpy array. For example:

```
pore_volume_pc = rqp.selective_version_of_collection(pc, property_kind = 'pore volume')
pore_volume_4d_array = pore_volume_pc.realizations_array_ref()  # numpy array indexed by␣
↪R, K, J, I
```

Of course such arrays could be very large, so they should be used with caution – for example reducing the data to zonal values before creating the 4D array. The advantage is that extremely efficient numpy operations can then be used. For example to compute the cell-by-cell mean pore volume across all realizations:

```
mean_across_ensemble_pv_3d_array = np.nanmean(pore_volume_4d_array, axis = 0)
```

The other high dimensional array methods currently offered by the PropertyCollection class are for handling facets and time indices. Here is a facet example:

```
permeability_pc = rqp.selective_version_of_collection(pc, property_kind = 'permeability␣
↪rock')
facet_list = permeability_pc.facet_list()  # could return ['K', 'I'], for example, if we␣
↪have PERMZ and PERMX data
permeability_4d_array = permeability_pc.facets_array_ref()
# numpy array above indexed by F, K, J, I where F is also an index into facet_list
```

And for a 4D property array where the extra axis covers time indices:

```
pressure_pc = rqp.selective_version_of_collection(pc, property_kind = 'pressure')
time_index_list = pressure_pc.time_index_list()
pressure_4d_array = pressure_pc.time_series_array_ref()
# numpy array above indexed by T, K, J, I where T is also an index into time_index_list
```

Beyond these 4D arrays, we could combine some of these higher dimensions to produce, for example, 5D arrays covering realisations and time indices, or 6D arrays covering realisations, time indices and facets, as well as the K, J, I of the cell indices of course!

6.7.13 Creating new grid property objects

The discussion so far has focussed on accessing property arrays from a RESQML dataset – making them available to application code as numpy arrays. At some point though, we might want to store a new property array in the dataset. The resqml.derived_model module has a function for this. Note that all the functions in the derived model module work from and to datasets stored on disc. After calling such a function it is necessary to re-instantiate a Model object in order to pick up on the changes.

To add a property, first create the data as a numpy array. Here, for example, we compute pressure change:

```
initial_pressure_part = pc.singleton(property_kind = 'pressure', time_index = 0)
initial_pressure_array = pc.cached_part_array_ref(initial_pressure_part)
pressure_units = pc.uom_for_part(initial_pressure_part)

final_pressure_array = pc.single_array_ref(property_kind = 'pressure', time_index =␣
↪final_time_index)
# see earlier notes for finding final_time_index

pressure_change_array = final_pressure_array - initial_pressure_array  # example␣
↪calculation
```

Then call the function to add the new array as shown below. The full argument list is shown here to facilitate the discussion which follows. In practice, for this example, all the arguments after uom could be omitted.

```
import resqpy.derived_model as rqdm

rqdm.add_one_grid_property_array(epc_file = epc_path,
                                 a = pressure_change_array,
                                 property_kind = 'pressure',
                                 grid_uuid = grid.uuid,
                                 source_info = 'final pressure minus initial',
                                 title = 'PRESSURE CHANGE',
                                 discrete = False,
```

(continues on next page)

(continued from previous page)

```
                                    uom = pressure_units,
                                    time_index = None,
                                    time_series_uuid = None,
                                    string_lookup_uuid = None,
                                    null_value = None,
                                    indexable_element = 'cells',
                                    facet_type = None, facet = None,
                                    realization = None,
                                    local_property_kind_uuid = None,
                                    count_per_element = 1,
                                    new_epc_file = None)
```

The paragraphs below look at the argument list for that function in some more detail.

To re-open the model after calling a function in the derived_model module, simply re-instatiate a Model object:

```
model = rq.Model(epc_path)
```

epc_file

The first argument is the RESQML epc file which contains the grid. By default the new property will be added to this RESQML dataset (both the epc and h5 files will be updated). Another argument, new_epc_file, can be used as well if a new dataset is required instead of an update (see below).

a

The second argument is the numpy array holding the new property. It should have the appropriate shape for the grid (taking into consideration the indexable_element and count_per_element arguments). Assuming the default value of 'cells' for the indexable element (and 1 for count_per_element), the required shape is (nk, nj, ni).

The dtype (element data type) of the array should also be appropriate. Numpy arrays tend to default to a dtype of float, which will be a 64 bit floating point representation. For discrete data, be sure to use an integer data type such as int (64 bit) or int32, or int8 or bool for boolean data.

property_kind

This argument must be set and should be one of the supported property kinds, unless a local property kind is needed for the array (see below).

grid_uuid

This should be set to the UUID of the grid to which the array pertains.

source_info

The source info is a human readable string that should be set in such a way to help people understand where the data has come from. It is not used for any automated processing purposes.

title

The title is used to populate the citation title in the metadata for the new property object. Application code later in the workflow might rely on this to find the correct array.

discrete

This is a boolean indicating whether the data is discrete (True) or continuous (False). Set to True for any integer or boolean array data, including categorical data.

uom

The units must be specified. See earlier section for a list of the most common units we work with.

time_index & time_series_uuid

If the new property is part of a recurrent series, these two arguments should be specified. Here they are left as None because we are computing a single pressure change array. If we were generating a series of arrays, indicating the pressure change per reporting timestep, then these arguments would be needed.

string_lookup_uuid

If the property is categorical, this argument must be set to the UUID of the string lookup table object. The lookup table should be added to the model before adding the arrays, unless it already exists in the dataset. How to create objects such as lookup tables will be discussed elsewhere.

null_value

Continuous data always uses NaN (not-a-number) as the null value, and this argument should be left as None. However, NaN cannot be used in an integer array, so RESQML allows an integer value to be specified as null for each discrete or categorical property. It is usual to use -1 as the null value unless that is a valid value for the property.

indexable_element

This defaults to 'cells', which most grid properties are for. For map making, the value 'columns' might well get used. There are several other possibilities. The shape of the array must be correct for the value of this argument.

facet_type & facet

The RESQML standard allows a property object to have any number of facets. However, the resqpy code, including this function, generally works with at most one facet per property. If no facet is applicable to the property then these arguments should be left as None. The RESQML standard lists a few common facet types, though we are free to make up new ones. Facet types currently in use include:

- 'direction': 'I', 'J', 'K', 'IJ', or 'IJK'
- 'what': 'oil', 'gas', 'water' – used by resqpy for saturation or other phase related properties
- 'netgross': 'net', or 'gross' – used for thickness properties

Other standard facet types are: 'conditions', 'statistics', or 'qualifier'. The standard facet types are defined in the RESQML schema definition file properties.xsd

realization

Set this to the realization number if the property is applicable to one realization within an ensemble.

local_property_kind_uuid

If the property kind of the array is a 'local' property kind (i.e. not specified in the RESQML standard) then the property kind must already have been added (or exist) in the model and this argument is set to its UUID.

count_per_element

RESQML allows more than one value to be stored together, for each indexable element. This is achieved by adding an extra dimension to the array, being the 'fastest' cycling (ie. last numpy index). For example, imagine generating an array holding a complex number for each cell. The numpy array would have shape (NK, NJ, NI, 2) and the count_per_element argument would be set to 2.

new_epc_file

If this argument is set to a file path, the epc_file is not modified. A new epc (and paired h5) file will be created. The grid object and the coordinate reference system it refers to are copied to the new dataset and the newly created property added.

6.8 Working with a Single Property

The previous tutorial looked at working with sets of property arrays, using the resqpy PropertyCollection class. This tutorial explores an alternative when working with a single property array. The resqpy Property class behaves more like the other resqpy high level object classes. In particular, one resqpy Property object corresponds to one RESQML object of class ContinuousProperty, DiscreteProperty or CategoricalProperty.

6.8.1 Accessing a property array for a given uuid

Assuming that an existing RESQML dataset has been opened, we can find the uuid of a particular property using the familiar methods from the Model class. For example:

```
blocked_well_uuid = Model.uuid(obj_type = 'BlockedWellboreRepresentation', title =
↪'INJECTOR_3')
assert blocked_well_uuid is not None
kh_uuid = Model.uuid(obj_type = 'ContinuousProperty', title = 'KH', related_uuid =␣
↪blocked_well_uuid)
```

It is a good idea to include the *related_uuid argument*, as shown above, to ensure that the property is 'for' the required representation object (in this example a blocked well). Having determined the uuid for a property, we can instantiate the corresponding resqpy object in the familiar way:

```
import resqpy.property as rqp
inj_3_kh = rqp.Property(model, uuid = kh_uuid)
```

To get at the actual property data as a numpy array, use the *array_ref()* method:

```
inj_3_kh_array = inj_3_kh.array_ref()
```

The *array_ref()* method has some optional arguments for coercing the array element data type (dtype), and for optionally returning a masked array.

The Property class includes similar methods to PropertyCollection for retrieving the metadata for the property. As the Property object is for a single property, the '_for_part' is dropped from the method names, as is the part argument. For example:

```
assert inj_3_kh.property_kind() == 'permeability thickness'
assert inj_3_kh.indexable_element() == 'cells'
kh_uom = inj_3_kh.uom()
```

Behind the scenes, the Property object has a singleton PropertyCollection class as an attribute. This can be used by calling code if access to other PropertyCollection functionality is needed. Here is an example that sets up a normalised version of the array data:

```
normalized_inj_3_kh = inj_3_kh.collection.normalized_part_array(inj_3_kh.part, use_
↪logarithm = True)
```

6.8.2 When to use PropertyCollection and when Property

The main advantage of the PropertyCollection class is that it allows identification of individual property arrays or groups of property arrays based on the metadata items. In particular, the property kind and facet information is a preferable way to track down a particular property than relying on the citation title.

On the other hand, once the uuid for a particular property has been established, it can be passed around in code and used to instantiate an individual Property object when required. This is simpler than having to deal with a PropertyCollection once the individual uuid is known.

6.9 The Units of Measure system

Resqpy implements the full RESQML units of measure system, making it easy for you to track the of the units in your reservoir model rigorously.

Resqpy also has some helper functions for coercing invalid units, and for converting values between any compatible units.

Resqpy's unit systems module is *resqpy.weights_and_measures*, and is typically imported as:

```
import resqpy.weights_and_measures as wam
```

6.9.1 The Energistics Unit of Measure Standard

The RESQML standard has a rigorous unit system, which is shared with other Energistics standards PRODML and WITSML. The standard defines the concepts of Quantities, Dimensions, and Units of Measure (uoms), and also defines the set of allowed values for each.

All properties in a RESQML model are stored with a corresponding unit of measure.

Resqpy contains a database of this unit system, along with helper functions to coerce invalid units into RESQML-compliant units, and to convert between units.

6.9.2 Quantities

A Quantity represents a set of units with the same dimension and same underlying measurement concept.

To get a set of all possible quantities, use *valid_quantities()*:

```
>>> wam.valid_quantities()
{'area', 'volume', 'length', ...}
```

To see details about each quantity, such as the list of supported units of measure, use:

```
>>> wam.valid_quantities(return_attributes=True)
{'area': {
    'dimension': 'L2',
    'baseForConversion': 'm2',
    'members': ['acre', 'b', 'cm2', 'ft2', 'ha', 'in2', 'km2', ...]
    },
 'volume': ...
}
```

6.9.3 Units of Measure

A RESQML unit of measure (or "uom") is a unit symbol, such as "m" or "bbl".

Each uom has an associated dimension, and may be compatible with multiple quantities.

Resqpy can try to coerce an input string into a valid RESQML unit of measure. *rq_uom()* understands some common aliases:

```
>>> wam.rq_uom("metre")
"m"
>>> wam.rq_uom("scf")
"ft3"
>>> wam.rq_uom("p.u.")
"%"
```

To see the valid set of units of measure, use *valid_uoms()*:

```
>>> wam.valid_uoms()
{'%', '%[area]', '%[mass]', '%[molar]', '%[vol]', '(bbl/d)/(bbl/d)', ...}
```

You can filter to a given Quantity of interest:

```
>>> wam.valid_uoms(quantity="volume")
{'1000 bbl',  '1000 ft3', '1000 gal[UK]', '1000 gal[US]', ...}
```

To see details of each unit of measure such as the name and dimension, pass `return_attributes=True` to return a dictionary. For example, for the "indian foot" unit of measure:

```
>>> wam.valid_uoms(return_attributes=True)["ft[Ind]"]
{'name': 'indian foot',
'dimension': 'L',
'isSI': False,
'category': 'atom',
'baseUnit': 'm',
'conversionRef': 'EPSG',
'isExact': False,
'A': 0,
'B': 12,
'C': 39.370142,
'D': 0,
'description': "Indian Foot = 0.99999566 British feet (A.R.Clarke 1865).
    British yard (= 3 British feet) taken to be J.S.Clark's 1865 value of 0.9144025␣
↪metres."}
```

6.9.4 Converting between units

Each unit has four associated conversion factors A, B, C and D, which define how one can convert to and from a base unit.

A value x can be converted into the base unit with the formula:

$$y = \frac{A + Bx}{C + Dx}$$

where y represents a value in the base unit.

Use *convert()* to convert values between any compatible units of measure:

```
>>> wam.convert(1, unit_from="ft", unit_to="m")
0.3048
>>> wam.convert(1, unit_from="ft", unit_to="ft[US]")
0.999998
```

This will also work with numpy arrays, pandas dataframes or even distributed dask objects:

```
>>> import numpy as np
>>> x = np.array([1,2,3])
>>> wam.convert(x, unit_from="km", unit_to="m")
np.array([1000, 2000, 3000])
```

You can also convert arrays in-place:

```
>>> z = np.array([1,2,3])
>>> wam.convert(z, unit_from="km", unit_to="m", inplace=True)
>>> z
np.array([1000, 2000, 3000])
```

6.10 A first look at Well Objects

This tutorial introduces the classes relating to wells and goes into more detail for some of the basic ones. Other tutorials will cover the remaining well classes in depth.

6.10.1 The RESQML and resqpy classes for wells

The RESQML standard contains several classes of object that relate to wells. Each of these has an equivalent resqpy class, named (in parenthesis) in this list:

- MdDatum (MdDatum) - a simple class holding a datum location for measured depths
- DeviationSurveyRepresentation (DeviationSurvey) - inclination and azimuth at given measured depths
- WellboreTrajectoryRepresentation (Trajectory) - xyz coordinates at given measured depths
- WellboreFrameRepresentation (WellboreFrame) - list of measured depths supporting well log properties
- WellboreMarkerFrameRepresentation (WellboreMarkerFrame) - list of picks (well markers)
- BlockedWellboreRepresentation (BlockedWell) - list of cells visited or perforated by a well

The resqpy WellboreFrame and BlockedWell support related properties, which can be handled with the PropertyCollection and/or Property classes. However, for working with well logs, the resqpy property module includes the following classes for convenience:

- (WellLogCollection) - for managing logs, including a method for exporting in LAS format
- (WellLog) - for simpler access to a single log

RESQML also has organisational classes relating to wells:

- WellboreFeature (WellboreFeature) - a named entity representing a real, planned or conceptual well
- WellboreInterpretation (WellboreInterpretation) - one possible incarnation of a wellbore feature

There are various relationships between these classes. For example, a deviation survey or a trajectory must refer to a measured depth datum, and a blocked well must refer to a trajectory. Any of the representation objects can relate to a wellbore interpretation, which in turn must relate to a wellbore feature. The use of these optional organisational objects is encouraged and some software requires them to be present.

In resqpy, the default behaviour is to use the same well name as the citation title for any of the well objects that are in use for a given well. Note that if there are multiple competing interpretations, then it is best to assign a different title to each of the interpretations (and any related representations).

Most of the well related resqpy classes are contained in the *well.py* module. The feature and interpretation classes are in the *organize.py* module. Code snippets in this tutorial assume the following imports:

```
import resqpy.model as rq
import resqpy.well as rqw
import resqpy.organize as rqo
```

6.10.2 The measured depth datum class: MdDatum

A measured depth datum is a simple object which locates the datum for measured depths within a coordinate reference system. A direct reference to an MD datum is required in both a deviation survey and a trajectory. (And the other well representation objects refer to a trajectory, so an MD datum is always needed.)

When reading an existing dataset, a resqpy MdDatum object can be instantiated in the usual way by first identifying the required uuid. In this example we follow relationships from an interpretation object:

```
model = rq.Model('existing_model.epc')
pq13b_sidetrack_interpretation_uuid = model.uuid(obj_type = 'WellboreInterpretation',
                                                 title = 'PQ13B_SIDETRACK')
assert pq13b_sidetrack_interpretation_uuid is not None
pq13b_sidetrack_survey_uuid = model.uuid(obj_type = 'DeviationSurveyRepresentation',
                                         related_uuid = pq13b_sidetrack_interpretation_
↪uuid)
assert pq13b_sidetrack_survey_uuid is not None
pq13_md_datum_uuid = model.uuid(obj_type = 'MdDatum',
                                related_uuid = pq13b_sidetrack_survey_uuid)
assert pq13_md_datum_uuid is not None
pq13_md_datum = rqw.MdDatum(model, uuid = pq13_md_datum_uuid)
```

The MdDatum class doesn't have any exciting methods. Code accessing such an object will usually simply refer to some of the attributes, such as:

- crs_uuid: the uuid of the coordinate reference system within which the datum is located
- location: a triple float being the xyz location of the datum
- md_reference: a human readable string from a prescribed set, such as 'mean sea level', or 'kelly bushing'

The list of valid MD reference strings is defined in the RESQML schema definition and is available in the resqpy well module as:

```
rqw.valid_md_reference_list
```

6.10.3 Creating a new measured depth datum object

Most of the tutorials so far have focussed on reading existing data. As the MdDatum is such a simple object, it is a good place to start looking at how we create new objects using resqpy. In this example, we will add a new MdDatum, located fifteen metres to the east and two metres north of the existing datum which we identified above:

```
# prepare whatever data we need to populate the new object
pq13_location = np.array(pq13_md_datum.location)
new_location = tuple(pq13_location + (15.0, 2.0, 0.0))

# instantiate the resqpy object using data
pq14_md_datum = rqw.MdDatum(model,
                            crs_uuid = pq13_md_datum.crs_uuid,
                            location = new_location,
                            md_reference = pq13_md_datum.md_reference,
                            title = 'PQ14')

# the md datum class does not involve any arrays, so no need to write anything to hdf5

# create an xml tree (in memory) and add it to the model's dictionary of parts
pq14_md_datum.create_xml()

# update the epc file on disc (more typically done after creating a bunch of new objects)
model.store_epc()
```

Note that for a real well that has been drilled, the actual location of the datum should be available from the drilling information, so the example above is rather unrealistic.

Other resqpy objects can be created in a similar way. Note, however:

- most classes are much more complex than MdDatum, so much more data needs to be prepared
- resqpy includes import options for some classes, for reading the data from other formats
- many classes include array data, which require an extra step writing to the hdf5 file
- it is usual to call the model's *store_epc()* method once after a batch of objects have been added

6.10.4 The Trajectory class

The WellboreTrajectoryRepresentation class (Trajectory in resqpy) plays a central role in the modelling of wells in a RESQML dataset. Apart from a deviation survey, the other well representation classes all require a reference to a trajectory. It is the class which holds information about the path taken by a wellbore in physical space.

To instantiate a resqpy Trajectory for an existing RESQML WellboreTrajectoryRepresentation use the familiar methods:

```
pq13b_traj_uuid = model.uuid(obj_type = 'WellboreTrajectoryRepresentation',
                             title = 'PQ13B_SIDETRACK')
pq13b_trajectory = rqw.Trajectory(model, uuid = pq13b_traj_uuid)
```

As the amount of array data is modest for a trajectory, it is all loaded into memory at the time of instantiation. The main data of interest are the list of xyz points defining the path of the wellbore (within a coordinate reference system). The xyz data is available as a numpy array of shape (N, 3) in the *control_points* attribute, e.g.:

```
td_z = pq13b_trajectory.control_points[-1, 2]
```

The measured depths corresponding to the xyz control points are also available in a numpy vector of shape (N,) e.g.:

```
td_md = q13b_trajectory.measured_depths[-1]
```

There are several other attributes, including:

- crs_uuid
- md_uom: the unit of measure (usually 'm' or 'ft') for the measured depths, which don't belong in any crs as such
- md_datum: an MdDatum object
- knot_count: an integer being the number of 'knots', or points in the arrays (i.e. the value of N above)
- line_kind_index: an integer in the range -1..5 indicating how the control points should be interpreted (see below)

It is common practice for application code to treat the trajectory as a piecewise linear spline between the control points. The *line_kind_index* indicates how the data can be interpreted more rigorously. It may have the following values:

- -1: null value, there is no line!
- 0: vertical: the trajectory follows a vertical path beneath the MdDatum location; control points need not be supplied
- 1: linear spline: a piecewise linear spline with sudden changes in direction at control points
- 2: natural cubic spline: a cubic spline with direction control at the two end points
- 3: cubic spline: a cubic spline with no sudden changes in direction
- 4: z linear cubic spline: another form of cubic spline
- 5: minimum curvature spline: the path which has least severe rate of change of direction

When converting from inclination and azimuth data, as acquired by a deviation survey, the minimum curvature interpretation is invariably applied, so the line kind index often has the value 5, even though applications often interpret the trajectory as if it had value 1. For many applications, the differences will be insignificant.

A resqpy Trajectory object has other attributes – some of the optional ones are:

- tangent_vectors: a numpy array of shape (N, 3) holding tangent vectors for the control points
- deviation_survey: a DeviationSurvey object from which the trajectory has been derived
- wellbore_interpretation: a related WellboreInterpretation object
- wellbore_feature: a WellboreFeature object indirectly related via an interpretation object

The Trajectory class offers some methods for setting up a new trajectory from other data sources. These can be triggered by use of appropriate arguments to the initialisation function. The methods are:

- compute_from_deviation_survey(): derives a trajectory from inclination and azimuth data on a minimum curvature basis
- load_from_dataframe(): takes MD, X, Y & Z values from columns of a pandas dataframe
- load_from_ascii_file(): similar to load_from_dataframe() but with the data in a tabular file
- load_from_cell_list(): sets the control points to the cell centres, for a list of cells in a grid
- load_from_wellspec(): similar to load_from_cell_list() but starting from a Nexus WELLSPEC file
- splined_trajectory(): from an existing trajectory, create a new one with more control points following a cubic spline

There is one commonly used method for finding the xyz location for a given measured depth:

- xyz_for_md(): returns the interpolated xyz point based on a simple piecewise linear spline interpretation

6.10.5 Creating a new trajectory object

In this example we will add a new Trajectory given the following pandas dataframe (the numbers are made up and might not be realistic!):

```
df = pd.DataFrame(((2170.00, 450123.45, 5013427.21, 2100.00),
                   (2227.00, 450108.95, 5013432.77, 2150.00),
                   (2288.00, 450081.02, 5013434.25, 2200.00),
                   (2349.00, 450067.83, 5013433.91, 2250.00),
                   (2399.82, 450064.05, 5013433.44, 2300.00)),
          columns = ('MD',     'X',        'Y',        'Z'))
```

We need to establish an MdDatum object for the well. Here we will assume that the datum is vertically above the first control point in our dataframe. We will also assume that the coordinate reference system object already exists:

```
datum_xyz = df['X'][0], df['Y'][0], df['Z'][0] - df['MD'][0]
md_datum = rqw.MdDatum(model,
                       crs_uuid = model.crs_uuid,  # handy if all your objects use the␣
↪same crs
                       location = datum_xyz,
                       md_reference = 'ground level',
                       title = 'spud datum')
md_datum.create_xml()
```

Now we have enough to instantiate the resqpy Trajectory:

```
trajectory = rqw.Trajectory(model,
                            md_datum = md_datum,
                            data_frame = df,
                            length_uom = 'm',  # this is the md_uom
                            well_name = 'Wildcat 1')
```

The trajectory now exists in memory as a resqpy object but it has not been added to the model in any persistent way. For temporary objects, this state is sometimes fine to work with. However we usually want to add the new object fully. Before doing that, we can optionally call the following method to create an interpretation object and a feature for the well:

```
trajectory.create_feature_and_interpretation()
```

Now we are ready to fully add the trajectory (and related objects) with:

```
trajectory.write_hdf5()
trajectory.create_xml()
```

This is followed by writing to the epc with the following, which will include all the new objects:

```
model.store_epc()
```

If the model contained just a Crs object before the sequence shown above, then after execution the model.parts() method will return something like:

```
['obj_LocalDepth3dCrs_672bbf86-e4be-11eb-a560-80e650222718.xml',
 'obj_MdDatum_6733ad4a-e4be-11eb-a560-80e650222718.xml',
 'obj_WellboreFeature_67368eb6-e4be-11eb-a560-80e650222718.xml',
 'obj_WellboreInterpretation_67369082-e4be-11eb-a560-80e650222718.xml',
 'obj_WellboreTrajectoryRepresentation_67352698-e4be-11eb-a560-80e650222718.xml']
```

The other well classes will be covered in later tutorials.

6.11 Surface Representations

This tutorial discusses surfaces, which can be used to represent the geometry of geological horizons, fault planes, and fluid contacts. In RESQML, the two main classes of object used for surfaces (with the equivalent resqpy class in brackets) are:

- TriangulatedSetRepresentation (Surface) - a triangulated surface can be used to represent torn or untorn surfaces
- Grid2dRepresentation (Mesh) – also known as a *lattice*, has a squared grid of points and can only fully define an untorn surface

The Grid2dRepresentation class (and its resqpy equivalent, Mesh) supports various options regarding the regularity of the lattice of points.

Two more classes are also sometimes used when constructing surfaces:

- PointSetRepresentation (PointSet) - a set of points; resqpy includes a function for making a *Delaunay Triangulation* from a point set
- PolylineSetRepresentation (PolylineSet) - can be used to represent a set of fault sticks

Resqpy includes a further class (CombinedSurface) which does not have a corresponding RESQML class. It allows a set of surfaces to be treated as a single surface, which can make some application functionality easier to implement.

In this tutorial we assume the following import statements:

```
import resqpy.model as rq
import resqpy.surface as rqs
import resqpy.organize as rqo
import resqpy.olio.uuid as bu
```

6.11.1 Working with an existing Surface

Instantiating a resqpy Surface object for an existing RESQML TriangulatedSetRepresentation follows the familiar steps of identifying the uuid and then calling the initialisation method. In this example we start with the title of a horizon interpretation:

```
model = rq.Model('existing_geo_model.epc')
horizon_interp_uuid = model.uuid(obj_type = 'HorizonInterpretation', title = 'Base␣
↪Cretaceous')
surface_uuid = model.uuid(obj_type = 'TriangulatedSetRepresentation', related_uuid =␣
↪horizon_interp_uuid)
base_cretaceous_surface = rqs.Surface(model, uuid = surface_uuid)
```

Application code will often need to access the triangulated data, for example to render the surface graphically, using the following method:

```
triangles, points = base_cretaceous_surface.triangles_and_points()
```

The *triangles* return value is a numpy int array of shape (NT, 3) where NT is the number of triangles and the 3 covers the three corners of a triangle. The values in *triangles* are indices into the *points* numpy float array, which has shape (NP, 3), with NP being the number of points and the 3 covering the xyz coordinates. Therefore this assertion should always pass:

```
assert np.all(triangles < len(points))
```

Almost all the other methods in the Surface class are for creating new surfaces in various ways.

6.11.2 Creating a new Surface

The resqpy Surface class includes many methods for setting up the data for new surface. A few of these methods can be triggered by using certain arguments to the *init* method. The general way, though, is to create an empty Surface object and then call one of the *set_from_* or *set_to_* methods. Here is an example which creates a simple horizontal plane covering a rectangular area:

```
min_x, max_x = 450000.00, 520000.00
min_y, max_y = 6400000.00, 6600000.00
surface_depth = 2700.00
xyz_box = np.array(((min_x, min_y, 0,0), (max_x, max_y, 0.0)))  # z values will be␣
↪ignored
# create an empty surface
horizontal_surface = rqs.Surface(model, crs_uuid = model.crs_uuid)
# populate the resqpy object for a horizontal plane
horizontal_surface.set_to_horizontal_plane(depth = surface_depth, box_xyz = xyz_box)
# write to persistent storage (not always needed, if the object is temporary)
horizontal_surface.write_hdf5()
horizontal_surface.create_xml()
model.store_epc()
```

The *set_to_horizontal_plane* method generates a very simple surface using four points and two triangles:

```
t, p = horizontal_surface.triangles_and_points()
assert len(t) == 2 and len(p) == 4
```

Here is a full list of the methods for setting up a new Surface:

- set_to_horizontal_plane - discussed above
- set_from_triangles_and_points - when the triangulation has been prepared in numpy arrays
- set_from_point_set - generates a Delaunay Triangulation for a set of points (computationally expensive)
- set_from_irregular_mesh - where the points form an irregular lattice (think of a stretched and warped piece of squared paper)
- set_from_sparse_mesh - similar to above but mesh may contain NaNs, which will result in holes in the surface
- set_from_mesh_object - starting from a resqpy Mesh object
- set_from_torn_mesh - points are in a numpy array with duplication at corners of 2D 'cells'; gaps will appear in the surface where corners of neighbouring cells are not coincident
- set_to_single_cell_faces_from_corner_points - creates a Surface representing all 6 faces of a hexahedral cell (typically from an IjkGridRepresentation geometry)

- set_to_multi_cell_faces_from_corner_points - similar to above but representing all the faces of a set of cells
- set_to_triangle - creates a Surface for a single triangle
- set_to_sail - creates a Surface with the geometry of a triangle wrapped on a sphere
- set_from_tsurf_file - import from a GOCAD tsurf file
- set_from_zmap_file - import from a zmap format ascii file
- set_from_roxar_file - import from an RMS format ascii file

If a Surface is created from a simple (untorn) mesh, with either *set_from_irregular_mesh* or *set_from_mesh_object*, then the following method can be used to locate which 2D 'cell' a particular triangle index is for. Resqpy includes functions for finding where a line intersects a triangulated surface. Those functions can return a triangle index which can be converted back to a mesh 'cell' (referred to as a column in the method name) with:

- column_from_triangle_index

Similarly, if a Surface is created using *set_to_single_cell_faces_from_corner_points* or *set_to_multi_cell_faces_from_corner_points*, the cell and face for a given triangle index can be identified with:

- cell_axis_and_polarity_from_triangle_index

6.11.3 The resqpy CombinedSurface class

The CombinedSurface class allows a set of Surface objects to be treated as a single composite surface for some purposes. It can be useful when looking for wellbore trajectory intersections and might also be convenient in some graphical applications.

A combined surface is initialised simply from a list of resqpy Surface objects, e.g.:

```
all_horizons = rqs.CombinedSurface([top_reservoir_surface, base_triassic_horizon, base_
↪reservoir_surface])
```

As this is a derived resqpy class, it is not written to persistent storage, there is no xml and it is not added to the model. There are only two useful methods. The first, *triangles_and_points* behaves just the same as the Surface method:

```
t, p = all_horizons.triangles_and_points()
```

And the second, *surface_index_for_triangle_index* identifies which surface, together with its local triangle index, a combined surface triangle index is equivalent to:

```
surface_index, local_triangle_index = all_horizons.surface_index_for_triangle_index(6721)
```

In that example, *surface_index* is an index into the list of surfaces passed to the initialisation of the combined surface object.

6.11.4 Introducing the Mesh class

The resqpy Mesh class is equivalent to the Grid2dRepresentation RESQML class. It can be used to represent a depth map for a surface such as a horizon and is characterised by usually having a regular two-dimensional lattice of points in the xy plane. RESQML allows various options for storing the data. Which option is in use is visible as the resqpy Mesh attribute *flavour* which can have the following values:

- 'explicit' - full xyz values are provided for every point, with an implied logical IJ orderliness in the xy space
- 'regular' - the xy values form a perfectly regular lattice and there are no z values

- 'reg&z' - the xy values form a perfectly regular lattice and there are explicit z values
- 'ref&z' - the xy values are stored in a separate referenced Mesh (typically of flavour 'regular'), there are explicit z values

The logical size of the lattice can be found with a pair of attributes: *ni* and *nj*. These hold the number of points in the I and J axes. Note that these hold a node or point count, not a 'cell' count.

6.11.5 Reading an existing Mesh

Initialising a resqpy Mesh object for an existing RESQML Grid2dRepresentation follows the familiar steps of identifying the uuid and passing that value to the __init__ method. For example:

```
top_reservoir_mesh_uuid = model.uuid(obj_type = '', title = 'Top Reservoir')
top_reservoir_mesh = rqs.Mesh(model, uuid = top_reservoir_mesh_uuid)
```

Regardless of which flavour a mesh is, a fully expanded numpy array of xyz values can be accessed with:

```
xyz_array = top_reservoir_mesh.full_array_ref()
```

Another generic method which will work for any flavour of Mesh object is *surface*, which generates a Surface object for the Mesh:

```
top_reservoir_surface = top_reservoir_mesh.surface(quad_triangles = True)
```

The *quad_triangles* boolean argument causes each 'square' (more generally, each quadrilateral) to be represented by four triangles rather than two, using an extra point at the centre of the square (the mean of the four vertices). This gives a unique representation whereas the default two triangle representation yields a different surface depending upon which diagonal is used for a non-planar quadrilateral.

For a mesh with a regular lattice of points, the origin and spacing can be found in the following way:

```
assert top_reservoir_mesh.flavour in ['regular', 'reg&z']
origin_xyz = top_reservoir_mesh.regular_origin
deltas_xyz = top_reservoir_mesh.regular_dxyz_dij
```

Those arrays contain xyz values even though the z values are typically zero and not used. The origin is a simple triple. The regular_dxyz_dij attribute is a numpy array of shape (2, 3) which holds the xyz step size for each step in I (first index zero) and J (first index one).

Of course the geometry exists within a coordinate reference system and that can be identified with the *crs_uuid* attribute.

6.11.6 More on the 'ref&z' flavour of Mesh

Where two or more meshes share a common xy lattice of points, differing only in z values, it can be useful to use the 'ref&z' flavour. The xy arrangement can be represented by a Mesh of flavour 'regular', to which the other meshes refer. Alternatively, one of the meshes can act as the master and have flavour 'reg&z', with the other meshes referring to it and having flavour 'ref&z'.

The main advantage of this way of working is that it is clear that the different sets of z values 'go with' the same set of xy values. Application code can make use of this knowledge. The following snippet checks that a mesh for a base reservoir horizon references one for the top reservoir, allowing the differences in z values to be meaningfully computed.

```
assert base_reservoir_mesh.flavour == 'ref&z'
assert bu.matching_uuids(base_reservoir_mesh.ref_uuid, top_reservoir_mesh.uuid)
thickness = base_reservoir_mesh.full_array_ref()[2] - top_reservoir_mesh.full_array_
↪ref()[2]
```

Note the use of the *ref_uuid* attribute in that snippet, to identify the mesh being referenced for the xy values.

6.11.7 The PointSet class

A set of points in 3D space can be held in a RESQML object of class PointSetRepresentation and the equivalent resqpy class is PointSet. That class includes a *full_array_ref* method which returns a numpy array of shape (N, 3) holding the xyz values of the points.

If a set of points is representing a surface, it is usually necessary to convert it to a Surface object using a Delaunay Triangulation, e.g.:

```
owc_point_set = rqs.PointSet(model, uuid = owc_contact_picks_point_set_uuid)
owc_surface = rqs.Surface(model, point_set = owc_point_set, title = 'oil-water contact␣
↪from picks')
```

Note that the Delaunay Triangulation can be a computationally expensive operation. It is probably worth storing the resulting surface as a persistent object:

```
owc_surface.write_hdf5()
owc_surface.create_xml()
model.store_epc()
```

6.11.8 A non-standard use of Mesh

The resqpy library includes a DataFrame class (and some derived classes) which, behind the scenes, map a numerical pandas dataframe onto a Mesh object of flavour 'reg&z' (or 'regular' in the case of multiple realisations, in which case the values are stored as continuous property objects). The RESQML standard did not intend Grid2dRepresentation objects to be used in this way, so such dataframes will not generally be usable by RESQML enabled software that does not use the resqpy API.

6.12 Creating a Test Dataset

This short tutorial makes use of a resqpy test function to generate a small dataset which can be used in other tutorials. Creating a RESQML dataset from scratch is significantly more complicated than reading an existing dataset and will be covered in later tutorials.

6.12.1 Importing from a test module

Test modules can be acquired by cloning/downloading from the repository.

For the following import to work, the resqpy code will need to have high priority in the environmental PATH. This can be achieved with something like:

```
import sys
sys.path.insert(0, '/path/to/your/resqml')
```

The test modules included in the resqpy repository are primarily for automated testing to help ensure that code changes don't break existing functionality. However, some of the modules can also be used to generate small test datasets for use in these tutorials, for example:

```
from tests.test_s_bend import test_s_bend_fn
```

If that import fails, you might need to track down the test_s_bend.py module and copy it into your working directory, and then try:

```
from test_s_bend import test_s_bend_fn
```

Alternatively, try modifying the PYTHONPATH environment variable prior to running your Python script so that one of those two formulations works.

6.12.2 Creating the S-bend dataset

Having imported the function, the S-bend dataset can be generated with:

```
test_s_bend('s_bend.epc')
```

You can substitute whatever path or filename you like, but you should use the extension `.epc` This file will be created, along with a paired hdf5 format file with the same name, but extension `.h5`

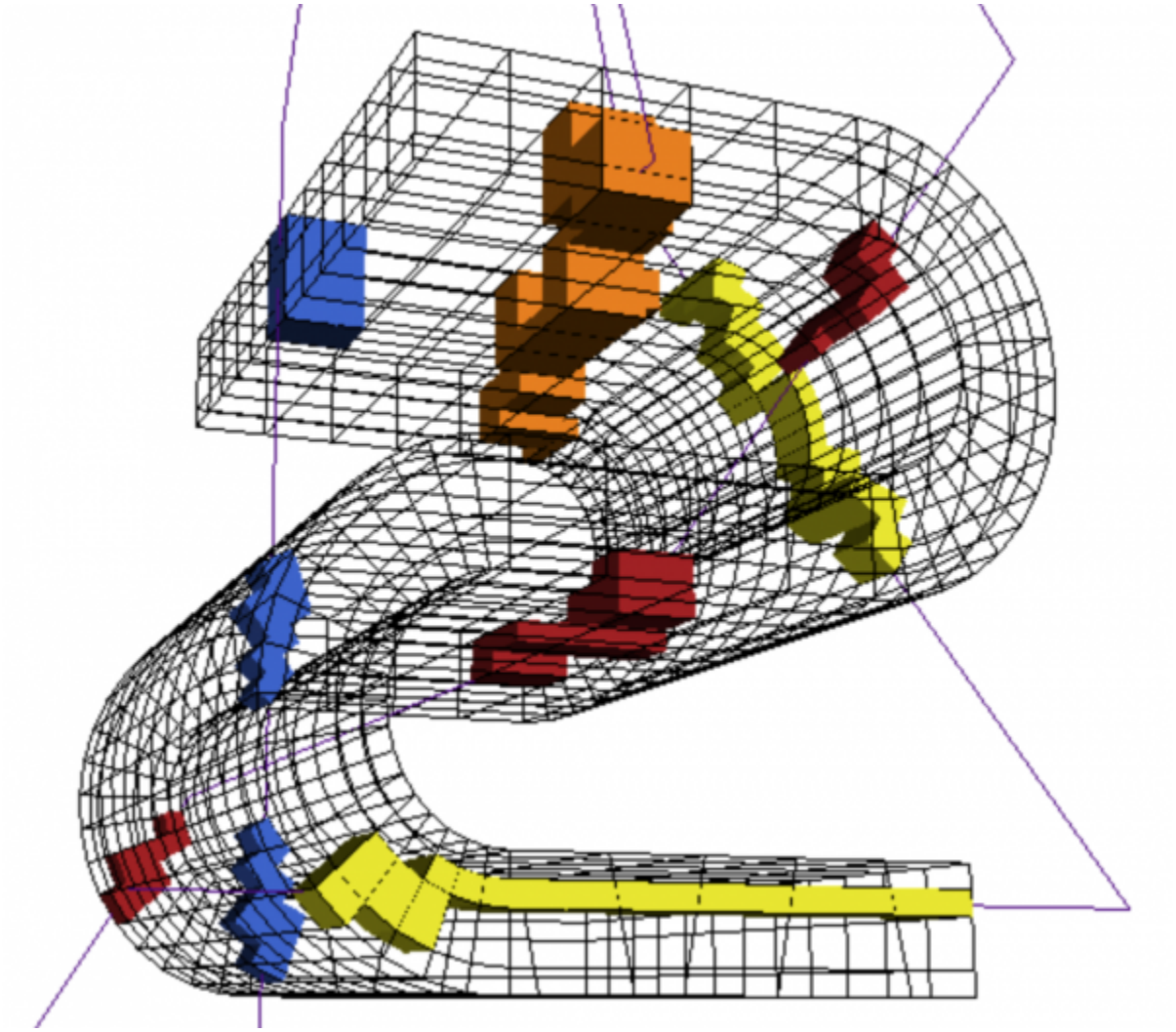
(The function generates a few warning messages, which might appear, depending on what Python log handling is active in your environment. These warnings can be ignored.)

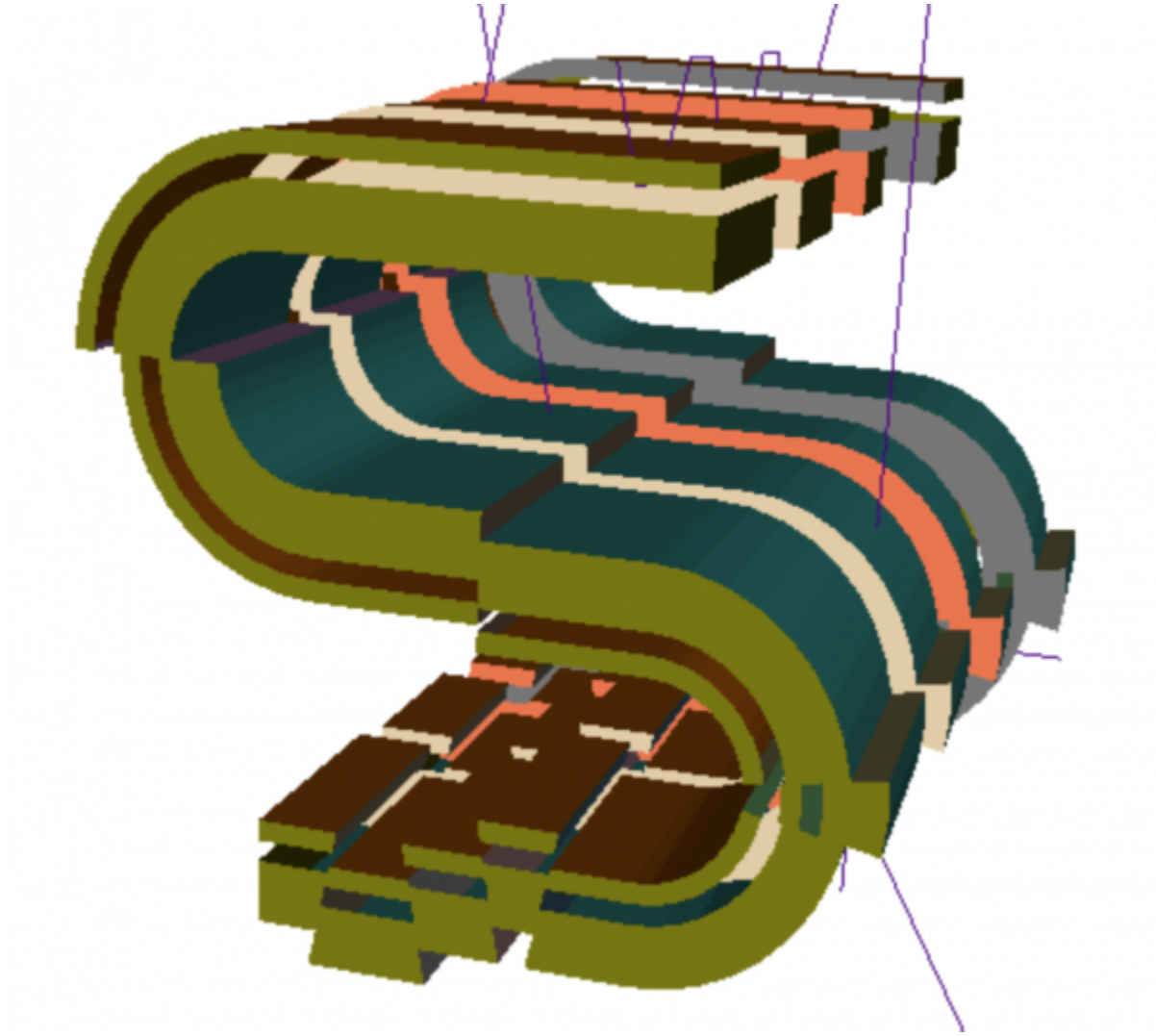
6.12.3 About the S-bend model

The S-bend model contains some Ijk Grid objects, Wellbore Trajectory Representations and Blocked Wellbore Representations, plus a few other objects. The model contains some quite extreme geometries, as its main purpose is for testing that other code can handle some of the more challenging features that can be represented in RESQML.

In particular, the 3 grids have variants of a geometry which include two recumbent folds (with an inverted section of reservoir between). They all contain a totally pinched out layer. Two of the grids include several faults in the geometry, including some with a lateral slip between split pillars. One of those two grids also contains a K Gap.

Three of the four wells also have rather unrealistic trajectories, designed to test well to grid intersection code.





6.12.4 Checking the contents of the S-bend dataset

To check that the S-bend model has been generated, you can open it as shown in the previous tutorial:

```
import resqpy.model as rq
s_bend = rq.Model('s_bend.epc')
```

You can then check the list of parts with:

```
s_bend.parts()
```

which should produce something like:

```
['obj_LocalDepth3dCrs_61e89d62-614e-11eb-940b-248a07af10b2.xml',
'obj_IjkGridRepresentation_61e8997a-614e-11eb-940b-248a07af10b2.xml',
'obj_MdDatum_61f00782-614e-11eb-940b-248a07af10b2.xml',
'obj_WellboreTrajectoryRepresentation_61f03f2c-614e-11eb-940b-248a07af10b2.xml',
'obj_WellboreTrajectoryRepresentation_61f343c0-614e-11eb-940b-248a07af10b2.xml',
```

(continues on next page)

(continued from previous page)

```
'obj_WellboreTrajectoryRepresentation_61f5ca28-614e-11eb-940b-248a07af10b2.xml',
'obj_WellboreTrajectoryRepresentation_61f87444-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_621a10a4-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_622a66fc-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_62361128-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_6245fbf6-614e-11eb-940b-248a07af10b2.xml',
'obj_IjkGridRepresentation_6274e52e-614e-11eb-940b-248a07af10b2.xml',
'obj_PropertyKind_6276f40e-614e-11eb-940b-248a07af10b2.xml',
'obj_DiscreteProperty_627572be-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_628708bc-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_62906132-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_629a94e0-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_62a95dcc-614e-11eb-940b-248a07af10b2.xml',
'obj_IjkGridRepresentation_62d7a2cc-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_62f7f6d0-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_630e9a02-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_63234d30-614e-11eb-940b-248a07af10b2.xml',
'obj_BlockedWellboreRepresentation_6343a2a6-614e-11eb-940b-248a07af10b2.xml']
```

(The hexadecimal uuids will differ from those shown here, and it's possible that the order of the list will be different.)

6.13 Working with the RelPerm class and equinor/pyscal

This tutorial describes two workflows that allow us to pass relative permeability and capillary pressure data between the resqpy library and the equinor\pyscal library.

We will be moving water-oil relperm data between resqpy's `resqpy.olio.relperm.RelPerm` class and pyscal's `pyscal.WaterOil` class.

Please note that similar workflows can be used for moving gas-oil data to/from the `pyscal.GasOil` class.

6.13.1 Importing the relperm and wateroil modules

In this tutorial we will be moving water-oil relperm data between the `RelPerm` object's `resqpy.olio.relperm.RelPerm.dataframe()` method and the `WaterOil` object's `pyscal.WaterOil.table` attribute.

```
from resqpy.olio.relperm import RelPerm
from pyscal import WaterOil
# import a plotting library for visual inspection of the data
import matplotlib.pyplot as plt
# import resqpy model module to interact with a resqpy model
import resqpy.model as rm
```

6.13.2 resqpy RelPerm.dataframe() to pyscal WaterOil.table

Initialize an instance of a `RelPerm` object that is stored in a resqpy `Model` instance. This can be done using the *uuid* of the existing `Grid2dRepresentation` object that acts as support for the dataframe of relperm data.

```
model = rm.Model('/path/to/my_file.epc')
relperm_wo = RelPerm(model = model, uuid = uuid)
```

The dataframe of water-oil relperm data is then accessed using the `resqpy.olio.relperm.RelPerm.dataframe()` method.

```
relperm_wo_df = relperm_wo_obj.dataframe()
```

To initialize a pyscal `WaterOil` object, we first need to define a water saturation end-point that is compatible with the `RelPerm` dataframe being inputted. In this case, we can define *swl*, which will be the first water saturation in the generated `WaterOil` table, and set it equal to the minimum water saturation value in the `RelPerm` dataframe.

```
swl = relperm_wo_df.min()['Sw']
pyscal_wo = WaterOil(swl = swl)
```

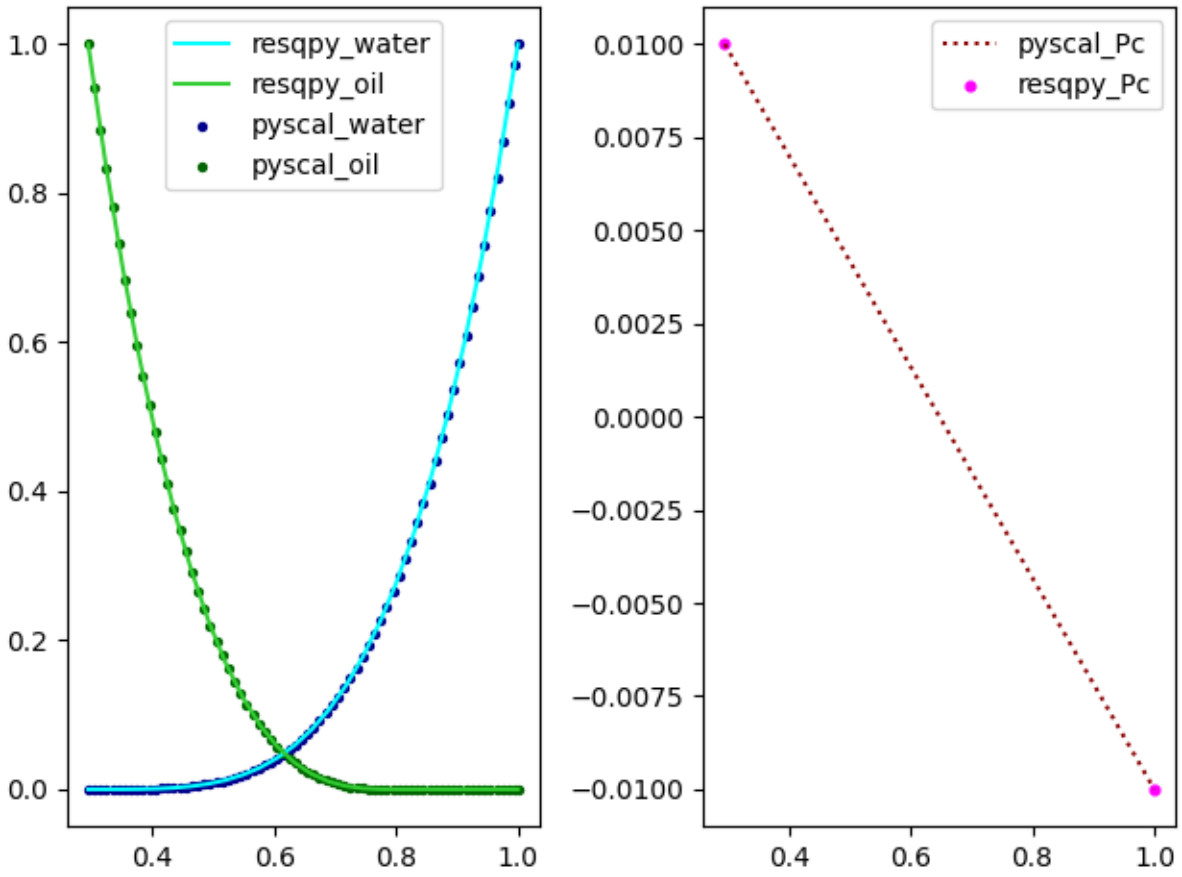
The pyscal `pyscal.WaterOil.add_fromtable()` method generates a relperm table by interpolating relative permeability and capillary pressure data from the inputted dataframe.

```
pyscal_wo.add_fromtable(dframe = relperm_wo_df, swcolname = 'Sw', krwcolname='Krw',␣
↪krowcolname='Kro', pccolname='Pc')
pyscal_wo_df = pyscal_wo.table
```

We'll generate some plots to visually compare the inputted `RelPerm` data to the `WaterOil` data.

```
fig, (ax1, ax2) = plt.subplots(nrows = 1, ncols = 2)
ax1.plot(relperm_wo_df['Sw'], relperm_wo_df['Krw'], label = 'resqpy_water', c = 'cyan')
ax1.plot(relperm_wo_df['Sw'], relperm_wo_df['Kro'], label = 'resqpy_oil', c = 'limegreen
↪')
ax1.scatter(pyscal_wo_df['SW'], pyscal_wo_df['KRW'], label = 'pyscal_water', c =
↪'darkblue', s = 8)
ax1.scatter(pyscal_wo_df['SW'], pyscal_df['KROW'], label = 'pyscal_oil', c = 'darkgreen',
↪ s = 8)
ax1.legend( )
ax2.scatter(relperm_wo_df['Sw'], relperm_wo_df['Pc'], label = 'resqpy_Pc', c = 'magenta',
↪ s = 12)
ax2.plot(pyscal_wo_df['SW'], pyscal_wo_df['PC'], label = 'pyscal_Pc', linestyle = 'dotted
↪', c = 'darkred' )
ax2.legend( )
fig.tight_layout()
plt.show()
```

The image below compares two sets of relperm and capillary pressure data:



6.13.3 pyscal WaterOil.table to resqpy RelPerm.dataframe()

Moving data in the opposite direction is simple, and involves reformatting the column names of the `WaterOil` table to be compatible with the `RelPerm` initialiser method. We reference the same `WaterOil` table instance, pyscal_wo_df, from the previous section.

```
model = rm.Model('/path/to/my_file.epc')
all_relevant_pyscal_cols = ['SW', 'SG', 'KRW', 'KRG', 'KROW', 'KROG']
cols = sorted(list(set(pyscal_wo_df.columns).intersection(set(all_relevant_pyscal_
↪cols))), reverse=True)
if 'PC' in pyscal_wo_df.columns:
    cols.append('PC')
col_remap_dict = {k: (k.capitalize() if len(k) < 4 else k.capitalize()[0:3]) for k in␣
↪cols}
pyscal_wo_df_processed = pyscal_wo_df[cols].rename(columns = col_remap_dict)
# intialize a new RelPerm object, write hdf5 and create xml for object
relperm_wo = RelPerm(model = model, df = pyscal_wo_df_processed)
relperm_wo.write_hdf5_and_create_xml()
```

6.14 Multiprocessing

This tutorial is about using multiprocessing with specific resqpy functions to speed up multiple function calls.

You should edit the file paths in the examples to point to your own files.

6.14.1 Installing Dask

To use the multi_processing module, *Dask* needs to be installed in the Python environment because it is not a dependency of the project. *Dask* is a flexible open-source Python library for parallel computing. It scales Python code from multi-core local machines to large distributed clusters on-prem or in the cloud.

Dask contains multiple modules but only the *distributed* module is needed here. *Dask Distributed* can be installed using pip, conda, or from source.

Pip

```
python -m pip install dask distributed
```

Conda

```
conda install dask distributed -c conda-forge
```

Source

```
git clone https://github.com/dask/distributed.git
cd distributed
python -m pip install .
```

If using a Job Queue Cluster, *Dask Jobqueue* must also be installed. This can be installed in the same ways.

Pip

```
python -m pip install dask-jobqueue
```

Conda

```
conda install dask-jobqueue -c conda-forge
```

Source

```
git clone https://github.com/dask/dask-jobqueue.git
cd dask-jobqueue
python -m pip install .
```

6.14.2 Cluster & Client Setup

If using a local machine, a `LocalCluster` must be setup. If using a job queing system, a `JobQueueCluster` can be used such as an `SGECluster`, `SLURMCluster`, `PBSCluster`, `LSFCluster` etc. Full details can be found at https://docs.dask.org/en/latest/deploying.html

A client can also be setup to provide a live feedback dashboard or to capture diagnosics, which is explained in the next section.

Local Cluster

Documentation of creating a `LocalCluster` can be found at https://distributed.dask.org/en/stable/api.html#distributed.LocalCluster

```
from dask.distributed import Client, LocalCluster

cluster = LocalCluster()
client = Client(cluster)
```

Job Queue Cluster Example

As an example, an SGE Cluster can be setup using *Dask Jobqueue*. Documentation of creating a `JobQueueCluster` can be found at https://jobqueue.dask.org/en/latest/api.html

```
from dask.distributed import client
from dask_jobqueue import SGECluster

cluster = SGECluster(
    processes=1,        # Number of workers per job.
    cores=96,           # Total amount of physical cores for all workers.
    memory="360 GiB",   # Usable memory per node.
    scheduler_options={"dashboard_address": ":0"}   # Other scheduler options.
)
client = Client(cluster)
```

6.14.3 Viewing the Client

If using a Local Cluster, the client dashboard is typically served at http://localhost:8787/status , but may be served elsewhere if this port is taken. The address of the dashboard will be displayed if you are in a Jupyter Notebook, or can be queried from `client.dashboard_link`.

Some clusters restrict the ports that are visible to the outside world. These ports may include the default port for the web interface, 8787. There are a few ways to handle this:

- Open port 8787 to the outside world. Often this involves asking your cluster administrator.
- Use a different port that is publicly accessible using the `scheduler_options` argument, like above.
- Use fancier techniques, like Port Forwarding.

You can capture some of the same information that the dashboard presents for offline processing using the `Client.get_task_stream` and `Client.profile` methods. These capture the start and stop time of every task and transfer, as well as the results of a statistical profiler. More info on this can be found at https://docs.dask.org/en/stable/diagnostics-distributed.html#capture-diagnostics

6.14.4 Uplading Packages/ Files to the Workers

If using a Job Queue Cluster, the resqpy package may need to be uploaded for the workers to use. A dependency file that contains the path of the installed resqpy package or the location of a local git clone of the repo can be uploaded to the client.

```
dependencies = """
import sys
sys.path.insert(0, "path/to/local/resqpy/clone")
"""

with tempfile.TemporaryDirectory() as tempdir:
    filename = os.path.join(tempdir, "dependencies.py")
    with open(filename, "w") as f:
        f.write(dependencies)

    client.wait_for_workers()
    client.upload_file(filename)
```

Environment variables may also need to be set such as the *Numba* thread limit, which can be done by running a defined function.

```
def set_numba_threads():
    os.environ["NUMBA_NUM_THREADS"] = "1"

client.run(set_numba_threads)
```

6.14.5 Adding a Logger

A custom logger and file handler can be setup in a similar way to the environment variables. The log levels of other loggers can also be specified, such as *Numba* in the following example.

```
def setup_logging():
    logging.basicConfig(
        filename="path/to/log/file",
        filemode='a',
        format='%(asctime)s,%(msecs)d %(name)s %(levelname)s %(message)s',
        datefmt='%H:%M:%S',
        level=logging.DEBUG,
    )
    logging.getLogger("numba").setLevel(logging.WARNING)

client.run(setup_logging)
```

6.14.6 Resqpy Wrapper Functions

To run the multiprocessing function, a wrapper function for the corresponding resqpy function is required. These can be found within the `multi_processing.wrappers` module. Currently there is only a wrapper function for the `find_faces_to_represent_surface_regular` function, however any wrapper function can be created, providing that it returns the following:

- index (*int*): the index passed to the function.
- success (*bool*): whether the function call was successful, whatever that definiton is.
- epc_file (*str*): the epc file path where the objects are stored.
- uuid_list (*List[str]*): list of UUIDs of relevant objects.

The multiprocessing function will combine all of the objects that have their UUIDs returned, into a single epc file.

6.14.7 Calling the Multiprocessing Function

The multiprocessing function must receive the following arguments:

- function (*Callable*): the wrapper function to be called, that must return the items described above.
- kwargs_list (*List[Dict[Any]]*): A list of keyword argument dictionaries that are used when calling the function.
- recombined_epc (*Path/str*): A pathlib Path or path string of where the combined epc will be saved.
- cluster (*LocalCluster/JobQueueCluster*): the relevant cluster, as explained above.
- consolidate (*bool*): if True and an equivalent part already exists in a model, it is not duplicated and the uuids are noted as equivalent.

```
from resqpy.multi_processing import function_multiprocessing

success_list = function_multiprocessing(func, kwargs_list, recombined_epc,␣
↪cluster=cluster)
```

A list of successes from the wrapper function in order of their call is returned.

Note: the `resqpy.multi_processing` sub-package was previously named `resqpy.multiprocessing`. The name was change with major release v4.0.0 in order to avoid potential namespace clashes with the standard python multiprocessing package.

CHAPTER

SEVEN

API REFERENCE

RESQML manipulation library.

model	Model class, roughly equivalent to a RESQML epc file.
crs	RESQML coordinate reference systems.
derived_model	Creating a derived resqml model from an existing one; mostly grid manipulations.
fault	Grid Connection Set class and related functions.
grid	The Grid Module.
grid_surface	Classes for RESQML objects related to surfaces.
lines	Polyline and PolylineSet classes and associated functions.
multi_processing	Multiprocessing module for running specific functions concurrently.
organize	Organizational object classes: features and interpretations.
property	Collections of properties for grids, wellbore frames, grid connection sets etc.
rq_import	Miscellaneous functions for importing from other formats.
strata	Stratigraphy related classes and valid values.
surface	Classes for RESQML objects related to surfaces.
time_series	Time series classes and functions.
unstructured	Unstructured grid and derived classes.
weights_and_measures	Weights and measures valid units and unit conversion functions.
well	Classes relating to wells.
olio	Low level supporting modules, mostly providing functions rather than classes.

7.1 resqpy.model

Model class, roughly equivalent to a RESQML epc file.

Classes

Model	Class for RESQML (v2) based models.
ModelContext	Context manager for easy opening and closing of resqpy models.

7.1.1 resqpy.model.Model

class resqpy.model.**Model**(*epc_file: Optional[str] = None*, *full_load: bool = True*, *epc_subdir: Optional[Union[str, Iterable]] = None*, *new_epc: bool = False*, *create_basics: Optional[bool] = None*, *create_hdf5_ext: Optional[bool] = None*, *copy_from: Optional[str] = None*, *quiet=False*)

Bases: `object`

Class for RESQML (v2) based models.

Examples

To open an existing dataset:

```
Model(epc_file = 'filename.epc')
```

To create a new, empty model ready to populate:

```
Model(epc_file = 'new_file.epc', new_epc = True, create_basics = True, create_hdf5_
↪ext = True)
```

Alternatively, use the module level convenience function:

```
new_model(epc_file = 'new_file.epc')
```

To copy an existing dataset then open the new copy:

```
Model(epc_file = 'new_file.epc', copy_from = 'existing.epc')
```

Public Data Attributes:

crs_root	XML node corresponding to self.crs_uuid, the 'main' crs for the model.

Commonly Used Methods:

__init__([epc_file, full_load, epc_subdir, ...])	Create an empty model; load it from epc_file if given.
parts([parts_list, obj_type, uuid, title, ...])	Returns a list of parts matching all of the arguments passed.
part([parts_list, obj_type, uuid, title, ...])	Returns the name of a part matching all of the arguments passed.
uuids([parts_list, obj_type, uuid, title, ...])	Returns a list of uuids of parts matching all of the arguments passed.
uuid([parts_list, obj_type, uuid, title, ...])	Returns the uuid of a part matching all of the arguments passed.
roots([parts_list, obj_type, uuid, title, ...])	Returns a list of xml root nodes of parts matching all of the arguments passed.
root([parts_list, obj_type, uuid, title, ...])	Returns the xml root node of a part matching all of the arguments passed.
titles([parts_list, obj_type, uuid, title, ...])	Returns a list of citation titles of parts matching all of the arguments passed.
title([parts_list, obj_type, uuid, title, ...])	Returns the citation title of a part matching all of the arguments passed.
store_epc([epc_file, main_xml_name, ...])	Write xml parts of model to epc file (HDF5 arrays are not written here).
part_for_uuid(uuid)	Returns the part name which has the given uuid.
root_for_uuid(uuid)	Returns the xml root for the part which has the given uuid.
uuid_for_part(part_name[, is_rels])	Returns the uuid for the named part.
type_of_part(part_name[, strip_obj])	Returns content type for the named part (does not apply to rels parts).
type_of_uuid(uuid[, strip_obj])	Returns content type for the uuid.
root_for_part(part_name[, is_rels])	Returns root of parsed xml tree for the named part.
citation_title_for_part(part)	Returns the citation title for the specified part.
grid([title, uuid, find_properties])	Returns a shared Grid (or RegularGrid) object for this model, by default the 'main' grid.
h5_release()	Releases (closes) the currently open hdf5 file.
title_for_root([root])	Returns the Title text from the Citation within the given root node.
title_for_part(part_name)	Returns the Title text from the Citation for the given main part name (not for rels).
create_unknown([root])	Creates an Unknown node and optionally adds as child of root.
iter_objs(cls)	Iterate over all available objects of given resqpy class within the model.
iter_wellbore_interpretations()	Iterable of all WellboreInterpretations associated with the model.
iter_trajectories()	Iterable of all trajectories associated with the model.
iter_md_datums()	Iterable of all MdDatum objects associated with the model.
iter_crs()	Iterable of all CRS objects associated with the model.
as_graph([uuids_subset])	Return representation of model as nodes and edges, suitable for plotting in a graph.

Methods:

initialize()	Set model contents to empty.
uuid_is_present(uuid)	Returns True if the uuid is present in the model's catalogue, False otherwise.
set_modified()	Marks the model as having been modified and assigns a new uuid.
uuids_as_int_related_to_uuid(uuid)	Returns set of ints being uuids of objects related to uuid by any category of relationship.
uuids_as_int_referenced_by_uuid(uuid)	Returns set of ints being uuids of objects which uuid has a reference to.
uuids_as_int_referencing_uuid(uuid)	Returns set of ints being uuids of objects which have a reference to uuid.
uuids_as_int_softly_related_to_uuid(uuid)	Returns set of ints being uuids of objects related to uuid by only a soft relationship.
create_tree_if_none()	Checks that model has an xml tree; if not, an empty tree is created; not usually called directly.
load_part(epc, part_name[, is_rels])	Load and parse xml tree for given part name, storing info in parts forest (or rels forest).
set_epc_file_and_directory(epc_file)	Sets the full path and directory of the epc_file.
fell_part(part_name)	Removes the named part from the in-memory parts forest.
remove_part_from_main_tree(part)	Removes the named part from the main (Content_Types) tree.
tidy_up_forests([tidy_main_tree, ...])	Removes any parts that do not have any related data in dictionaries.
load_epc(epc_file[, full_load, epc_subdir, ...])	Load xml parts of model from epc file (HDF5 arrays are not loaded).
parts_list_of_type([type_of_interest, uuid])	Returns a list of part names for parts of type of interest, optionally matching a uuid.
list_of_parts([only_objects])	Return a complete list of parts.
number_of_parts()	Retuns the number of parts in the model, including external parts such as the link to an hdf5 file.
parts_count_by_type([type_of_interest])	Returns a sorted list of (type, count) for parts.
parts_count_dict()	Returns a dictionary mapping from RESQML object type to count of parts.
parts_list_filtered_by_related_uuid(...[, ...])	From a list of parts, returns a list of those parts which have a relationship with the given uuid.
supporting_representation_for_part(part)	Returns the uuid of the supporting representation for the part, if found, otherwise None.
parts_list_filtered_by_supporting_uuid(...)	From a list of parts, returns a list of those parts which have the given uuid as supporting representation.
parts_list_related_to_uuid_of_type(uuid[, ...])	Returns a list of parts of type of interest that relate to part with given uuid.
external_parts_list()	Returns a list of part names for external part references.
uuid_for_root(root_node)	Returns the uuid for an object given an xml root node.
tree_for_part(part_name[, is_rels])	Returns parsed xml tree for the named part.
change_hdf5_uuid_in_hdf5_references(node, ...)	Scan node for hdf5 references and set the uuid of the hdf5 file itself to new_uuid.

continues on next page

Table 1 – continued from previous page

change_uuid_in_hdf5_references(node, ...)	Scan node for hdf5 references using the old_uuid and replace with the new_uuid.
change_uuid_in_supporting_representation_reference(...)	Look for supporting representation reference using the old_uuid and replace with the new_uuid.
change_filename_in_hdf5_rels([new_hdf5_filename])	Scan relationships forest for hdf5 external parts and patch in a new filename.
copy_part(existing_uuid, new_uuid[, ...])	Makes a new part as a copy of an existing part with only a new uuid set; the new part can then be modified.
root_for_ijk_grid([uuid, title])	Return root for IJK Grid part.
root_for_time_series([uuid])	Return root for time series part.
resolve_grid_root([grid_root, uuid])	If grid root argument is None, returns the root for the IJK Grid part instead.
add_grid(grid_object[, check_for_duplicates])	Add grid object to list of shareable grids for this model.
grid_list_uuid_list()	Returns list of uuid's for the grid objects in the cached grid list.
grid_for_uuid_from_grid_list(uuid)	Returns the cached grid object matching the given uuid, if found in the grid list, otherwise None.
resolve_time_series_root([time_series_root])	If time_series_root is None, finds the root for a time series in the model.
h5_set_default_override(override)	Sets the default hdf5 filename override mode for the model.
h5_uuid_and_path_for_node(node[, tag])	Returns a (hdf5_uuid, hdf5_internal_path) pair for an xml array node.
h5_uuid_list(node)	Returns a list of all uuids for hdf5 external part(s) referred to in recursive tree.
h5_uuid()	Returns the uuid of the 'main' hdf5 file.
h5_file_name([uuid, override, file_must_exist])	Returns full path for hdf5 file with given uuid.
h5_access([uuid, mode, override, file_path])	Returns an open h5 file handle for the hdf5 file with the given uuid.
h5_array_shape_and_type(h5_key_pair)	Returns the shape and dtype of the array, as stored in the hdf5 file.
h5_array_element(h5_key_pair[, index, ...])	Returns one element from an hdf5 array and/or caches the array.
h5_array_slice(h5_key_pair, slice_tuple)	Loads a slice of an hdf5 array.
h5_overwrite_array_slice(h5_key_pair, ...)	Overwrites (updates) a slice of an hdf5 array.
h5_clear_filename_cache()	Clears the cached filenames associated with all ext uuids.
create_root()	Initialises an empty main xml tree for model.
add_part(content_type, uuid, root[, ...])	Adds a (recently created) node as a new part in the model's parts forest.
patch_root_for_part(part, root)	Updates the xml tree for the part without changing the uuid.
remove_part(part_name[, ...])	Removes a part from the parts forest; optionally remove corresponding rels part and other relationships.
new_obj_node(flavour[, name_space, ...])	Creates a new main object element and sets attributes (does not add children).
referenced_node(ref_node[, consolidate])	For a given xml reference node, returns the node for the object referred to, if present.
create_ref_node(flavour, title, uuid[, ...])	Create a reference node, optionally add to root.

continues on next page

Table 1 – continued from previous page

uom_node(root, uom)	Add a generic unit of measure sub element to root.
create_rels_part()	Adds a relationships reference node as a new part in the model's parts forest.
create_citation([root, title, originator])	Creates a citation xml node and optionally appends as a child of root.
create_doc_props([add_as_part, root, originator])	Creates a document properties stub node and optionally adds as child of root and/or to parts forest.
create_crs_reference([root, crs_uuid])	Creates a node refering to an existing crs node and optionally adds as child of root.
create_md_datum_reference(md_datum_root[, root])	Creates a node refering to an existing measured depth datum and optionally adds as child of root.
create_hdf5_ext([add_as_part, root, title, ...])	Creates an hdf5 external node and optionally adds as child of root and/or to parts forest.
create_hdf5_dataset_ref(hdf5_uuid, ...[, title])	Creates a pair of nodes referencing an hdf5 dataset (array) and adds to root.
create_supporting_representation([...])	Craate a supporting representation reference node refering to an IjkGrid and optionally add to root.
create_source(source[, root])	Create an extra meta data node holding information on the source of the data, optionally add to root.
create_patch(p_uuid[, ext_uuid, root, ...])	Create a node for a patch of values, including ref to hdf5 data set, optionally add to root.
create_time_series_ref(time_series_uuid[, root])	Create a reference node to a time series, optionally add to root.
create_solitary_point3d(flavour, root, xyz)	Creates a subelement to root for a solitary point in 3D space.
create_reciprocal_relationship(node_a, ...)	Adds a node to each of a pair of trees in the rels forest, to represent a two-way relationship.
create_reciprocal_relationship_uuids(uuid_a, ...)	Adds a node to each of a pair of trees in the rels forest, to represent a two-way relationship.
duplicate_node(existing_node[, add_as_part])	Creates a deep copy of the xml node (typically from another model) and optionally adds as part.
force_consolidation_uuid_equivalence(...)	Force immigrant object to be teated as equivalent to resident during consolidation.
force_consolidation_equivalence_for_class_type(c)	Force immigrant objects of type(c) to be teated as equivalent where only extra metadata differs during consolidation.
remove_extra_metadata(uuid)	Removes extra metadata from in memory xml for uuid.
copy_part_from_other_model(other_model, part)	Fully copies part in from another model, with referenced parts, hdf5 data and relationships.
copy_uuid_from_other_model(other_model, uuid)	Fully copies part for uuid in from another model, with referenced parts, hdf5 data and relationships.
copy_all_parts_from_other_model(other_model)	Fully copies parts in from another model, with referenced parts, hdf5 data and relationships.
iter_grid_connection_sets()	Yields grid connection set objects, one for each gcs in this model.
sort_parts_list_by_timestamp(parts_list)	Returns a copy of the parts list sorted by citation block creation date, with the newest first.
check_catalogue_dictionaries([...])	Checks internal consistency of catalogue dictionaries, raising assertion exception if inconsistent.

__init__(*epc_file: Optional[str] = None*, *full_load: bool = True*, *epc_subdir: Optional[Union[str, Iterable]] = None*, *new_epc: bool = False*, *create_basics: Optional[bool] = None*, *create_hdf5_ext: Optional[bool] = None*, *copy_from: Optional[str] = None*, *quiet=False*)

Create an empty model; load it from epc_file if given.

Note: if epc_file is given and the other arguments indicate that it will be a new dataset (new_epc is True or copy_from is given) then any existing .epc and .h5 file(s) with this name will be deleted immediately

---

Parameters

- **epc_file** (*str, optional*) – if present, and new_epc is False and copy_from is None, the name of an existing epc file which is opened, unzipped and parsed to determine the list of parts and relationships comprising the model; if present, and new_epc is True or copy_from is specified, the name of a new file to be created - any existing file (and .h5 paired hdf5 file) will be immediately deleted if None, an empty model is created (ie. with no parts) unless copy_from is present
- **full_load** (*boolean*) – only relevant if epc_file is not None and new_epc is False (or copy_from is specified); if True (recommended), the xml for each part is parsed and stored in a tree structure in memory; if False, only the list of parts is loaded
- **epc_subdir** (*string or list of strings, optional*) – if present, parts are only included in the load if they are in the top level directory of the epc internal structure, or in the specified subdirectory (or one of the subdirectories in the case of a list); only relevant if epc_file is not None and new_epc is False (or copy_from is specified)
- **new_epc** (*boolean, default False*) – if True, a new model is created, empty unless copy_from is given
- **create_basics** (*boolean, optional*) – if True and epc_file is None or new_epc is True, then the minimum essential parts are added to the empty Model; this is equivalent to calling the create_root(), create_rels_part() and create_doc_props() methods; if None, defaults to the same value as new_epc
- **create_hdf5_ref** (*boolean, optional*) – if True and new_epc is True and create_basics is True and epc_file is not None, then an hdf5 external part is created, equivalent to calling the create_hdf5_ext() method; an empty hdf5 file is also created; if None, defaults to same value as new_epc
- **copy_from** – (str, optional): if present, and epc_file is also present, then the epc file named in copy_from, together with its paired h5 file, are copied to epc_file (overwriting any previous instances) before epc_file is opened; this argument is primarily to facilitate repeated testing of code that modifies the resqml dataset, eg. by appending new parts
- **quiet** (*boolean, default False*) – if True, reading and saving info logging messages are suppressed

Returns

The newly created Model object

add_grid(*grid_object*, *check_for_duplicates=False*)

Add grid object to list of shareable grids for this model.

Parameters

- **grid_object** (*grid.Grid object*) – the ijk grid object to be added to the list of grids in the model

- **check_for_duplicates** (*boolean, default False*) – if True, a check is made for any grid objects already in the grid list with the same root as the new grid object

Returns
None

add_part(*content_type*, *uuid*, *root*, *add_relationship_part=True*, *epc_subdir=None*)

Adds a (recently created) node as a new part in the model's parts forest.

Parameters

- **content_type** (*string*) – the resqml object class of the new part
- **uuid** (*uuid.UUID*) – the uuid for the new part
- **root** – the root node of the xml tree for the new part
- **add_relationship_part** – (boolean, default True): if True, a relationship part is also created to go with the new part (empty of actual relationships)
- **epc_subdir** (*string, optional*) – if present, the subdirectory path within the epc that the part is to be located within; if None, the xml will reside at the top level of the epc

Returns
None

Notes

NB: xml tree for part is not written to epc file by this function (store_epc() handles that); do not use this function for the main rels extension part: use create_rels_part() instead

as_graph(*uuids_subset=None*)

Return representation of model as nodes and edges, suitable for plotting in a graph.

Note: The graph can be most readily visualised with other packages such as NetworkX and HoloViews, which are not part of resqpy.

For a guide to plotting graphs interactively, see: http://holoviews.org/user_guide/Network_Graphs.html

---

example:

```
# Create the nodes and edges
nodes, edges = model.as_graph()

# Load into a NetworkX graph
import networkx as nx
g = nx.Graph()
g.add_nodes_from(nodes.items())
g.add_edges_from(edges)

# Import holoviews
import holoviews as hv
from holoviews import opts
hv.extension('bokeh')

# Plot
hv.Graph.from_networkx(g, nx.layout.spring_layout).opts(
```

(continues on next page)

(continued from previous page)

```
    tools=['hover'], node_color='resqml_type', cmap='Category10'
)
```

Parameters

uuids_subset (*iterable*) – If present, only consider uuids in this list. By default, use all uuids in the model.

Returns

2-tuple of nodes and edges –

- nodes: dict mapping uuid to attributes (e.g. citation title)
- edges: set of unordered pairs of uuids, representing relationships

change_filename_in_hdf5_rels(*new_hdf5_filename=None*)

Scan relationships forest for hdf5 external parts and patch in a new filename.

Parameters

new_hdf5_filename – the new filename to patch into the xml; if None, the epc filename is used with no directory path and with the extension changed to .h5

Returns

None

Notes

no check is made that the new filename is for an existing file; all hdf5 file references will be modified

change_hdf5_uuid_in_hdf5_references(*node*, *old_uuid*, *new_uuid*)

Scan node for hdf5 references and set the uuid of the hdf5 file itself to new_uuid.

Parameters

- **node** – the root node of an xml tree within which hdf5 internal paths are to have hdf5 uuids changed
- **old_uuid** (*uuid.UUID or str*) – the ext uuid currently to be found in the hdf5 references; if None, all will be replaced
- **new_uuid** (*uuid.UUID or str*) – the new ext (hdf5) uuid to replace the old one

Returns

None

Note: use this method when the uuid of the hdf5 ext part is changing; if the uuid of the high level part itself is changing use change_uuid_in_hdf5_references() instead

---

change_uuid_in_hdf5_references(*node*, *old_uuid*, *new_uuid*)

Scan node for hdf5 references using the old_uuid and replace with the new_uuid.

Parameters

- **node** – the root node of an xml tree within which hdf5 internal paths are to have uuids changed
- **old_uuid** (*uuid.UUID or str*) – the uuid currently to be found in the hdf5 references

- **new_uuid** (*uuid.UUID or str*) – the new uuid to replace the old one

Returns
None

Notes

use this method when the uuid of the high level part itself is changing; if the uuid of the hdf5 ext part itself is changing, use change_hdf5_uuid_in_hdf5_references() instead; this method does not modify the internal path names in the hdf5 file itself, if that has already been written

change_uuid_in_supporting_representation_reference(*node*, *old_uuid*, *new_uuid*, *new_title=None*)

Look for supporting representation reference using the old_uuid and replace with the new_uuid.

Parameters

- **node** – the root node of an xml tree within which the supporting representation uuid is to be changed
- **old_uuid** (*uuid.UUID or str*) – the uuid currently to be found in the supporting representation reference
- **new_uuid** (*uuid.UUID or str*) – the new uuid to replace the old one
- **new_title** (*string, optional*) – if present, the title stored in the xml reference block is changed to this

Returns
boolean – True if the change was carried out; False otherwise

Notes

this method is typically used to temporarily set a supporting representation to a locally mocked representation object when the actual supporting representation is not present in the dataset

check_catalogue_dictionaries(*referred_parts_must_be_present=True*, *check_xml=True*)

Checks internal consistency of catalogue dictionaries, raising assertion exception if inconsistent.

Parameters

- **referred_parts_must_be_present** (*bool, default True*) – if True, raises an exception if a referenced part is not present in the model (such a scenario is allowed by the RESQML standard)
- **check_xml** (*bool, default True*) – if True, xml is scoured to check that references are consistent with the Model uuid_rels_dict internal dictionary

Note: this is a thorough but slow check, use sparing; primarily intended for unit tests and debugging

citation_title_for_part(*part*)

Returns the citation title for the specified part.

copy_all_parts_from_other_model(*other_model*, *realization=None*, *consolidate=True*)

Fully copies parts in from another model, with referenced parts, hdf5 data and relationships.

Parameters

- **model** (*other*) – the source model from which to copy parts
- **realization** (*int, optional*) – if present, the realization attribute of property parts will be set to this value, instead of the value in use in the other model if any
- **consolidate** (*boolean, default True*) – if True, where equivalent part already exists in this model, do not duplicate but instead note uuids as equivalent, modifying references and relationships of other copied parts appropriately

Notes

part names already existing in this model are not duplicated; default hdf5 file used in this model and assumed in other_model

copy_part(*existing_uuid*, *new_uuid*, *change_hdf5_refs=False*)

Makes a new part as a copy of an existing part with only a new uuid set; the new part can then be modified.

Parameters

- **existing_uuid** (*uuid.UUID*) – the uuid of the existing part
- **new_uuid** (*uuid.UUID*) – the uuid to inject into the new part after copying of the xml tree
- **change_hdf5_refs** (*boolean*) – if True, the new tree is scanned for hdf5 refs using the existing_uuid and they are replaced with the new_uuid

Returns

string being the new part name

Notes

Resqml objects have a unique identifier and should be considered immutable; therefore to modify an object, it should first be duplicated; this function does some of the xml work needed for such duplication: the xml tree is copied; the uuid attribute in the root node is changed; the new part is added to the parts forest (with its name matched to the new uuid); NB: relationships are not currently copied or modified; also note that hdf5 data and high level objects maintained by other modules are not duplicated here; use this method to duplicate a part within a model prior to modifying the duplicated part in some way; to import a part from another model, use copy_part_from_other_model() instead; for copying a grid it is best to use the higher level derived_model.copy_grid() function

copy_part_from_other_model(*other_model*, *part*, *realization=None*, *consolidate=True*, *force=False*, *cut_refs_to_uuids=None*, *cut_node_types=None*, *self_h5_file_name=None*, *h5_uuid=None*, *other_h5_file_name=None*, *uuid_int=None*)

Fully copies part in from another model, with referenced parts, hdf5 data and relationships.

Parameters

- **model** (*other*) – the source model from which to copy a part
- **part** (*string*) – the part name in the other model to copy into this model
- **realization** (*int, optional*) – if present and the part is a property, the realization will be set to this value, instead of the value in use in the other model if any

- **consolidate** (*boolean, default True*) – if True and an equivalent part already exists in this model, do not duplicate but instead note uuids as equivalent
- **force** (*boolean, default False*) – if True, the part itself is copied without much checking and all references are required to be handled by an entry in the consolidation object
- **cut_refs_to_uuids** (*list of UUIDs, optional*) – if present, then xml reference nodes referencing any of the listed uuids are cut out in the copy; use with caution
- **cut_node_types** (*list of str, optional*) – if present, any child nodes of a type in the list will be cut out in the copy; use with caution
- **self_h5_file_name** (*string, optional*) – h5 file name for this model; can be passed as an optimisation when calling method repeatedly
- **h5_uuid** (*uuid, optional*) – UUID for this model's hdf5 external part; can be passed as an optimisation when calling method repeatedly
- **other_h5_file_name** (*string, optional*) – h5 file name for other model; can be passed as an optimisation when calling method repeatedly
- **uuid_int** (*int, optional*) – if present, the uuid (as int) of part; if uuid already established use this argument as an optimisation; note: no checks for consistency are made here

Returns

the part name of the part in this model, after copying; may differ from requested part if consolidate is True; None in the case of failure

Notes

if the part name already exists in this model, no action is taken; default hdf5 file used in this model and assumed in other_model

copy_uuid_from_other_model(*other_model*, *uuid*, *realization=None*, *consolidate=True*, *force=False*, *cut_refs_to_uuids=None*, *cut_node_types=None*, *self_h5_file_name=None*, *h5_uuid=None*, *other_h5_file_name=None*)

Fully copies part for uuid in from another model, with referenced parts, hdf5 data and relationships.

Parameters

- **model** (*other*) – the source model from which to copy a part
- **uuid** (*UUID*) – the uuid of the part in the other model to copy into this model
- **realization** (*int, optional*) – if present and the part is a property, the realization will be set to this value, instead of the value in use in the other model if any
- **consolidate** (*boolean, default True*) – if True and an equivalent part already exists in this model, do not duplicate but instead note uuids as equivalent
- **force** (*boolean, default False*) – if True, the part itself is copied without much checking and all references are required to be handled by an entry in the consolidation object
- **cut_refs_to_uuids** (*list of UUIDs, optional*) – if present, then xml reference nodes referencing any of the listed uuids are cut out in the copy; use with caution
- **cut_node_types** (*list of str, optional*) – if present, any child nodes of a type in the list will be cut out in the copy; use with caution

- **self_h5_file_name** (*string, optional*) – h5 file name for this model; can be passed as an optimisation when calling method repeatedly
- **h5_uuid** (*uuid, optional*) – UUID for this model's hdf5 external part; can be passed as an optimisation when calling method repeatedly
- **other_h5_file_name** (*string, optional*) – h5 file name for other model; can be passed as an optimisation when calling method repeatedly

Returns
the uuid of the part in this model, after copying; may differ from requested uuid if consolidate is True; None in the case of failure

Notes

if the part already exists in this model, no action is taken; default hdf5 file used in this model and assumed in other_model

create_citation(*root=None*, *title=''*, *originator=None*)

Creates a citation xml node and optionally appends as a child of root.

Parameters

- **root** (*optional*) – if not None, the newly created citation node is appended as a child to this node
- **title** (*string*) – the citation title: a human readable string; this is the main point of having a citation node, so the argument should be used wisely
- **originator** (*string, optional*) – the name of the human being who created the object which this citation is for; default is to use the login name

Returns
newly created citation xml node

create_crs_reference(*root=None*, *crs_uuid=None*)

Creates a node refering to an existing crs node and optionally adds as child of root.

Parameters

- **root** – the xml node to which the new reference node is to appended as a child (ie. the xml node for the object that is referring to the crs)
- **crs_uuid** – the uuid of the crs

Returns
newly created crs reference xml node

create_doc_props(*add_as_part=True*, *root=None*, *originator=None*)

Creates a document properties stub node and optionally adds as child of root and/or to parts forest.

Parameters

- **add_as_part** (*boolean, default True*) – if True, the newly created node is also added as a part
- **root** (*optional, usually None*) – if not None, the newly created node is appended to this root as a child
- **originator** (*string, optional*) – used as the creator in the doc props node; if None, the login name is used

Returns
the newly created doc props xml node

Note: the doc props part of a resqml dataset is intended to hold documentation and other stuff that is not covered by the standard; there should be exactly one doc props part

---

`create_hdf5_dataset_ref`(*hdf5_uuid*, *object_uuid*, *group_tail*, *root*, *title='Hdf Proxy'*)

Creates a pair of nodes referencing an hdf5 dataset (array) and adds to root.

Parameters

- **hdf5_uuid** (*uuid.UUID*) – the uuid of the hdf5 external part being referenced
- **object_uuid** (*uuid.UUID*) – the uuid of the high level object (part) which owns the hdf5 array being referenced
- **group_tail** (*string*) – the tail of the hdf5 internal path, which is appended to the part name section of the internal path
- **root** – the xml node to which the newly created hdf5 reference is appended as a child
- **title** (*string*) – used as the Title text in the citation node, usually left at the default 'Hdf Proxy'

Returns
the newly created xml node holding the hdf5 internal path

`create_hdf5_ext`(*add_as_part=True*, *root=None*, *title='Hdf Proxy'*, *originator=None*, *file_name=None*, *uuid=None*)

Creates an hdf5 external node and optionally adds as child of root and/or to parts forest.

Parameters

- **add_as_part** (*boolean, default True*) – if True the newly created ext node is added to the model as a part
- **root** (*optional, usually None*) – if not None, the newly created ext node is appended as a child of this node
- **title** (*string*) – used as the Title text in the citation node, usually left at the default 'Hdf Proxy'
- **originator** (*string, optional*) – the name of the human being who created the ext object; default is to use the login name
- **file_name** (*string, optional*) – the filename to be stored as the Target in the relationship node; if None, will default to the epc filename with the extenstion replaced with .h5
- **uuid** (*uuid.UUID, optional*) – the ext uuid tp associate with the external part; if None, a new uuid will be generated

Returns
newly created hdf5 external part xml node

Note: this method is typically called when creating a new dataset (Model); if the intention is to share an existing hdf5 file, then pass the file_name and (ext) uuid arguments; if the intention is to create a new hdf5 file amongst many used by the Model, then pass the file_name

---

create_md_datum_reference(*md_datum_root*, *root=None*)

Creates a node refering to an existing measured depth datum and optionally adds as child of root.

Parameters

- **md_datum_root** – the root xml node for the measured depth datum being referenced
- **root** – the xml node to which the new reference node is to appended as a child (ie. the xml node for the object that is referring to the md datum)

Returns

newly created measured depth datum reference xml node

create_patch(*p_uuid*, *ext_uuid=None*, *root=None*, *patch_index=0*, *hdf5_type='DoubleHdf5Array'*, *xsd_type='double'*, *null_value=None*, *const_value=None*, *const_count=None*, *points=False*)

Create a node for a patch of values, including ref to hdf5 data set, optionally add to root.

Parameters

- **p_uuid** (*uuid.UUID*) – the uuid of the object for which this patch is a component
- **ext_uuid** (*uuid.UUID*) – the uuid of the hdf5 external part holding the array; required unless const_value and const_count are not None
- **root** – if not None, the newly created patch of values xml node is appended as a child to this node
- **patch_index** (*int, default 0*) – the patch index number; patches must be numbered sequentially starting at 0
- **hdf5_type** (*string, default 'DoubleHdf5Array'*) – the type of the hdf5 array; usually one of 'DoubleHdf5Array', 'IntegerHdf5Array', or 'BooleanHdf5Array'; replaced with equivalent constant array type if const_value is not None
- **xsd_type** (*string, default 'double'*) – the xsd simple type of each element of the array
- **null_value** – Used in a null value sub-node to specify what value in an array of discrete data represents null; if None, a value of -1 is used for signed integers, 2^32 - 1 for uints (even 64 bit uints!)
- **const_value** (*float, int, or bool, optional*) – if not None, the patch is created as a constant array; const_count must also be present if const_value is not None
- **const_count** (*int, optional*) – the number of elements in (size of) the constant array; required if const_value is not None, ignored otherwise
- **points** (*bool, default False*) – if True, the created node will be for a patch of points, otherwise a patch of values

Returns

newly created xml node for the patch of values

Note: this function does not write the data to the hdf5; that should be done separately before calling this method; RESQML usually stores array data in the hdf5 file however constant arrays are flagged as such in the xml and no data is stored in the hdf5

---

create_reciprocal_relationship(*node_a*, *rel_type_a*, *node_b*, *rel_type_b*, *avoid_duplicates=True*)

Adds a node to each of a pair of trees in the rels forest, to represent a two-way relationship.

Parameters

- **node_a** – one of the two xml nodes to be related
- **rel_type_a** (*string*) – the Type (role) associated with node_a in the relationship; usually 'sourceObject' or 'destinationObject'
- **node_b** – the other xml node to be related
- **rel_type_b** (*string*) – the Type (role) associated with node_b in the relationship usually 'sourceObject' or 'destinationObject' (opposite of rel_type_a)
- **avoid_duplicates** (*boolean, default True*) – if True, xml for a relationship is not added where it already exists; if False, a duplicate will be created in this situation

Returns
None

Note: this method has the same effect as create_reciprocal_relationship_uuids() but takes xml root nodes rather than uuids as arguments

---

create_reciprocal_relationship_uuids(*uuid_a*, *rel_type_a*, *uuid_b*, *rel_type_b*, *avoid_duplicates=True*)

Adds a node to each of a pair of trees in the rels forest, to represent a two-way relationship.

Parameters

- **uuid_a** – uuid of one of the two parts to be related
- **rel_type_a** (*string*) – the Type (role) associated with uuid_a in the relationship; usually 'sourceObject' or 'destinationObject'
- **uuid_b** – uuid of the other part to be related
- **rel_type_b** (*string*) – the Type (role) associated with uuid_b in the relationship usually 'sourceObject' or 'destinationObject' (opposite of rel_type_a)
- **avoid_duplicates** (*boolean, default True*) – if True, xml for a relationship is not added where it already exists; if False, a duplicate will be created in this situation

Returns
None

Note: this method has the same effect as create_reciprocal_relationship() but takes uuids rather than xml root nodes as arguments

---

create_ref_node(*flavour*, *title*, *uuid*, *content_type=None*, *root=None*)

Create a reference node, optionally add to root.

Parameters

- **flavour** (*string*) – the resqml object class (type of part) for which a reference node is required
- **title** (*string*) – used as the Title subelement text in the reference node
- **uuid** – (uuid.UUID): the uuid of the part being referenced
- **content_type** (*string, optional*) – if None, the referenced content type is determined from the flavour argument (recommended)

- **root** (*optional*) – if not None, an xml node to which the reference node is appended as a child

Returns
newly created reference xml node

create_rels_part()

Adds a relationships reference node as a new part in the model's parts forest.

Returns
newly created main relationships reference xml node

Note: there can only be one relationships reference part in the model

create_root()

Initialises an empty main xml tree for model.

Note: not usually called directly

create_solitary_point3d(*flavour*, *root*, *xyz*)

Creates a subelement to root for a solitary point in 3D space.

Parameters

- **flavour** (*string*) – the object class (type) of the point node to be created
- **root** – the xml node to which the newly created solitary point node is appended as a child
- **xyz** (*triple float*) – the x, y, z coordinates of the solitary point

Returns
the newly created xml node for the solitary point

create_source(*source*, *root=None*)

Create an extra meta data node holding information on the source of the data, optionally add to root.

Parameters

- **source** (*string*) – text describing the source of an object
- **root** – if not None, the newly created extra metadata node is appended as a child of this node

Returns
the newly created extra metadata xml node

create_supporting_representation(*grid_root=None*, *support_uuid=None*, *root=None*, *title=None*, *content_type='obj_IjkGridRepresentation'*)

Craate a supporting representation reference node refering to an IjkGrid and optionally add to root.

Parameters

- **grid_root** – the xml node of the grid (or other) object which is the supporting representation being referred to; could also be used for other classes of supporting object; this or support_uuid must be provided
- **support_uuid** – the uuid of the grid (or other supporting representation) being referred to; this or grid_root must be provided

- **root** – if not None, the newly created supporting representation node is appended as a child to this node
- **title** – the Title to use in the supporting representation node
- **content_type** – the resqml object class of the supporting representation being referenced; defaults to 'obj_IjkGridRepresentation'

Returns
newly created xml node for supporting representation reference

Notes

a property array needs a supporting representation which is the structure that the property values belong to; for example, a grid property array has the grid object as the supporting representation; one of grid_root or support_uuid should be passed when calling this method

create_time_series_ref(*time_series_uuid*, *root=None*)

Create a reference node to a time series, optionally add to root.

Parameters

- **time_series_uuid** (*uuid.UUID*) – the uuid of the time series part being referenced
- **root** (*optional*) – if present, the newly created time series reference xml node is added as a child to this node

Returns
the newly created time series reference xml node

create_tree_if_none()

Checks that model has an xml tree; if not, an empty tree is created; not usually called directly.

create_unknown(*root=None*)

Creates an Unknown node and optionally adds as child of root.

Parameters
root (*optional*) – if present, the newly created Unknown node is appended as a child of this xml node

Returns
the newly created Unknown xml node

property crs_root

XML node corresponding to self.crs_uuid, the 'main' crs for the model.

duplicate_node(*existing_node*, *add_as_part=True*)

Creates a deep copy of the xml node (typically from another model) and optionally adds as part.

Parameters

- **existing_node** – the existing xml node, usually in another model, to be duplicated
- **add_as_part** (*boolean, default True*) – if True, the newly created xml node is added as a part in this model

Returns
the newly created duplicate xml node

Notes

hdf5 data is not copied by this function and any reference to hdf5 arrays will be naively duplicated; the uuid of the part is not changed by this function, so if the source and target models are the same, add as part should be set False and calling code will need to assign a new uuid prior to adding as part; if add_as_part is True, the part is only added if the uuid does not already exist in this model

external_parts_list()

Returns a list of part names for external part references.

Returns
list of strings being the part names for external part references

---

Note: in practice, external part references are only used for hdf5 files; furthermore, all current datasets have adopted the practice of using a single hdf5 file for a given epc file

---

fell_part(*part_name*)

Removes the named part from the in-memory parts forest.

Parameters
part_name (*string*) – the name of the part to be removed

Note: no check is made for references or relationships to the part being deleted; not usually called directly

---

force_consolidation_equivalence_for_class_ignoring_extra_metadata(*other_model*, *resqpy_class*)

Force immigrant objects of type to be teated as equivalent where only extra metadata differs during consolidation.

Notes

this method should be called prior to calling copy_part_from_other_model() or copy_uuid_from_other_model() to override the more stringent equivalence checks which include extra metadata; the resqpy class must have an is_equivalent() method which supports the check_extra_metadata boolean argument; typically used for organisational classes such as features and interpretations; an exception is raised if more than one matching part already exists in this model, for a particular immigrant title (for the object type)

force_consolidation_uuid_equivalence(*immigrant_uuid*, *resident_uuid*)

Force immigrant object to be teated as equivalent to resident during consolidation.

grid(*title=None*, *uuid=None*, *find_properties=True*)

Returns a shared Grid (or RegularGrid) object for this model, by default the 'main' grid.

Parameters

- **title** (*string, optional*) – if present, the citation title of the IjkGridRepresentation
- **uuid** (*uuid.UUID, optional*) – if present, the uuid of the IjkGridRepresentation
- **find_properties** (*boolean, default True*) – if True, the property_collection attribute of the returned grid object will be populated

Returns
grid.Grid object for the specified grid or the main ijk grid part in this model

Note: if neither title nor uuid are given, the model should contain just one grid, or a grid named 'ROOT'; unlike most classes of object, a central list of resqpy Grid objects can be maintained within a Model by using this method which will return a shared object from this list, instantiating a new object and adding it to the list when necessary; an assertion exception will be raised if a suitable grid part is not present in the model

grid_for_uuid_from_grid_list(*uuid*)

Returns the cached grid object matching the given uuid, if found in the grid list, otherwise None.

grid_list_uuid_list()

Returns list of uuid's for the grid objects in the cached grid list.

h5_access(*uuid=None*, *mode='r'*, *override='default'*, *file_path=None*)

Returns an open h5 file handle for the hdf5 file with the given uuid.

Parameters

- **uuid** (*uuid.UUID*) – the uuid of the hdf5 external part reference for which the open file handle is required; required if override is False and file_path is None
- **mode** (*string*) – the hdf5 file mode ('r', 'w' or 'a') with which to open the file
- **override** (*str or bool, default 'default'*) – if str, one of 'default', 'none', 'dir' or 'full'; if bool (deprecated), False means 'dir' and True means 'full'; if 'default', the default h5 override mode for the model is used; if 'dir', any directory in the rels hdf5 file name is replaced with the epc's directory; if 'full', the hdf5 full path is generated by using the epc path but replacing the .epc extension with .h5
- **file_path** (*string, optional*) – if present, is used as the hdf5 file path, otherwise the path will be determined based on the uuid and override arguments

Returns

a file handle to the opened hdf5 file

Note: an exception will be raised if the hdf5 file cannot be opened; note that sometimes another piece of code accessing the file might cause a 'resource unavailable' exception

h5_array_element(*h5_key_pair*, *index=None*, *cache_array=False*, *object=None*, *array_attribute=None*, *dtype='float'*, *required_shape=None*)

Returns one element from an hdf5 array and/or caches the array.

Parameters

- **h5_key_pair** (*uuid.UUID, string*) – the uuid of the hdf5 external part reference and the hdf5 internal path for the array
- **index** (*pair or triple int, optional*) – if None, the only purpose of the call is to ensure that the array is cached in memory; if not None, the (k0, pillar_index) or (k0, j0, i0) index of the cell for which the value is required
- **cache_array** (*boolean, default False*) – if True, a copy of the whole array is cached in memory as an attribute of the object; if already cached, the array is not uncached, regardless of this argument

- **object** (*optional, defaults to self*) – the object in which a cached version of the array is an attribute, or will be created as an attribute if cache_array is True
- **array_attribute** (*string*) – the attribute name to use for the cached version of the array, required to cache or access cached array
- **dtype** (*string or data type*) – the data type of the elements of the array (need not match hdf5 array in precision)
- **required_shape** (*tuple of ints, optional*) – if not None, the hdf5 array will be reshaped to this shape; if index is not None, it is taken to be applicable to the required shape; required if the array is bool data that was written with resqpy specific dtype of 'pack'

Returns

if index is None, then None; if index is not None, then the value of the array for the cell identified by index

---

Note: this function can be used to access an individual element from an hdf5 array, or to cache a whole array in memory; when accessing an individual element, the index style must match the array indexing; in particular for IJK grid points, a (k0, pillar_index) is needed when the grid has split pillars, whereas a (k0, j0, i0) is needed when the grid does not have any split pillars

---

h5_array_shape_and_type(*h5_key_pair*)

Returns the shape and dtype of the array, as stored in the hdf5 file.

Parameters

h5_key_pair (*uuid.UUID, string*) – the uuid of the hdf5 external part reference and the hdf5 internal path for the array

Returns

(tuple of ints, type) – simply the shape and dtype attributes of the referenced hdf5 array; (None, None) is returned if the hdf5 file is not found, or the array is not found within it

h5_array_slice(*h5_key_pair*, *slice_tuple*)

Loads a slice of an hdf5 array.

Parameters

- **h5_key_pair** (*uuid, string*) – the uuid of the hdf5 ext part and the hdf5 internal path to the required hdf5 array
- **slice_tuple** (*tuple of slice objects*) – each element should be constructed using the python built-in function slice()

Returns

numpy array that is a hyper-slice of the hdf5 array, with the same ndim as the source hdf5 array

Notes

this method always fetches from the hdf5 file and does not attempt local caching; the whole array is not loaded; all axes continue to exist in the returned array, even where the sliced extent of an axis is 1

h5_clear_filename_cache()

Clears the cached filenames associated with all ext uuids.

h5_file_name(*uuid=None*, *override='default'*, *file_must_exist=True*)

Returns full path for hdf5 file with given uuid.

Parameters

- **uuid** (*uuid.UUID, optional*) – the uuid of the hdf5 external part reference for which the file name is required; if None, the 'main' hdf5 uuid is used
- **override** (*str or bool, default 'default'*) – if str, one of 'default', 'none', 'dir' or 'full'; if bool (deprecated), False means 'dir' and True means 'full'; if 'default', the default h5 override mode for the model is used; if 'dir', any directory in the rels hdf5 file name is replaced with the epc's directory; if 'full', the hdf5 full path is generated by using the epc path but replacing the .epc extension with .h5
- **file_must_exist** (*boolean, default True*) – if True, the existence of the hdf5 file is checked
- **found** (*upon first call and a FileNotFound exception is raised if the file is not*) –

Returns

string being the full path of the hdf5 file

Notes

in practice, a resqml model usually consists of a pair of files in the same directory, with names like: a.epc and a.h5; to allow copying, moving and renaming of files, the practical approach is simply to assume a one-to-one correspondence between epc and hdf5 files, and assume they are in the same directory, which will be the default behaviour of resqpy; to change the default behaviour for the model, call the h5_set_default_override() method; an hdf5 file name is cached once determined for a given ext uuid; to clear the cache, call the h5_clear_filename_cache() method

h5_overwrite_array_slice(*h5_key_pair*, *slice_tuple*, *array_slice*)

Overwrites (updates) a slice of an hdf5 array.

Parameters

- **h5_key_pair** (*uuid, string*) – the uuid of the hdf5 ext part and the hdf5 internal path to the required hdf5 array
- **slice_tuple** (*tuple of slice objects*) – each element should be constructed using the python built-in function slice()
- **array_slice** (*numpy array of shape to match slice_tuple*) – the data to write

Notes

this method naively updates a slice in an hdf5 array without using mpi to look after parallel updates; metadata (such as uuid or property min, max values) is not modified in any way by the method

h5_release()

Releases (closes) the currently open hdf5 file.

Returns
None

h5_set_default_override(*override*)

Sets the default hdf5 filename override mode for the model.

Parameters
override (*str*) – 'none', 'dir' or 'full' being the override mode to use by default

---

Note: this mode will be used by default when determining a filename for accessing hdf5 data; see h5_file_name() notes for more information

---

h5_uuid()

Returns the uuid of the 'main' hdf5 file.

h5_uuid_and_path_for_node(*node*, *tag='Values'*)

Returns a (hdf5_uuid, hdf5_internal_path) pair for an xml array node.

Parameters

- **node** – xml node for which the array reference is required
- **tag** (*string, default 'Values'*) – the tag of the child of node for which the array reference is required

Returns
(uuid.UUID, string) pair being the uuid of the hdf5 external part reference and the hdf5 internal path for the array of interest

---

Note: this method provides the key data needed to actually access array data within the resqml dataset

---

h5_uuid_list(*node*)

Returns a list of all uuids for hdf5 external part(s) referred to in recursive tree.

initialize()

Set model contents to empty.

Note: not usually called directly (semi-private)

---

iter_crs()

Iterable of all CRS objects associated with the model.

Yields
crs – instance of `resqpy.crs.CRS`

iter_grid_connection_sets()

Yields grid connection set objects, one for each gcs in this model.

iter_md_datums()

Iterable of all MdDatum objects associated with the model.

Yields
md_datum – instance of `resqpy.well.MdDatum`

iter_objs(*cls*)

Iterate over all available objects of given resqpy class within the model.

Note: The resqpy class must expose a class attribute *resqml_type*, and must support being created with the signature: *obj = cls(model, uuid=uuid)*.

example use:

```
for well in model.iter_objs(cls=resqpy.well.WellboreFeature):
   print(well.title, well.uuid)
```

Parameters
cls – resqpy class to iterate

Yields
list of instances of cls

iter_trajectories()

Iterable of all trajectories associated with the model.

Yields
trajectory – instance of `resqpy.well.Trajectory`

iter_wellbore_interpretations()

Iterable of all WellboreInterpretations associated with the model.

Yields
wellbore – instance of `resqpy.organize.WellboreInterpretation`

list_of_parts(*only_objects=True*)

Return a complete list of parts.

load_epc(*epc_file*, *full_load=True*, *epc_subdir=None*, *copy_from=None*, *quiet=False*)

Load xml parts of model from epc file (HDF5 arrays are not loaded).

Parameters

- **epc_file** (*string*) – the path of the epc file
- **full_load** (*boolean, default True*) – if True (recommended), the xml for each part is parsed and stored in a tree structure in memory; if False, only the list of parts is loaded
- **epc_subdir** (*string or list of strings, optional*) – if present, only parts in the top level directory within the epc structure, or in the specified subdirectory(ies) are included in the load
- **copy_from** (*string, optional*) – if present, the .epc and .h5 are copied from this source to epc_file (and paired .h5) prior to opening epc_file; any previous files named as epc_file will be overwritten
- **quiet** (*boolean, default False*) – if True, info logging message is emitted as debug

Returns
None

Note: when copy_from is specified, the entire contents of the source dataset are copied, regardless of the epc_subdir setting which only affects the subsequent load into memory

---

`load_part`(*epc*, *part_name*, *is_rels=None*)

Load and parse xml tree for given part name, storing info in parts forest (or rels forest).

Parameters

- **epc** – an open ZipFile handle for the epc file
- **part_name** (*string*) – the name of the 'file' within the epc bundle containing the part (or relationship)
- **is_rels** – (boolean, optional): if True, the part to be loaded is a relationship part; if False, it is a main part; if None, its value is derived from the part name

Returns
boolean – True if part loaded successfully, False if part failed to load

---

Note: parts forest must already have been initialized before calling this method; if False is returned, calling code should probably delete part from forest; not usually called directly

---

`new_obj_node`(*flavour*, *name_space='resqml2'*, *is_top_lvl_obj=True*)

Creates a new main object element and sets attributes (does not add children).

Parameters

- **flavour** (*string*) – the resqml object class (type of part) for which a new xml tree is required
- **name_space** (*string, default 'resqml2'*) – the xml namespace identifier to use for the node
- **is_top_lvl_obj** (*boolean, default True*) – if True, the xsi:type is set in the xml node, as required for top level objects (parts); if False, the type atttribute is not set

Returns
newly created root node for xml tree for flavour of object, without any children

`number_of_parts`()

Retuns the number of parts in the model, including external parts such as the link to an hdf5 file.

`part`(*parts_list=None*, *obj_type=None*, *uuid=None*, *title=None*, *title_mode='is'*, *title_case_sensitive=False*, *metadata={}*, *extra={}*, *related_uuid=None*, *related_mode=None*, *epc_subdir=None*, *multiple_handling='exception'*)

Returns the name of a part matching all of the arguments passed.

Parameters

- **parts** (*(as for)*) –
- **multiple_handling** (*string, default 'exception'*) – one of 'exception', 'none', 'first', 'oldest', 'newest'

Returns
string being the part name of the single part matching all of the criteria, or None

Notes

this method can be used where a single part is being identified; if no parts match the criteria, None is returned; if more than one part matches the criteria, the multiple_handling argument determines what happens: 'exception' causes a ValueError exception to be raised; 'none' causes None to be returned; 'first' causes the first part (as stored in the epc file or added) to be returned; 'oldest' causes the part with the oldest creation timestamp in the citation block to be returned; 'newest' causes the newest part to be returned

part_for_uuid(*uuid*)

Returns the part name which has the given uuid.

Parameters

uuid (*uuid.UUID object or string*) – the uuid of the part of interest

Returns

a string being the part name which matches the uuid, or None if not found

parts(*parts_list=None*, *obj_type=None*, *uuid=None*, *title=None*, *title_mode='is'*, *title_case_sensitive=False*, *metadata={}*, *extra={}*, *related_uuid=None*, *related_mode=None*, *epc_subdir=None*, *sort_by=None*)

Returns a list of parts matching all of the arguments passed.

Parameters

- **parts_list** (*list of strings, optional*) – if present, an 'input' list of parts to be filtered; if None then all the parts in the model are considered
- **obj_type** (*string, optional*) – if present, only parts of this resqml type will be included
- **uuid** (*uuid.UUID, optional*) – if present, the uuid of a part to select
- **title** (*string, optional*) – if present, a citation title or substring to filter on, based on the title_mode argument
- **title_mode** (*string, default 'is'*) – one of 'is', 'starts', 'ends', 'contains', 'is not', 'does not start', 'does not end', or 'does not contain'; how to compare each part's citation title with the title argument; ignored if title is None
- **title_case_sensitive** (*boolean, default False*) – if True, title comparisons are made on a case sensitive basis; otherwise comparisons are insensitive to case
- **key** (*extra (dictionary of*) – value pairs, optional): if present, only parts which have within their top level metadata all the items in this argument, are included in the filtered list
- **key** – value pairs, optional): if present, only parts which have within their extra metadata all the items in this argument, are included in the filtered list
- **related_uuid** (*uuid.UUID, optional*) – if present, only parts which are related to this uuid are included in the filtered list
- **related_mode** (*Optional[int]*) – if provided, filters by the type of relationship. 0 is parts referenced by related_uuid, 1 is parts that reference related_uuid, 2 is other soft related parts.
- **epc_subdir** (*string, optional*) – if present, only parts which reside within the specified subdirectory path of the epc are included in the filtered list
- **sort_by** (*string, optional*) – one of 'newest', 'oldest', 'title', 'uuid', 'type'

Returns
a list of strings being the names of parts which match all filter arguments

Examples

a full list of parts in the model:

```
model.parts()
```

a list of IjkGrid parts:

```
model.parts(obj_type = 'IjkGridRepresentation')
```

a list containing the part name for a uuid:

```
model.parts(uuid = 'a869e7cc-5d30-4b31-8502-c74b1d87c777')
```

a list of IjkGrid parts with titles beginning LGR, sorted by title:

```
model.parts(obj_type = 'IjkGridRepresentation', title = 'LGR', title_mode =
↪'starts', sort_by = 'title')
```

parts_count_by_type(*type_of_interest=None*)

Returns a sorted list of (type, count) for parts.

Parameters
type_of_interest (*string, optional*) – if not None, the returned list only contains one pair, with a count for that type, ie. resqml object class

Returns
list of pairs, each being (string, int) representing part type, ie. resqml object class, without leading obj underscore, and count

parts_count_dict()

Returns a dictionary mapping from RESQML object type to count of parts.

Returns
dict mapping string to int with key being RESQML class and value being count of number of parts

Note: only RESQML classes with at least one object present in the model will be included in the dictionary

parts_list_filtered_by_related_uuid(*parts_list*, *uuid*, *uuid_is_source=None*, *related_mode=None*)

From a list of parts, returns a list of those parts which have a relationship with the given uuid.

Parameters

- **parts_list** (*list of strings*) – input list of parts from which a selection is made
- **uuid** (*uuid.UUID*) – the uuid of a part for which related parts are required
- **uuid_is_source** (*boolean, default None*) – if None, relationships in either direction qualify; if True, only those where uuid is sourceObject qualify; if False, only those where uuid is destinationObject qualify

- **related_mode** (*Optional[int]*) – if provided, filters by the type of relationship. 0 is parts referenced by this uuid, 1 is parts that reference this uuid, 2 is other soft related parts.

Returns
list of strings being the subset of parts_list which are related to the object with the given uuid

---

Note: the part to which the given uuid applies might or might not be in the input parts list; this method scans the relationship info for every present part, looking for uuid in rels

---

parts_list_filtered_by_supporting_uuid(*parts_list*, *uuid*)

From a list of parts, returns a list of those parts which have the given uuid as supporting representation.

Parameters

- **parts_list** (*list of strings*) – input list of parts from which a selection is made
- **uuid** (*uuid.UUID*) – the uuid of a supporting representation part for which related parts are required

Returns
list of strings being the subset of parts_list which have as their supporting representation the object with the given uuid

---

Note: the part to which the given uuid applies might or might not be in the input parts list

---

parts_list_of_type(*type_of_interest=None*, *uuid=None*)

Returns a list of part names for parts of type of interest, optionally matching a uuid.

Parameters

- **type_of_interest** (*string*) – the resqml object class of interest, in string form, eg. 'obj_IjkGridRepresentation'
- **uuid** (*uuid.UUID object, optional*) – if present, only a part with this uuid is included in the list

Returns
a list of strings being the part names which match the arguments

---

Note: usually either a type of interest or a uuid is passed; if neither are passed, all parts are returned; this method is maintained for backward compatibility and for efficiency reasons; it is equivalent to: self.parts(obj_type = type_of_interest, uuid = uuid)

---

parts_list_related_to_uuid_of_type(*uuid*, *type_of_interest=None*)

Returns a list of parts of type of interest that relate to part with given uuid.

Parameters

- **uuid** (*uuid.UUID*) – the uuid of a part for which related parts are required
- **type_of_interest** (*string*) – the type of parts (resqml object class) of the related parts of interest

Returns
list of strings being the part names of the type of interest, related to the uuid

patch_root_for_part(*part*, *root*)

Updates the xml tree for the part without changing the uuid.

referenced_node(*ref_node*, *consolidate=False*)

For a given xml reference node, returns the node for the object referred to, if present.

Note: if consolidating and an equivalent referenced object exists, the uuid in the ref_node is modified by this method; it does not update entries in the uuid_rels_dict

remove_extra_metadata(*uuid*)

Removes extra metadata from in memory xml for uuid.

Note: this method will not modify any resqpy objects already instantiated

remove_part(*part_name*, *remove_relationship_part=True*)

Removes a part from the parts forest; optionally remove corresponding rels part and other relationships.

remove_part_from_main_tree(*part*)

Removes the named part from the main (Content_Types) tree.

Note: not usually called directly

resolve_grid_root(*grid_root=None*, *uuid=None*)

If grid root argument is None, returns the root for the IJK Grid part instead.

Parameters

- **grid_root** (*optional*) – if not None, this method simply returns this argument
- **uuid** (*uuid.UUID, optional*) – if present, the uuid of the ijk grid part for which the root is required; if None, an ijk grid part is sought and the root for that part is returned

Returns

root node in xml tree for the ijk grid part in this model ('ROOT' grid if more than one present)

Notes

if grid_root and uuid are both None and there are multiple grids in the model, the oldest grid with a citation title of 'ROOT' will be returned; an exception will be raised if no grid part is present in the model

resolve_time_series_root(*time_series_root=None*)

If time_series_root is None, finds the root for a time series in the model.

Parameters

time_series_root (*optional*) – if not None, this method simply returns this argument

Returns

root node in xml tree for the time series part in this model, or None if there is no time series part

Note: an assertion exception will be raised if time_series_root is None and there is more than one time series part in the model

root(*parts_list=None*, *obj_type=None*, *uuid=None*, *title=None*, *title_mode='is'*, *title_case_sensitive=False*, *metadata={}*, *extra={}*, *related_uuid=None*, *related_mode=None*, *epc_subdir=None*, *multiple_handling='exception'*)

Returns the xml root node of a part matching all of the arguments passed.

Parameters

part (*(as for)* –

Returns

lxml.etree.Element object being the root node of the xml for the single part matching all of the criteria, or None

root_for_ijk_grid(*uuid=None*, *title=None*)

Return root for IJK Grid part.

Parameters

- **uuid** (*uuid.UUID, optional*) – if present, the uuid of the ijk grid part for which the root is required; if None, a single ijk grid part is expected and the root for that part is returned
- **title** (*string, optional*) – if present, the citation title for the grid; defaults to 'ROOT' if more than one ijk grid present and no uuid supplied

Returns

root node in xml tree for the ijk grid part in this model

Notes

if uuid and title are both supplied, they must match in the corresponding grid part; if a title but no uuid is given, the first ijk grid encountered that has a matching title will be returned; if neither title nor uuid are given, the first ijk grid with title 'ROOT' will be returned, unless there is only one grid part in which case the root npde for that part is returned regardless; failure to find a matching grid part results in an assertion exception

root_for_part(*part_name*, *is_rels=None*)

Returns root of parsed xml tree for the named part.

Parameters

- **part_name** (*string*) – the part name for which the root of the xml tree is required
- **is_rels** (*boolean, optional*) – if True, the part is a relationship part; if False, it is a main part; if None, its value is determined from the part name

Returns

root node of the parsed xml tree (defined in lxml or ElementTree package) for the named part

root_for_time_series(*uuid=None*)

Return root for time series part.

argument:

uuid (uuid.UUID, optional): if present, the uuid of the time series part for which the root is required;
if None, a single time series part is expected and the root for that part is returned

Returns

root node in xml tree for the time series part in this model

Note: if no uuid is given and the model contains more than one time series, the one with the earliest creation date is returned

root_for_uuid(*uuid*)

Returns the xml root for the part which has the given uuid.

Parameters
uuid (*uuid.UUID object or string*) – the uuid of the part of interest

Returns
the xml root node for the part with the given uuid, or None if not found

roots(*parts_list=None*, *obj_type=None*, *uuid=None*, *title=None*, *title_mode='is'*, *title_case_sensitive=False*, *metadata={}*, *extra={}*, *related_uuid=None*, *related_mode=None*, *epc_subdir=None*, *sort_by=None*)

Returns a list of xml root nodes of parts matching all of the arguments passed.

Parameters
parts (*(as for)*) –

Returns
list of lxml.etree.Element objects

set_epc_file_and_directory(*epc_file*)

Sets the full path and directory of the epc_file.

Parameters
epc_file (*string*) – the path of the epc file

Note: not usually needed to be called directly, except perhaps when creating a new dataset

set_modified()

Marks the model as having been modified and assigns a new uuid.

Note: this modification tracking functionality is not part of the resqml standard and is only loosely applied by the library code; not usually called directly

sort_parts_list_by_timestamp(*parts_list*)

Returns a copy of the parts list sorted by citation block creation date, with the newest first.

store_epc(*epc_file=None*, *main_xml_name='[Content_Types].xml'*, *only_if_modified=False*, *quiet=False*)

Write xml parts of model to epc file (HDF5 arrays are not written here).

Parameters

- **epc_file** (*string*) – the name of the output epc file to be written (any existing file will be overwritten)
- **main_xml_name** (*string, do not pass*) – this argument should not be passed as the resqml standard requires the default value; (the argument exists in code because the resqml standard value is based on a slight misunderstanding of the opc standard, so could perhaps change in future versions of resqml)
- **only_if_modified** (*boolean, default False*) – if True, the epc file is only written if the model is flagged as having been modified (at least one part added or removed)

- **quiet** (*boolean, default False*) – if True, info logging is emitted at debug level

Returns
None

Note: the main tree, parts forest and rels forest must all be up to date before calling this method

supporting_representation_for_part(*part*)

Returns the uuid of the supporting representation for the part, if found, otherwise None.

tidy_up_forests(*tidy_main_tree=True*, *tidy_others=False*, *remove_extended_core=True*)

Removes any parts that do not have any related data in dictionaries.

Note: not usually called directly

title(*parts_list=None*, *obj_type=None*, *uuid=None*, *title=None*, *title_mode='is'*, *title_case_sensitive=False*, *metadata={}*, *extra={}*, *related_uuid=None*, *related_mode=None*, *epc_subdir=None*, *multiple_handling='exception'*)

Returns the citation title of a part matching all of the arguments passed.

Parameters
part (*(as for)*) –

Returns
string being the citation title of the single part matching all of the criteria, or None

title_for_part(*part_name*)

Returns the Title text from the Citation for the given main part name (not for rels).

Parameters
part_name (*string*) – the name of the part for which the citation title is required

Returns
string being the Title text from the citation node which is a child of the root xml node for the part, or None if not found

title_for_root(*root=None*)

Returns the Title text from the Citation within the given root node.

Parameters
root – the xml node for the object for which the citation title is required

Returns
string being the Title text from the citation node which is a child of root, or None if not found

titles(*parts_list=None*, *obj_type=None*, *uuid=None*, *title=None*, *title_mode='is'*, *title_case_sensitive=False*, *metadata={}*, *extra={}*, *related_uuid=None*, *related_mode=None*, *epc_subdir=None*, *sort_by=None*)

Returns a list of citation titles of parts matching all of the arguments passed.

Parameters
parts (*(as for)*) –

Returns
list of strings being the citation titles of matching parts

tree_for_part(*part_name*, *is_rels=None*)

Returns parsed xml tree for the named part.

Parameters

- **part_name** (*string*) – the part name for which the xml tree is required
- **is_rels** (*boolean, optional*) – if True, the part is a relationship part; if False, it is a main part; if None, its value is determined from the part name

Returns
parsed xml tree (defined in lxml or ElementTree package) for the named part

type_of_part(*part_name*, *strip_obj=False*)

Returns content type for the named part (does not apply to rels parts).

Parameters

- **part_name** (*string*) – the part for which the type is required
- **strip_obj** (*boolean, default False*) – if True, the leading obj and underscore is removed from the returned string

Returns
string being the type (resqml object class) for the named part

type_of_uuid(*uuid*, *strip_obj=False*)

Returns content type for the uuid.

Parameters

- **uuid** (*uuid.UUID or str*) – the uuid for which the type is required
- **strip_obj** (*boolean, default False*) – if True, the leading obj and underscore is removed from the returned string

Returns
string being the type (resqml object class) for the named part

uom_node(*root*, *uom*)

Add a generic unit of measure sub element to root.

Parameters

- **root** – xml node to which unit of measure subelement (child) will be added
- **uom** (*string*) – the resqml unit of measure

Returns
newly created unit of measure node (having already been added to root)

Note: does not currently check that uom is a valid Energistics unit of measure; use weights_and_measures module functionality to check if needed

---

uuid(*parts_list=None*, *obj_type=None*, *uuid=None*, *title=None*, *title_mode='is'*, *title_case_sensitive=False*, *metadata={}*, *extra={}*, *related_uuid=None*, *related_mode=None*, *epc_subdir=None*, *multiple_handling='exception'*)

Returns the uuid of a part matching all of the arguments passed.

Parameters
part ((*as for*) –

Returns
uuid of the single part matching all of the criteria, or None

uuid_for_part(*part_name*, *is_rels=None*)

Returns the uuid for the named part.

Parameters

- **part_name** (*string*) – the part name for which the uuid is required
- **is_rels** (*boolean, optional*) – if True, the part is a relationship part; if False, it is a main part; if None, its value is determined from the part name

Returns
uuid.UUID for the specified part

Note: this method will fail with an exception if the part is not in this model; a quicker alternative to this method is simply to extract the uuid from the part name using olio.xml_et.uuid_in_part_name()

---

uuid_for_root(*root_node*)

Returns the uuid for an object given an xml root node.

Parameters
root_node – the xml root node for the object for which the uuid is required

Returns
uuid.UUID for the specified object

uuid_is_present(*uuid*)

Returns True if the uuid is present in the model's catalogue, False otherwise.

uuids(*parts_list=None*, *obj_type=None*, *uuid=None*, *title=None*, *title_mode='is'*, *title_case_sensitive=False*, *metadata={}*, *extra={}*, *related_uuid=None*, *related_mode=None*, *epc_subdir=None*, *sort_by=None*)

Returns a list of uuids of parts matching all of the arguments passed.

Parameters
parts ((*as for*) –

Returns
list of uuids

uuids_as_int_referenced_by_uuid(*uuid*)

Returns set of ints being uuids of objects which uuid has a reference to.

Note: this method returns a set of ints; use olio.uuid.uuid_from_int() to get a UUID object

---

uuids_as_int_referencing_uuid(*uuid*)

Returns set of ints being uuids of objects which have a reference to uuid.

Note: this method returns a set of ints; use olio.uuid.uuid_from_int() to get a UUID object

---

uuids_as_int_related_to_uuid(*uuid*)

Returns set of ints being uuids of objects related to uuid by any category of relationship.

Note: this method returns a set of ints; use olio.uuid.uuid_from_int() to get a UUID object

uuids_as_int_softly_related_to_uuid(*uuid*)

Returns set of ints being uuids of objects related to uuid by only a soft relationship.

Note: resqpy uses the term 'soft relationship' for those relationships held in the _rels xml area but not as reference nodes in the main xml of either part involved in the relationship; the Model.create_reciprocal_relationship() and create_reciprocal_relationship_uuid() methods can be used by application code to create such soft relationships; this method returns a set of ints; use olio.uuid.uuid_from_int() to get a UUID object

7.1.2 resqpy.model.ModelContext

class resqpy.model.**ModelContext**(*epc_file*, *mode='r'*)

Bases: `object`

Context manager for easy opening and closing of resqpy models.

When a model is opened this way, any open file handles are safely closed when the "with" clause exits. Optionally, the epc can be written back to disk upon exit.

Example:

```
with ModelContext("my_model.epc", mode="rw") as model:
    print(model.uuids())
```

Note: The "write_hdf5" and "create_xml" methods of individual resqpy objects still need to be invoked as usual.

Methods:

__init__(epc_file[, mode])	Open a resqml file, safely closing file handles upon exit.
__enter__()	Enter the runtime context, return a model.
__exit__(exc_type, exc_value, exc_tb)	Exit the runtime context, close the model.

__init__(*epc_file*, *mode='r'*) → None

Open a resqml file, safely closing file handles upon exit.

Parameters

- **epc_file** (*str*) – path to existing resqml file
- **mode** (*str, default 'r'*) – one of "read", "read/write", "create", or shorthands "r", "rw", "c"

Notes

the modes operate as follows: - In "read" mode, an existing epc file is opened; any changes are not saved to disk automatically, but can still be saved by calling *model.store_epc()*; - In "read/write" mode, changes are written to disk when the context exists; - In "create" mode, a new model is created and saved upon exit; any pre-existing model will be deleted

Functions

new_model	Returns a new, empty Model object with basics and hdf5 ext part set up.

7.1.3 resqpy.model.new_model

resqpy.model.**new_model**(*epc_file*, *quiet=False*) → *Model*

Returns a new, empty Model object with basics and hdf5 ext part set up.

7.2 resqpy.crs

RESQML coordinate reference systems.

Classes

Crs	Coordinate reference system object.

7.3 resqpy.derived_model

Creating a derived resqml model from an existing one; mostly grid manipulations.

Functions

add_edges_per_column_property_array	Adds an edges per column grid property from a numpy array to an existing resqml dataset.
add_faults	Extends epc file with a new grid which is a version of the source grid with new curtain fault(s) added.
add_one_blocked_well_property	Adds a blocked well property from a numpy array to an existing resqml dataset.
add_one_grid_property_array	Adds a grid property from a numpy array to an existing resqml dataset.
add_single_cell_grid	Creates a model with a single cell IJK Grid, with a cuboid cell aligned with x,y,z axes, enclosing points.
add_wells_from_ascii_file	Adds new md datum, trajectory, interpretation and feature objects for each well in a tabular ascii file..
add_zone_by_layer_property	Adds a discrete zone property (and local property kind) with indexable element of layers.
coarsened_grid	Generates a coarsened version of an unsplit source grid, optionally inheriting properties.
copy_grid	Creates a copy of the IJK grid object in the target model (usually prior to modifying points in situ).
drape_to_surface	Return a new grid with geometry draped to a surface.
extract_box	Extends an existing model with a new grid extracted as a logical IJK box from the source grid.
extract_box_for_well	Extends an existing model with a new grid extracted as an IJK box around a well trajectory in the source grid.
fault_throw_scaling	Extends epc with a new grid with fault throws multiplied by scaling factors.
gather_ensemble	Creates a composite resqml dataset by merging all parts from all models in list, assigning realization numbers.
global_fault_throw_scaling	Rewrites epc with a new grid with all the fault throws multiplied by the same scaling factor.
interpolated_grid	Extends an existing model with a new grid geometry linearly interpolated between the two source_grids.
local_depth_adjustment	Applies a local depth adjustment to the grid, adding as a new grid part in the model.
refined_grid	Generates a refined version of the source grid, optionally inheriting properties.
single_layer_grid	Extends an existing model with a new version of the source grid converted to a single, thick, layer.
tilted_grid	Extends epc file with a new grid which is a version of the source grid tilted.
unsplit_grid	Extends epc file with a new grid which is a version of the source grid with all faults healed.
zonal_grid	Extends an existing model with a new version of the source grid converted to a single, thick, layer per zone.
zone_layer_ranges_from_array	Returns a list of (zone_min_k0, zone_max_k0, zone_index) derived from zone_array.

7.3.1 resqpy.derived_model.add_edges_per_column_property_array

resqpy.derived_model.**add_edges_per_column_property_array**(*epc_file*, *a*, *property_kind*, *grid_uuid=None*, *source_info='imported'*, *title=None*, *discrete=False*, *uom=None*, *time_index=None*, *time_series_uuid=None*, *string_lookup_uuid=None*, *null_value=None*, *facet_type=None*, *facet=None*, *realization=None*, *local_property_kind_uuid=None*, *extra_metadata={}*, *new_epc_file=None*)

Adds an edges per column grid property from a numpy array to an existing resqml dataset.

Parameters

- **epc_file** (*string*) – file name to load model resqml model from (and rewrite to if new_epc_file is None)
- **a** (*3D numpy array*) – the property array to be added to the model; expected shape (nj,ni,2,2) or (nj,ni,4)
- **property_kind** (*string*) – the resqml property kind
- **grid_uuid** (*uuid object or string, optional*) – the uuid of the grid to which the property relates; if None, the property is attached to the 'main' grid
- **source_info** (*string*) – typically the name of a file from which the array has been read but can be any information regarding the source of the data
- **title** (*string*) – this will be used as the citation title when a part is generated for the array; for simulation models it is desirable to use the simulation keyword when appropriate
- **discrete** (*boolean, default False*) – if True, the array should contain integer (or boolean) data; if False, float
- **uom** (*string, default None*) – the resqml units of measure for the data; not relevant to discrete data
- **time_index** (*integer, default None*) – if not None, the time index to be used when creating a part for the array
- **time_series_uuid** (*uuid object or string, default None*) – required if time_index is not None
- **string_lookup_uuid** (*uuid object or string, optional*) – required if the array is to be stored as a categorical property; set to None for non-categorical discrete data; only relevant if discrete is True
- **null_value** (*int, default None*) – if present, this is used in the metadata to indicate that this value is to be interpreted as a null value wherever it appears in the data (use for discrete data only)
- **facet_type** (*string*) – resqml facet type, or None
- **facet** (*string*) – resqml facet, or None
- **realization** (*int*) – realization number, or None
- **local_property_kind_uuid** (*uuid.UUID or string*) – uuid of local property kind, or None

- **extra_metadata** (*dict, optional*) – any items in this dictionary are added as extra metadata to the new property
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new property object; if present, a new epc file (& associated h5 file) is created to contain a copy of the grid and the new property

Returns
uuid.UUID - the uuid of the newly created property

Notes

the RESQML protocol for saving edges per column properties uses a clockwise ordering of the 4 edges of a column; the resqpy protocol uses 2 dimensions of extent 2, being the axis (J, I) and face (-, +); this function assumes the array is in RESQML protocol if it has shape (nj, ni, 4) and resqpy protocol if it has shape (nj, ni, 2, 2); when reloading the property it will be presented in RESQML protocol; calling code can use property module functions reformat_column_edges_from_resqml_format() and reformat_column_edges_to_resqml_format() to convert between the protocols if needed

7.3.2 resqpy.derived_model.add_faults

resqpy.derived_model.**add_faults**(*epc_file*, *source_grid*, *polylines=None*, *lines_file_list=None*, *lines_crs_uuid=None*, *full_pillar_list_dict=None*, *left_right_throw_dict=None*, *create_gcs=True*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *new_grid_title=None*, *new_epc_file=None*)

Extends epc file with a new grid which is a version of the source grid with new curtain fault(s) added.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **polylines** (*lines.PolylineSet or list of lines.Polyline, optional*) – list of poly lines for which curtain faults are to be added; either this or lines_file_list or full_pillar_list_dict must be present
- **lines_file_list** (*list of str, optional*) – a list of file paths, each containing one or more poly lines in simple ascii format§; see notes; either this or polylines or full_pillar_list_dicr must be present
- **lines_crs_uuid** (*uuid, optional*) – if present, the uuid of a coordinate reference system with which to interpret the contents of the lines files; if None, the crs used by the grid will be assumed
- **full_pillar_list_dict** (*dict mapping str to list of pairs of ints, optional*) – dictionary mapping from a fault name to a list of pairs of ints being the ordered neigbouring primary pillar (j0, i0) defining the curtain fault; either this or polylines or lines_file_list must be present
- **left_right_throw_dict** (*dict mapping str to pair of floats, optional*) – dictionary mapping from a fault name to a pair of floats being the semi-throw adjustment on the left and the right of the fault (see notes); semi-throw values default to (+0.5, -0.5)

- **create_gcs** (*boolean, default True*) – if True, and faults are being defined by lines, a grid connection set is created with one feature per new fault and associated organisational objects are also created; ignored if lines_file_list is None
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the unsplit grid (& crs)

Returns

a new grid (grid.Grid object) which is a copy of the source grid with the structure modified to incorporate the new faults

Notes

full_pillar_list_dict is typically generated by Grid.make_face_sets_from_pillar_lists(); pillars will be split as needed to model the new faults, though existing splits will be used as appropriate, so this function may also be used to add a constant to the throw of existing faults; the left_right_throw_dict contains a pair of floats for each fault name (as found in keys of full_pillar_list_dict); these throw values are lengths in the uom of the crs used by the grid (which must have the same xy units as z units);

this function does not add a GridConnectionSet to the model – calling code may wish to do that

7.3.3 resqpy.derived_model.add_one_blocked_well_property

resqpy.derived_model.**add_one_blocked_well_property**(*epc_file*, *a*, *property_kind*, *blocked_well_uuid*, *source_info='imported'*, *title=None*, *discrete=False*, *uom=None*, *time_index=None*, *time_series_uuid=None*, *string_lookup_uuid=None*, *null_value=None*, *indexable_element='cells'*, *facet_type=None*, *facet=None*, *realization=None*, *local_property_kind_uuid=None*, *count_per_element=1*, *points=False*, *extra_metadata={}*, *new_epc_file=None*)

Adds a blocked well property from a numpy array to an existing resqml dataset.

Parameters

- **epc_file** (*string*) – file name to load model resqml model from (and rewrite to if new_epc_file is None)
- **a** (*1D numpy array*) – the blocked well property array to be added to the model
- **property_kind** (*string*) – the resqml property kind

- **blocked_well_uuid** (*uuid object or string*) – the uuid of the blocked well to which the property relates
- **source_info** (*string*) – typically the name of a file from which the array has been read but can be any information regarding the source of the data
- **title** (*string*) – this will be used as the citation title when a part is generated for the array
- **discrete** (*boolean, default False*) – if True, the array should contain integer (or boolean) data; if False, float
- **uom** (*string, default None*) – the resqml units of measure for the data; not relevant to discrete data
- **time_index** (*integer, default None*) – if not None, the time index to be used when creating a part for the array
- **time_series_uuid** (*uuid object or string, default None*) – required if time_index is not None
- **string_lookup_uuid** (*uuid object or string, optional*) – required if the array is to be stored as a categorical property; set to None for non-categorical discrete data; only relevant if discrete is True
- **null_value** (*int, default None*) – if present, this is used in the metadata to indicate that this value is to be interpreted as a null value wherever it appears in the data (use for discrete data only)
- **indexable_element** (*string, default 'cells'*) – the indexable element in the supporting representation (the blocked well); valid values are 'cells', 'intervals' (which includes unblocked intervals), or 'nodes'
- **facet_type** (*string*) – resqml facet type, or None
- **facet** (*string*) – resqml facet, or None
- **realization** (*int*) – realization number, or None
- **local_property_kind_uuid** (*uuid.UUID or string*) – uuid of local property kind, or None
- **count_per_element** (*int, default 1*) – the number of values per indexable element; if greater than one then this must be the fastest cycling axis in the cached array, ie last index; if greater than 1 then a must be a 2D array
- **points** (*bool, default False*) – if True, this is a points property with an extra dimension of extent 3
- **extra_metadata** (*dict, optional*) – any items in this dictionary are added as extra metadata to the new property
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new property object; if present, a new epc file (& associated h5 file) is created to contain a copy of the blocked well (and dependencies) and the new property

Returns

uuid.UUID of newly created property object

7.3.4 resqpy.derived_model.add_one_grid_property_array

resqpy.derived_model.**add_one_grid_property_array**(*epc_file*, *a*, *property_kind*, *grid_uuid=None*, *source_info='imported'*, *title=None*, *discrete=False*, *uom=None*, *time_index=None*, *time_series_uuid=None*, *string_lookup_uuid=None*, *null_value=None*, *indexable_element='cells'*, *facet_type=None*, *facet=None*, *realization=None*, *local_property_kind_uuid=None*, *count_per_element=1*, *const_value=None*, *expand_const_arrays=False*, *points=False*, *extra_metadata={}*, *use_int32=True*, *new_epc_file=None*)

Adds a grid property from a numpy array to an existing resqml dataset.

Parameters

- **epc_file** (*string*) – file name to load resqml model from (and rewrite to if new_epc_file is None)
- **a** (*3D numpy array*) – the property array to be added to the model; for a constant array set this None and use the const_value argument, otherwise this array is required
- **property_kind** (*string*) – the resqml property kind
- **grid_uuid** (*uuid object or string, optional*) – the uuid of the grid to which the property relates; if None, the property is attached to the 'main' grid
- **source_info** (*string*) – typically the name of a file from which the array has been read but can be any information regarding the source of the data
- **title** (*string*) – this will be used as the citation title when a part is generated for the array; for simulation models it is desirable to use the simulation keyword when appropriate
- **discrete** (*boolean, default False*) – if True, the array should contain integer (or boolean) data; if False, float
- **uom** (*string, default None*) – the resqml units of measure for the data; not relevant to discrete data
- **time_index** (*integer, default None*) – if not None, the time index to be used when creating a part for the array
- **time_series_uuid** (*uuid object or string, default None*) – required if time_index is not None
- **string_lookup_uuid** (*uuid object or string, optional*) – required if the array is to be stored as a categorical property; set to None for non-categorical discrete data; only relevant if discrete is True
- **null_value** (*int, default None*) – if present, this is used in the metadata to indicate that this value is to be interpreted as a null value wherever it appears in the data (use for discrete data only)
- **indexable_element** (*string, default 'cells'*) – the indexable element in the supporting representation (the grid)
- **facet_type** (*string*) – resqml facet type, or None
- **facet** (*string*) – resqml facet, or None
- **realization** (*int*) – realization number, or None

- **local_property_kind_uuid** (*uuid.UUID or string*) – uuid of local property kind, or None
- **count_per_element** (*int, default 1*) – the number of values per indexable element; if greater than one then this must be the fastest cycling axis in the cached array, ie last index
- **const_value** (*float or int, optional*) – if present, a constant array is added 'filled' with this value, in which case argument a should be None
- **expand_const_arrays** (*bool, default False*) – if True and a const_value is provided, a fully expanded array is added to the model instead of a const array
- **points** (*bool, default False*) – if True, this is a points property with an extra dimension of extent 3
- **extra_metadata** (*dict, optional*) – any items in this dictionary are added as extra metadata to the new property
- **use_int32** (*bool, default True*) – if True, and the array a has int64 bit elements, they are written as 32 bit data to hdf5; if False, 64 bit data is written in that situation
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new property object; if present, a new epc file (& associated h5 file) is created to contain a copy of the grid and the new property

Returns

uuid.UUID of newly created property object

7.3.5 resqpy.derived_model.add_single_cell_grid

resqpy.derived_model.**add_single_cell_grid**(*points*, *new_grid_title=None*, *new_epc_file=None*, *xy_units='m'*, *z_units='m'*, *z_inc_down=True*)

Creates a model with a single cell IJK Grid, with a cuboid cell aligned with x,y,z axes, enclosing points.

7.3.6 resqpy.derived_model.add_wells_from_ascii_file

resqpy.derived_model.**add_wells_from_ascii_file**(*epc_file*, *crs_uuid*, *trajectory_file*, *comment_character='#'*, *space_separated_instead_of_csv=False*, *well_col='WELL'*, *md_col='MD'*, *x_col='X'*, *y_col='Y'*, *z_col='Z'*, *length_uom='m'*, *md_domain=None*, *drilled=False*, *z_inc_down=True*, *new_epc_file=None*)

Adds new md datum, trajectory, interpretation and feature objects for each well in a tabular ascii file..

Parameters

- **epc_file** (*string*) – file name to load model resqml model from (and rewrite to if new_epc_file is None)
- **crs_uuid** (*uuid.UUID*) – the unique identifier of the coordinate reference system applicable to the x,y,z data; if None, a default crs will be created, making use of the length_uom and z_inc_down arguments
- **trajectory_file** (*string*) – the path of the ascii file holding the well trajectory data to be loaded
- **comment_character** (*string, default '#'*) – character deemed to introduce a comment in the trajectory file

- **space_separated_instead_of_csv** (*boolean, default False*) – if True, the columns in the trajectory file are space separated; if False, comma separated
- **well_col** (*string, default 'WELL'*) – the heading for the column containing well names
- **md_col** (*string, default 'MD'*) – the heading for the column containing measured depths
- **x_col** (*string, default 'X'*) – the heading for the column containing X (usually easting) data
- **y_col** (*string, default 'Y'*) – the heading for the column containing Y (usually northing) data
- **z_col** (*string, default 'Z'*) – the heading for the column containing Z (depth or elevation) data
- **length_uom** (*string, default 'm'*) – the units of measure for the measured depths; should be 'm' or 'ft'
- **md_domain** (*string, optional*) – the source of the original deviation data; may be 'logger' or 'driller'
- **drilled** (*boolean, default False*) – True should be used for wells that have been drilled; False otherwise (planned, proposed, or a location being studied)
- **z_inc_down** (*boolean, default True*) – indicates whether z values increase with depth; only used in the creation of a default coordinate reference system; ignored if crs_uuid is not None
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new property object; if present, a new epc file (& associated h5 file) is created to contain a copy of the grid and the new property

Returns

int – the number of wells added

Notes

ascii file must be table with first line being column headers, with columns for WELL, MD, X, Y & Z; actual column names can be set with optional arguments; all the objects are added to the model, with array data being written to the hdf5 file for the trajectories; the md_domain and drilled values are stored in the RESQML metadata but are only for human information and do not generally affect computations

7.3.7 resqpy.derived_model.add_zone_by_layer_property

resqpy.derived_model.**add_zone_by_layer_property**(*epc_file*, *grid_uuid=None*, *zone_by_layer_vector=None*, *zone_by_cell_property_uuid=None*, *use_dominant_zone=False*, *use_local_property_kind=True*, *null_value=-1*, *title='ZONE'*, *realization=None*, *extra_metadata={}*)

Adds a discrete zone property (and local property kind) with indexable element of layers.

Parameters

- **epc_file** (*string*) – file name to load resqml model from and to update with the zonal property

- **grid_uuid** (*uuid.UUID or str, optional*) – required unless the model has only one grid, or one named ROOT
- **zone_by_layer_vector** (*nk integers, optional*) – either this or zone_by_cell_property_uuid must be given; a 1D numpy array, tuple or list of ints, being the zone number to which each layer belongs
- **zone_by_cell_property_uuid** (*uuid.UUID or str, optional*) – either this or zone_by_layer_vector must be given; the uuid of a discrete property with grid as supporting representation and cells as indexable elements, holidng the zone to which the cell belongs
- **use_dominant_zone** (*boolean, default False*) – if True and more than one zone is represented within the cells of a layer, then the whole layer is assigned to the zone with the biggest count of cells in the layer; if False, an exception is raised if more than one zone is represented by the cells of a layer; ignored if zone_by_cell_property_uuid is None
- **use_local_property_kind** (*boolean, default True*) – if True, the new zone by layer property is given a local property kind titled 'zone'; if False, the property kind will be set to 'discrete'
- **null_value** (*int, default -1*) – the value to use if a layer does not belong to any zone (rarely used)
- **title** (*str, default 'ZONE'*) – the citation title of the new zone by layer property
- **realization** (*int, optional*) – if present the new zone by layer property is marked as belonging to this realization
- **extra_metadata** (*dict, optional*) – any items in this dictionary are added as extra metadata to the new property

Returns

numpy vector of zone numbers (by layer), uuid of newly created property

7.3.8 resqpy.derived_model.coarsened_grid

resqpy.derived_model.**coarsened_grid**(*epc_file*, *source_grid*, *fine_coarse*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *set_parent_window=None*, *infill_missing_geometry=True*, *new_grid_title=None*, *new_epc_file=None*)

Generates a coarsened version of an unsplit source grid, optionally inheriting properties.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **fine_coarse** (*resqpy.olio.fine_coarse.FineCoarse object*) – the mapping between cells in the fine (source) and coarse (output) grids
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid, with values upscaled or sampled
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False

- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **set_parent_window** (*boolean or str, optional*) – if True or 'parent', the coarsened grid has its parent window attribute set; if False, the parent window is not set; if None, the default will be True if new_epc_file is None or False otherwise; if 'grandparent' then an intervening parent window with no refinement or coarsening will be skipped and its box used in the parent window for the new grid, relating directly to the original grid
- **infill_missing_geometry** (*boolean, default True*) – if True, an attempt is made to generate grid geometry in the source grid wherever it is undefined; if False, any undefined geometry will result in an assertion failure
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the refined grid (& crs)

Returns

new grid object being the coarsened grid; the epc and hdf5 files are written to as an intentional side effect

---

Note: this function coarsens an entire grid; to coarsen a local area of a grid, first use the extract_box function and then use this function on the extracted grid; in such a case, using a value of 'grandparent' for the set_parent_window argument will relate the coarsened grid back to the original

---

7.3.9 resqpy.derived_model.copy_grid

resqpy.derived_model.**copy_grid**(*source_grid*, *target_model=None*, *copy_crs=True*)

Creates a copy of the IJK grid object in the target model (usually prior to modifying points in situ).

Note: this function is not usually called directly by application code; it does not write to the hdf5 file nor create xml for the copied grid; the copy will be a resqpy Grid even if the source grid is a RegularGrid

---

7.3.10 resqpy.derived_model.drape_to_surface

resqpy.derived_model.**drape_to_surface**(*epc_file*, *source_grid=None*, *surface=None*, *scaling_factor=None*, *ref_k0=0*, *ref_k_faces='top'*, *quad_triangles=True*, *border=None*, *store_displacement=False*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *new_grid_title=None*, *new_epc_file=None*)

Return a new grid with geometry draped to a surface.

Extend a resqml model with a new grid where the reference layer boundary of the source grid has been re-draped to a surface.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file

- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **surface** (*surface.Surface object, optional*) – the surface to drape the grid to; if None, a surface is generated from the reference layer boundary (which can then be scaled with the scaling_factor)
- **scaling_factor** (*float, optional*) – if not None, prior to draping, the surface is stretched vertically by this factor, away from a horizontal plane located at the surface's shallowest depth
- **ref_k0** (*integer, default 0*) – the reference layer (zero based) to drape to the surface
- **ref_k_faces** (*string, default 'top'*) – 'top' or 'base' identifying which bounding interface to use as the reference
- **quad_triangles** (*boolean, default True*) – if True and surface is None, each cell face in the reference boundary layer is represented by 4 triangles (with a common vertex at the face centre) in the generated surface; if False, only 2 trianges are used for each cell face (which gives a non-unique solution)
- **cell_range** (*integer, default 0*) – the number of cells away from faults which will have depths adjusted to spatially smooth the effect of the throw scaling (ie. reduce sudden changes in gradient due to the scaling)
- **offset_decay** (*float, default 0.5*) – the factor to reduce depth shifts by with each cell step away from faults (used in conjunction with cell_range)
- **store_displacement** (*boolean, default False*) – if True, 3 grid property parts are created, one each for x, y, & z displacement of cells' centres brought about by the local depth shift
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the draped grid (& crs)

Returns

new grid (grid.Grid object), with geometry draped to surface

Notes

at least one of a surface or a scaling factor must be given; if no surface is given, one is created from the fault-healed grid points for the reference layer interface; if a scaling factor other than 1.0 is given, the surface is flexed vertically, relative to its shallowest point; layer thicknesses measured along pillars are maintained; cell volumes may change; the coordinate reference systems for the surface and the grid are assumed to be the same; this function currently uses an exhaustive, computationally and memory intensive algorithm; setting quad_triangles argument to False should give a factor of 2 speed up and reduction in memory requirement; the epc file and associated hdf5 file are appended to (extended) with the new grid, as a side effect of this function

7.3.11 resqpy.derived_model.extract_box

resqpy.derived_model.**extract_box**(*epc_file=None*, *source_grid=None*, *box=None*, *box_inactive=None*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *set_parent_window=None*, *new_grid_title=None*, *new_epc_file=None*)

Extends an existing model with a new grid extracted as a logical IJK box from the source grid.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **box** (*numpy int array of shape (2, 3)*) – the minimum and maximum kji0 indices in the source grid (zero based) to include in the extracted grid; note that cells with index equal to maximum value are included (unlike with python ranges)
- **box_inactive** (*numpy bool array, optional*) – if present, shape must match box and values will be or'ed in with the inactive mask inherited from the source grid; if None, inactive mask will be as inherited from source grid
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid, with values taken from the specified box
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **set_parent_window** (*boolean, optional*) – if True, the extracted grid has its parent window attribute set; if False, the parent window is not set; if None, the default will be True if new_epc_file is None or False otherwise
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the extracted grid (& crs)

Returns

new grid object with extent as implied by the box argument

Note: the epc file and associated hdf5 file are appended to (extended) with the new grid, unless a new_epc_file is specified, in which case the grid and inherited properties are written there instead

7.3.12 resqpy.derived_model.extract_box_for_well

resqpy.derived_model.**extract_box_for_well**(*epc_file=None*, *source_grid=None*, *min_k0=None*, *max_k0=None*, *trajectory_epc=None*, *trajectory_uuid=None*, *blocked_well_uuid=None*, *column_ji0=None*, *column_xy=None*, *well_name=None*, *radius=None*, *outer_radius=None*, *active_cells_shape='tube'*, *quad_triangles=True*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *inherit_well=False*, *set_parent_window=None*, *new_grid_title=None*, *new_epc_file=None*)

Extends an existing model with a new grid extracted as an IJK box around a well trajectory in the source grid.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **min_k0** (*integers, optional*) – layer range to include; default is full vertical range of source grid
- **max_k0** (*integers, optional*) – layer range to include; default is full vertical range of source grid
- **trajectory_epc** (*string, optional*) – the source file for the trajectory or blocked well, if different to that for the source grid
- **trajectory_uuid** (*uuid.UUID*) – the uuid of the trajectory object for the well, if working from a trajectory
- **blocked_well_uuid** (*uuid.UUID*) – the uuid of the blocked well object, an alternative to working from a trajectory; must include blocking against source_grid
- **column_ji0** (*integer pair, optional*) – an alternative to providing a trajectory: the column indices of a 'vertical' well
- **column_xy** (*float pair, optional*) – an alternative to column_ji0: the x, y location used to determine the column
- **well_name** (*string, optional*) – name to use for column well, ignored if trajectory_uuid is not None
- **radius** (*float, optional*) – the radius around the wellbore to include in the box; units are those of grid xy values; radial distances are applied horizontally regardless of well inclination; if not present, only cells penetrated by the trajectory are included
- **outer_radius** (*float, optional*) – an outer radius around the wellbore, beyond which an inactive cell mask for the source_grid will be set to True (inactive); units are those of grid xy values
- **active_cells_shape** (*string, default 'tube'*) – the logical shape of cells marked as active in the extracted box; 'tube' results in an active shape with circular cross section in IJ

planes, that follows the trajectory; 'prism' activates all cells in IJ columns where any cell is within the tube; 'box' leaves the entire IJK cuboid active

- **quad_triangles** (*boolean, default True*) – if True, cell K faces are treated as 4 triangles (with a common face centre point) when computing the intersection of the trajectory with layer interfaces (horizons); if False, the K faces are treated as 2 triangles
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid, with values taken from the extracted box
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **inherit_well** (*boolean, default False*) – if True, the new model will have a copy of the well trajectory, its crs (if different from that of the grid), and any related wellbore interpretation and feature
- **set_parent_window** (*boolean, optional*) – if True, the extracted grid has its parent window attribute set; if False, the parent window is not set; if None, the default will be True if new_epc_file is None or False otherwise
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the extracted grid (& crs)

Returns

(grid, box) where – grid is the new Grid object with extent as determined by source grid geometry, trajectory and radius arguments; and box is a numpy int array of shape (2, 3) with first axis covering min, max and second axis covering k,j,i; the box array holds the minimum and maximum indices (zero based) in the source grid that have been included in the extraction (nb. maximum indices are included, unlike the usual python protocol)

Notes

this function is designed to work fully for vertical and deviated wells; for horizontal wells use blocked well mode; the extracted box includes all layers between the specified min and max horizons, even if the trajectory terminates above the deeper horizon or does not intersect horizon(s) for other reasons; when specifying a column well by providing x,y the IJ column with the centre of the topmost k face closest to the given point is selected; if an outer_radius is given, a boolean property will be created for the source grid with values set True where the centres of the cells are beyond this distance from the well, measured horizontally; if outer_radius and new_epc_file are both given, the source grid will be copied to the new epc

7.3.13 resqpy.derived_model.fault_throw_scaling

resqpy.derived_model.**fault_throw_scaling**(*epc_file*, *source_grid=None*, *scaling_factor=None*, *connection_set=None*, *scaling_dict=None*, *ref_k0=0*, *ref_k_faces='top'*, *cell_range=0*, *store_displacement=False*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *inherit_gcs=True*, *new_grid_title=None*, *new_epc_file=None*)

Extends epc with a new grid with fault throws multiplied by scaling factors.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **scaling_factor** (*float, optional*) – if present, the default scaling factor to apply to split pillars which do not appear in any of the faults in the scaling dictionary; if None, such pillars are left unchanged
- **connection_set** (*fault.GridConnectionSet object*) – the connection set with associated fault feature list, used to identify which faces (and hence pillars) belong to which named fault
- **scaling_dict** (*dictionary mapping string to float*) – the scaling factor to apply to each named fault; any faults not included in the dictionary will be left unadjusted (unless a default scaling factor is given as scaling_factor arg)
- **ref_k0** (*integer, default 0*) – the reference layer (zero based) to use when determining the pre-existing throws
- **ref_k_faces** (*string, default 'top'*) – 'top' or 'base' identifying which bounding interface to use as the reference
- **cell_range** (*integer, default 0*) – the number of cells away from faults which will have depths adjusted to spatially smooth the effect of the throw scaling (ie. reduce sudden changes in gradient due to the scaling)
- **store_displacement** (*boolean, default False*) – if True, 3 grid property parts are created, one each for x, y, & z displacement of cells' centres brought about by the fault throw scaling
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **inherit_gcs** (*boolean, default True*) – if True, any grid connection set objects related to the source grid will be inherited by the modified grid
- **new_grid_title** (*string*) – used as the citation title text for the new grid object

- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the derived grid (& crs)

Returns

new grid (grid.Grid object), with fault throws scaled according to values in the scaling dictionary

Notes

grid points are moved along pillar lines; stretch is towards or away from mid-point of throw; same shift is applied to all layers along pillar; pillar lines assumed to be straight; if a large fault is represented by a series of parallel minor faults 'stepping' down, each minor fault will have the scaling factor applied independently, leading to some unrealistic results

7.3.14 resqpy.derived_model.gather_ensemble

resqpy.derived_model.**gather_ensemble**(*case_epc_list*, *new_epc_file*, *consolidate=True*, *shared_grids=True*, *shared_time_series=True*, *create_epc_lookup=True*)

Creates a composite resqml dataset by merging all parts from all models in list, assigning realization numbers.

Parameters

- **case_epc_list** (*list of strings*) – paths of individual realization epc files
- **new_epc_file** (*string*) – path of new composite epc to be created (with paired hdf5 file)
- **consolidate** (*boolean, default True*) – if True, simple parts are tested for equivalence and where similar enough a single shared object is established in the composite dataset
- **shared_grids** (*boolean, default True*) – if True and consolidate is True, then grids are also consolidated with equivalence based on extent of grids (and citation titles if grid extents within the first case are not distinct); ignored if consolidate is False
- **shared_time_series** (*boolean, default False*) – if True and consolidate is True, then time series are consolidated with equivalence based on title, without checking that timestamp lists are the same
- **create_epc_lookup** (*boolean, default True*) – if True, a StringLookupTable is created to map from realization number to case epc path

Notes

property objects will have an integer realization number assigned, which matches the corresponding index into the case_epc_list; if consolidating with shared grids, then only properties will be gathered from realisations after the first and an exception will be raised if the grids are not matched between realisations

7.3.15 resqpy.derived_model.global_fault_throw_scaling

resqpy.derived_model.**global_fault_throw_scaling**(*epc_file*, *source_grid=None*, *scaling_factor=None*, *ref_k0=0*, *ref_k_faces='top'*, *cell_range=0*, *store_displacement=False*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *inherit_gcs=True*, *new_grid_title=None*, *new_epc_file=None*)

Rewrites epc with a new grid with all the fault throws multiplied by the same scaling factor.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **scaling_factor** (*float*) – the scaling factor to apply to the throw across all split pillars
- **ref_k0** (*integer, default 0*) – the reference layer (zero based) to use when determining the pre-existing throws
- **ref_k_faces** (*string, default 'top'*) – 'top' or 'base' identifying which bounding interface to use as the reference
- **cell_range** (*integer, default 0*) – the number of cells away from faults which will have depths adjusted to spatially smooth the effect of the throw scaling (ie. reduce sudden changes in gradient due to the scaling)
- **store_displacement** (*boolean, default False*) – if True, 3 grid property parts are created, one each for x, y, & z displacement of cells' centres brought about by the fault throw scaling
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **inherit_gcs** (*boolean, default True*) – if True, any grid connection set objects related to the source grid will be inherited by the modified grid
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the derived grid (& crs)

Returns

new grid (grid.Grid object), with all fault throws scaled by the scaling factor

Notes

a scaling factor of 1 implies no change; calls fault_throw_scaling(), see also documentation for that function

7.3.16 resqpy.derived_model.interpolated_grid

resqpy.derived_model.**interpolated_grid**(*epc_file*, *grid_a*, *grid_b*, *a_to_b_0_to_1=0.5*, *split_tolerance=0.01*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *new_grid_title=None*, *new_epc_file=None*)

Extends an existing model with a new grid geometry linearly interpolated between the two source_grids.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to
- **grid_a** (*grid.Grid objects*) – a pair of RESQML grid objects representing the end cases, between which the new grid will be interpolated
- **grid_b** (*grid.Grid objects*) – a pair of RESQML grid objects representing the end cases, between which the new grid will be interpolated
- **a_to_b_0_to_1** (*float, default 0.5*) – the interpolation factor in the range zero to one; a value of 0.0 will yield a copy of grid a, a value of 1.0 will yield a copy of grid b, intermediate values will yield a grid with all points interpolated
- **split_tolerance** (*float, default 0.01*) – maximum offset of corner points for shared point to be generated; units are same as those in grid crs; only relevant if working from corner points, ignored otherwise
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with grid_a
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the interpolated grid (& crs)

Returns

new grid object (grid.Grid) with geometry interpolated between grid a and grid b

Notes

the hdf5 file used by the grid_a model is appended to, so it is recommended that the grid_a model's epc is specified as the first argument (unless a new epc file is required, sharing the hdf5 file)

7.3.17 resqpy.derived_model.local_depth_adjustment

resqpy.derived_model.**local_depth_adjustment**(*epc_file*, *source_grid*, *centre_x*, *centre_y*, *radius*, *centre_shift*, *use_local_coords*, *decay_shape='quadratic'*, *ref_k0=0*, *store_displacement=False*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *new_grid_title=None*, *new_epc_file=None*)

Applies a local depth adjustment to the grid, adding as a new grid part in the model.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – a multi-layer RESQML grid object; if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **centre_x** (*floats*) – the centre of the depth adjustment, corresponding to the location of maximum change in depth; crs is implicitly that of the grid but see also use_local_coords argument
- **centre_y** (*floats*) – the centre of the depth adjustment, corresponding to the location of maximum change in depth; crs is implicitly that of the grid but see also use_local_coords argument
- **radius** (*float*) – the radius of adjustment of depths; units are implicitly xy (projected) units of grid crs
- **centre_shift** (*float*) – the maximum vertical depth adjustment; units are implicily z (vertical) units of grid crs; use positive value to increase depth, negative to make shallower
- **use_local_coords** (*boolean*) – if True, centre_x & centre_y are taken to be in the local coordinates of the grid's crs; otherwise the global coordinates
- **decay_shape** (*string*) – 'linear' yields a cone shaped change in depth values; 'quadratic' (the default) yields a bell shaped change
- **ref_k0** (*integer, default 0*) – the layer in the grid to use as reference for determining the distance of a pillar from the centre of the depth adjustment; the corners of the top face of the reference layer are used
- **store_displacement** (*boolean, default False*) – if True, 3 grid property parts are created, one each for x, y, & z displacement of cells' centres brought about by the local depth shift
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False

- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the adjusted grid (& crs)

Returns

new grid object which is a copy of the source grid with the local depth adjustment applied

7.3.18 resqpy.derived_model.refined_grid

resqpy.derived_model.**refined_grid**(*epc_file*, *source_grid*, *fine_coarse*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *source_grid_uuid=None*, *set_parent_window=None*, *infill_missing_geometry=True*, *new_grid_title=None*, *new_epc_file=None*)

Generates a refined version of the source grid, optionally inheriting properties.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid unless source_grid_uuid is specified to identify the grid
- **fine_coarse** (*resqpy.olio.fine_coarse.FineCoarse object*) – the mapping between cells in the fine (output) and coarse (source) grids
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid, with values resampled in the simplest way onto the finer grid
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **source_grid_uuid** (*uuid.UUID, optional*) – the uuid of the source grid – an alternative to the source_grid argument as a way of identifying the grid
- **set_parent_window** (*boolean or str, optional*) – if True or 'parent', the refined grid has its parent window attribute set; if False, the parent window is not set; if None, the default will be True if new_epc_file is None or False otherwise; if 'grandparent' then an intervening parent window with no refinement or coarsening will be skipped and its box used in the parent window for the new grid, relating directly to the original grid
- **infill_missing_geometry** (*boolean, default True*) – if True, an attempt is made to generate grid geometry in the source grid wherever it is undefined; if False, any undefined geometry will result in an assertion failure

- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the refined grid (& crs)

Returns

new grid object being the refined grid; the epc and hdf5 files are written to as an intentional side effect

Notes

this function refines an entire grid; to refine a local area of a grid, first use the extract_box function and then use this function on the extracted grid; in such a case, using a value of 'grandparent' for the set_parent_window argument will relate the refined grid back to the original; if geometry infilling takes place, cached geometry and mask arrays within the source grid object will be modified as a side-effect of the function (but not written to hdf5 or changed in xml)

7.3.19 resqpy.derived_model.single_layer_grid

resqpy.derived_model.**single_layer_grid**(*epc_file*, *source_grid=None*, *k0_min=None*, *k0_max=None*, *inactive_laissez_faire=True*, *new_grid_title=None*, *new_epc_file=None*)

Extends an existing model with a new version of the source grid converted to a single, thick, layer.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – a multi-layer RESQML grid object; if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **k0_min** (*int, optional*) – the minimum layer number in the source grid (zero based) to include in the single layer version; default is zero (ie. top layer in source grid)
- **k0_max** (*int, optional*) – the maximum layer number in the source grid (zero based) to include in the single layer version; default is nk - 1 (ie. bottom layer in source grid)
- **inactive_laissez_faire** (*boolean, optional*) – if True, a cell in the single layer grid will be set active if any of the corresponding cells in the source grid are active; otherwise all corresponding cells in the source grid must be active for the single layer cell to be active; default is True
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the single layer grid (& crs)

Returns

new grid object (grid.Grid) with a single layer representation of the source grid

7.3.20 resqpy.derived_model.tilted_grid

resqpy.derived_model.**tilted_grid**(*epc_file*, *source_grid=None*, *pivot_xyz=None*, *azimuth=None*, *dip=None*, *store_displacement=False*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *new_grid_title=None*, *new_epc_file=None*)

Extends epc file with a new grid which is a version of the source grid tilted.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **pivot_xyz** (*triple float*) – a point in 3D space on the pivot axis, which is horizontal and orthogonal to azimuth
- **azimuth** – the direction of tilt (orthogonal to tilt axis), as a compass bearing in degrees
- **dip** – the angle to tilt the grid by, in degrees; a positive value tilts points in direction azimuth downwards (needs checking!)
- **store_displacement** (*boolean, default False*) – if True, 3 grid property parts are created, one each for x, y, & z displacement of cells' centres brought about by the tilting
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the tilted grid (& crs)

Returns

a new grid (grid.Grid object) which is a copy of the source grid tilted in 3D space

7.3.21 resqpy.derived_model.unsplit_grid

resqpy.derived_model.**unsplit_grid**(*epc_file*, *source_grid=None*, *inherit_properties=False*, *inherit_realization=None*, *inherit_all_realizations=False*, *new_grid_title=None*, *new_epc_file=None*)

Extends epc file with a new grid which is a version of the source grid with all faults healed.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file

- **source_grid** (*grid.Grid object, optional*) – if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **inherit_properties** (*boolean, default False*) – if True, the new grid will have a copy of any properties associated with the source grid
- **inherit_realization** (*int, optional*) – realization number for which properties will be inherited; ignored if inherit_properties is False
- **inherit_all_realizations** (*boolean, default False*) – if True (and inherit_realization is None), properties for all realizations will be inherited; if False, only properties with a realization of None are inherited; ignored if inherit_properties is False or inherit_realization is not None
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the unsplit grid (& crs)

Returns

a new grid (grid.Grid object) which is an unfaulted copy of the source grid

Notes

the faults are healed by shifting the thrown sides up and down to the midpoint, only along the line of the fault; to smooth the adjustments away from the line of the fault, use the global_fault_throw_scaling() function first

7.3.22 resqpy.derived_model.zonal_grid

resqpy.derived_model.**zonal_grid**(*epc_file*, *source_grid=None*, *zone_title=None*, *zone_uuid=None*, *zone_layer_range_list=None*, *k0_min=None*, *k0_max=None*, *use_dominant_zone=False*, *inactive_laissez_faire=True*, *new_grid_title=None*, *new_epc_file=None*)

Extends an existing model with a new version of the source grid converted to a single, thick, layer per zone.

Parameters

- **epc_file** (*string*) – file name to rewrite the model's xml to; if source grid is None, model is loaded from this file
- **source_grid** (*grid.Grid object, optional*) – a multi-layer RESQML grid object; if None, the epc_file is loaded and it should contain one ijk grid object (or one 'ROOT' grid) which is used as the source grid
- **zone_title** (*string*) – if not None, a discrete property with this as the citation title is used as the zone property
- **zone_uuid** (*string or uuid*) – if not None, a discrete property with this uuid is used as the zone property (see notes)
- **zone_layer_range_list** (*list of (int, int, int)*) – each entry being (min_k0, max_k0, zone_index); alternative to working from a zone array
- **k0_min** (*int, optional*) – the minimum layer number in the source grid (zero based) to include in the zonal version; default is zero (ie. top layer in source grid)
- **k0_max** (*int, optional*) – the maximum layer number in the source grid (zero based) to include in the zonal version; default is nk - 1 (ie. bottom layer in source grid)

- **use_dominant_zone** (*boolean, default False*) – if True, the most common zone value in each layer is used for the whole layer; if False, then variation of zone values in active cells in a layer will raise an assertion error
- **inactive_laissez_faire** (*boolean, optional*) – if True, a cell in the zonal grid will be set active if any of the corresponding cells in the source grid are active; otherwise all corresponding cells in the source grid must be active for the zonal cell to be active; default is True
- **new_grid_title** (*string*) – used as the citation title text for the new grid object
- **new_epc_file** (*string, optional*) – if None, the source epc_file is extended with the new grid object; if present, a new epc file (& associated h5 file) is created to contain the zonal grid (& crs)

Returns

new grid object (grid.Grid) with one layer per zone of the source grid

Notes

usually one of zone_title or zone_uuid or zone_layer_range_list should be passed, if none are passed then a single layer grid is generated; zone_layer_range_list will take precendence if present

7.3.23 resqpy.derived_model.zone_layer_ranges_from_array

resqpy.derived_model.**zone_layer_ranges_from_array**(*zone_array*, *min_k0=0*, *max_k0=None*, *use_dominant_zone=False*)

Returns a list of (zone_min_k0, zone_max_k0, zone_index) derived from zone_array.

Parameters

- **zone_array** (*3D numpy int array or masked array*) – array holding zone index value per cell
- **min_k0** (*int, default 0*) – the minimum layer number (0 based) to be included in the ranges
- **max_k0** (*int, default None*) – the maximum layer number (0 based) to be included in the ranges; note that this layer is included (unlike in python ranges); if None, the maximum layer number in zone_array is used
- **use_dominant_zone** (*boolean, default False*) – if True, the most common zone value in each layer is used for the whole layer; if False, then variation of zone values in active cells in a layer will raise an assertion error

Returns

a list of (int, int, int) being (zone_min_k0, zone_max_k0, zone_index) for each zone index value present

Notes

the function requires zone indices (for active cells, if zone_array is masked) within a layer to be consistent: an assertion error is raised otherwise; the returned list is sorted by layer ranges rather than zone index; if use_dominant_zone is True then a side effect of the function is to modify the values in zone_array to be consistent across each layer, effectively reassigning some cells to a different zone!

7.4 resqpy.fault

Grid Connection Set class and related functions.

Classes

GridConnectionSet	Class for obj_GridConnectionSetRepresentation holding pairs of connected faces, usually for faults.

Functions

add_connection_set_and_tmults	Add a grid connection set to a resqml model, based on a fault include file and a dictionary of fault:tmult pairs.
cell_set_skin_connection_set	Add a grid connection set containing external faces of selected set of cells.
combined_tr_mult_from_gcs_mults	Returns a triplet of transmissibility multiplier arrays over grid faces by combining those from gcs'es.
grid_columns_property_from_gcs_property	Derives a new grid columns property (map) from a single-grid gcs property using values for k faces.
k_gap_connection_set	Returns a new GridConnectionSet representing K face connections where a K gap is zero thickness.
pinchout_connection_set	Returns a new GridConnectionSet representing non-standard K face connections across pinchouts.
remove_external_faces_from_faces_df	Returns a subset of the rows of faces dataframe, excluding rows on external faces.
standardize_face_indicator_in_faces_df	Sets face indicators to uppercase I, J or K, always with + or - following direction, in situ.
zero_base_cell_indices_in_faces_df	Decrements all the cell indices in the fault face dataframe, in situ (or increments if reverse is True).

7.4.1 resqpy.fault.add_connection_set_and_tmults

resqpy.fault.**add_connection_set_and_tmults**(*model*, *fault_incl*, *tmult_dict=None*)

Add a grid connection set to a resqml model, based on a fault include file and a dictionary of fault:tmult pairs.

Grid connection set added to resqml model, with extra_metadata on the fault interpretation containing the MULTFL values

Parameters

- **model** – resqml model object

- **fault_incl** – fullpath to fault include file or list of fullpaths to fault include files
- **tmult_dict** – dictionary of fault name/transmissibility multiplier pairs (must align with faults in include file). Optional, if blank values in the fault.include file will be used instead

Returns
grid connection set uuid

7.4.2 resqpy.fault.cell_set_skin_connection_set

resqpy.fault.**cell_set_skin_connection_set**(*grid*, *cell_set_mask*, *feature_name*, *feature_type='geobody boundary'*, *title=None*, *create_organizing_objects_where_needed=True*)

Add a grid connection set containing external faces of selected set of cells.

Parameters

- **grid** (*Grid*) – the grid for which the connection set is required
- **cell_set_mask** (*numpy bool array of shape grid.extent_kji*) – True values identify cells included in the set
- **feature_name** (*str*) – the name of the skin feature
- **feature_type** (*str, default 'geobody boundary'*) – 'fault', 'horizon' or 'geobody boundary'
- **title** (*str, optional*) – the citation title to use for the gcs; defaults to the feature_name
- **create_organizing_objects_where_needed** (*bool, default True*) – if True, feature and interpretation objects will be created if they do not exist

Returns
the newly created grid connection set

Notes

this function does not take into consideration split pillars, it assumes cells are neighbouring based on the cell indices; faces on the outer skin of the grid are not included in the connection set; any cell face between a cell in the cell set and one not in it will be included in the connection set, therefore the set may contain internal skin faces as well as the outer skin

7.4.3 resqpy.fault.combined_tr_mult_from_gcs_mults

resqpy.fault.**combined_tr_mult_from_gcs_mults**(*model*, *gcs_tr_mult_uuid_list*, *merge_mode='minimum'*, *sided=None*, *fill_value=1.0*, *active_only=True*, *apply_baffles=False*)

Returns a triplet of transmissibility multiplier arrays over grid faces by combining those from gcs'es.

Parameters

- **model** (Model) – the model containing all the relevant objects
- **gcs_tr_mult_uuid_list** (*list of UUID*) – uuids of the individual grid connection set transmissibility multiplier properties to be combined
- **merge_mode** (*str, default 'minimum'*) – one of 'minimum', 'multiply', 'maximum', 'exception'; how to handle multiple values applicable to the same grid face

- **sided** (*bool, optional*) – whether to apply values on both sides of each gcs cell-face pair; if None, will default to False if merge mode is multiply, True otherwise
- **fill_value** (*float, optional*) – the value to use for grid faces not present in any of the gcs'es; if None, NaN will be used
- **active_only** (*bool, default True*) – if True and an active property exists for a grid connection set, then only active faces are used when combining to make the grid face arrays
- **apply_baffles** (*bool, default False*) – if True, where a baffle property exists for a grid connection set, a transmissibility multiplier of zero will be used for faces marked as True, overriding the multiplier property values at such faces

Returns

triple numpy float arrays being transmissibility multipliers for K, J, and I grid faces; arrays have
shapes (nk + 1, nj, ni), (nk, nj + 1, ni), and (nk, nj, ni + 1) respectively

Notes

each gcs, which is the supporting representation for each input tr mult property, must be for a single grid and that grid must be the same for all the gcs'es; the three arrays returned by this function can be flattened and concatenated to create a composite array compatible with RESQML documentation for grid properties with indexable elmement of 'faces'

7.4.4 resqpy.fault.grid_columns_property_from_gcs_property

resqpy.fault.**grid_columns_property_from_gcs_property**(*model*, *gcs_property_uuid*, *null_value=nan*, *title=None*, *multiple_handling='default'*)

Derives a new grid columns property (map) from a single-grid gcs property using values for k faces.

Parameters

- **model** (Model) – the model in which the existing objects are to be found and the new property added
- **gcs_property_uuid** (*UUID*) – the uuid of the existing grid connection set property
- **null_value** (*float or int, default NaN*) – the value to use in columns where no K faces are present in the gcs
- **title** (*str, optional*) – the title for the new grid property; defaults to that of the gcs property
- **multiple_handling** (*str, default 'mean'*) – one of 'default', 'mean', 'min', 'max', 'min_k', 'max_k', 'exception'; determines how a value is generated when more than one K face is present in the gcs for a column

Returns
uuid of the newly created Property (RESQML ContinuousProperty, DiscreteProperty, CategoricalProperty or PointsProperty)

Notes

the grid connection set which is the support for gcs_property must involve only one grid; the resulting columns grid property is of the same class as the original gcs property; the write_hdf() and create_xml() methods are called by this function, for the new property, which is added to the model; the default multiple handling mode is mean for continuous data, any for discrete (inc categorical); in the case of discrete (including categorical) data, a null_value of NaN will be changed to -1

7.4.5 resqpy.fault.k_gap_connection_set

resqpy.fault.**k_gap_connection_set**(*grid*, *skip_inactive=True*, *feature_name='k gap connection'*, *tolerance=0.001*)

Returns a new GridConnectionSet representing K face connections where a K gap is zero thickness.

Parameters

- **grid** (*grid.Grid*) – the grid for which a K gap connection set is required
- **skip_inactive** (*boolean, default True*) – if True, connections are not included where there is an inactive cell above or below the pinchout; if False, such connections are included
- **feature_name** (*string, default 'pinchout'*) – the name to use as citation title in the feature and interpretation
- **tolerance** (*float, default 0.001*) – the minimum vertical distance below which a K gap is deemed to be zero thickness; units are implicitly the z units of the coordinate reference system used by grid

Notes

this function does not write to hdf5, nor create xml for the new grid connection set; however, it does create one feature and a corresponding interpretation and creates xml for those; note that the entries in the connection set will be for logically K-neighbouring pairs of cells – such pairs are omitted from the standard transmissibilities due to the presence of the K gap layer

7.4.6 resqpy.fault.pinchout_connection_set

resqpy.fault.**pinchout_connection_set**(*grid*, *skip_inactive=True*, *feature_name='pinchout'*)

Returns a new GridConnectionSet representing non-standard K face connections across pinchouts.

Parameters

- **grid** (*grid.Grid*) – the grid for which a pinchout connection set is required
- **skip_inactive** (*boolean, default True*) – if True, connections are not included where there is an inactive cell above or below the pinchout; if False, such connections are included
- **feature_name** (*string, default 'pinchout'*) – the name to use as citation title in the feature and interpretation

Notes

this function does not write to hdf5, nor create xml for the new grid connection set; however, it does create one feature and a corresponding interpretation and creates xml for those

7.4.7 resqpy.fault.remove_external_faces_from_faces_df

resqpy.fault.**remove_external_faces_from_faces_df**(*faces*, *extent_kji*, *remove_all_k_faces=False*)

Returns a subset of the rows of faces dataframe, excluding rows on external faces.

7.4.8 resqpy.fault.standardize_face_indicator_in_faces_df

resqpy.fault.**standardize_face_indicator_in_faces_df**(*faces*)

Sets face indicators to uppercase I, J or K, always with + or - following direction, in situ.

7.4.9 resqpy.fault.zero_base_cell_indices_in_faces_df

resqpy.fault.**zero_base_cell_indices_in_faces_df**(*faces*, *reverse=False*)

Decrements all the cell indices in the fault face dataframe, in situ (or increments if reverse is True).

7.5 resqpy.grid

The Grid Module.

Classes

Grid	Class for RESQML Grid (extent and geometry) within RESQML model object.
RegularGrid	Class for completely regular block grids, usually aligned with xyz axes.

Functions

any_grid	Returns a Grid or RegularGrid or UnstructuredGrid object depending on the extra metadata in the xml.
establish_zone_property_kind	MOVED: Returns zone local property kind object, creating the xml and adding as part if not found in model.
extract_grid_parent	Returns the uuid of the parent grid for the supplied grid
find_cell_for_x_sect_xz	Returns the (k0, j0) or (k0, i0) indices of the cell containing point x,z in the cross section.
grid_flavour	Returns a string indicating type of grid geometry, currently 'IjkGrid' or 'IjkBlockGrid'.
is_regular_grid	Returns True if the xml root node is for a RegularGrid.

7.5.1 resqpy.grid.any_grid

resqpy.grid.**any_grid**(*parent_model*, *uuid=None*, *find_properties=True*)

Returns a Grid or RegularGrid or UnstructuredGrid object depending on the extra metadata in the xml.

Parameters

- **parent_model** (Model) – the model within which the grid exists
- **uuid** (*UUID*) – the uuid of the grid object to be instantiated
- **find_properties** (*bool, default True*) – passed onward to the instantiation method

Note: full list of resqpy grid class objects which could be returned: Grid, RegularGrid, UnstructuredGrid, TetraGrid, HexaGrid, PyramidGrid, PrismGrid

7.5.2 resqpy.grid.establish_zone_property_kind

resqpy.grid.**establish_zone_property_kind**(*model*)

MOVED: Returns zone local property kind object, creating the xml and adding as part if not found in model.

7.5.3 resqpy.grid.extract_grid_parent

resqpy.grid.**extract_grid_parent**(*grid*)

Returns the uuid of the parent grid for the supplied grid

7.5.4 resqpy.grid.find_cell_for_x_sect_xz

resqpy.grid.**find_cell_for_x_sect_xz**(*x_sect*, *x*, *z*)

Returns the (k0, j0) or (k0, i0) indices of the cell containing point x,z in the cross section.

Parameters

- **x_sect** (*numpy float array of shape (nk, nj or ni, 2, 2, 2 or 3)*) – the cross section x,z or x,y,z data
- **x** (*float*) –
- **z** (*float*) – y-coordinate of point of interest in the cross section space

Note: the x_sect data is in the form returned by x_section_corner_points() or split_gap_x_section_points(); the 2nd of the returned pair is either a J index or I index, whichever was not the axis specified when generating the x_sect data; returns (None, None) if point inclusion not detected; if xyz data is provided, the y values are ignored; note that the point of interest x,z coordinates are in the space of x_sect, so if rotation has occurred, the x value is no longer an easting and is typically picked off a cross section plot

7.5.5 resqpy.grid.grid_flavour

resqpy.grid.**grid_flavour**(*grid_root*)

Returns a string indicating type of grid geometry, currently 'IjkGrid' or 'IjkBlockGrid'.

7.5.6 resqpy.grid.is_regular_grid

resqpy.grid.**is_regular_grid**(*grid_root*)

Returns True if the xml root node is for a RegularGrid.

7.6 resqpy.grid_surface

Classes for RESQML objects related to surfaces.

Classes

GridSkin	Class of object consisting of outer skin of grid (not a RESQML class in its own right).

7.6.1 resqpy.grid_surface.GridSkin

class resqpy.grid_surface.**GridSkin**(*grid*, *quad_triangles=True*, *use_single_layer_tactics=True*, *is_regular=False*)

Bases: object

Class of object consisting of outer skin of grid (not a RESQML class in its own right).

Methods:

__init__(grid[, quad_triangles, ...])	Returns a composite surface object consisting of outer skin of grid.
find_first_intersection_of_trajectory(trajectory)	Returns the first intersection of the trajectory with the torn skin.

__init__(*grid*, *quad_triangles=True*, *use_single_layer_tactics=True*, *is_regular=False*)

Returns a composite surface object consisting of outer skin of grid.

find_first_intersection_of_trajectory(*trajectory*, *start=0*, *start_xyz=None*, *nudge=None*, *exclude_kji0=None*)

Returns the first intersection of the trajectory with the torn skin.

Returns the x,y,z and K,J,I and axis, polarity & segment.

Parameters

- **trajectory** (*well.Trajectory object*) – the trajectory to be intersected with the skin

- **start** (*int, default 0*) – the trajectory segment number to start the search from
- **start_xyz** (*triple float, optional*) – if present, this point should lie on the start segment and search continues from this point
- **nudge** (*float, optional*) – if present and positive, the start point is nudged forward by this distance (grid uom); if present and negative (more typical for skin entry search), the start point is nudged back a little
- **exclude_kji0** (*triple int, optional*) – if present, the indices of a cell to exclude as a possible result

Returns

5-tuple of –

- (triple float): xyz coordinates of the intersection point in the crs of the grid
- (triple int): kji0 of the cell that is intersected, which might be a pinched out or otherwise inactive cell
- (int): 0, 1 or 2 for K, J or I axis of cell face
- (int): 0 for -ve face, 1 for +ve face
- (int): trajectory knot prior to the intersection (also the segment number)

Note: if the GridSkin object has been initialised using single layer tactics, then the k0 value will be zero for any initial entry through a sidewall of the grid or through a fault face

---

Functions

bisector_from_faces	Creates a boolean array denoting the bisection of the grid by the face sets.
column_bisector_from_faces	Returns a numpy bool array denoting the bisection of the top layer of the grid by the curtain face sets.
create_column_face_mesh_and_surface	Creates a Mesh and corresponding Surface representing a column face.
find_faces_to_represent_surface	Returns a grid connection set containing those cell faces which are deemed to represent the surface.
find_faces_to_represent_surface_regular	Returns a grid connection set containing those cell faces which are deemed to represent the surface.
find_faces_to_represent_surface_regular_optimised	Returns a grid connection set containing those cell faces which are deemed to represent the surface.
find_faces_to_represent_surface_staffa	Returns a grid connection set containing those cell faces which are deemed to represent the surface.
find_first_intersection_of_trajectory_with_cell_surface	Return first intersection with cell's surface found along a trajectory.
find_first_intersection_of_trajectory_with_layer_interface	Returns info about the first intersection of well trajectory(s) with layer interface.
find_first_intersection_of_trajectory_with_surface	Returns xyz and other info of the first intersection of well trajectory with surface.
find_intersection_of_trajectory_interval_with_column_face	Searches for intersection of a single trajectory segment with an I or J column face.
find_intersections_of_trajectory_with_layer_interface	Returns an array of column indices and an array of xyz of intersections of well trajectory with layer interface.
find_intersections_of_trajectory_with_surface	Returns an array of triangle indices and an array of xyz of intersections of well trajectory with surface.
generate_surface_for_blocked_well_cells	Returns a surface or list of surfaces representing the faces of the cells visited by the well.
generate_torn_surface_for_layer_interface	Returns a Surface object generated from the grid layer interface points.
generate_torn_surface_for_x_section	Returns a Surface object generated from the grid cross section points.
generate_untorn_surface_for_layer_interface	Returns a Surface object generated from the grid layer interface points after any faults are 'healed'.
generate_untorn_surface_for_x_section	Returns a Surface object generated from the grid cross section points for an unfaulted grid.
get_boundary	Cretaes a dictionary of the indices that bound the surface (where the faces are True).
intersect_numba	Finds the faces that intersect the surface in 3D.
point_is_within_cell	Returns True if point xyz is within cell kji0, but not on its surface.
populate_blocked_well_from_trajectory	Populate an empty blocked well object based on the intersection of its trajectory with a grid.
shadow_from_faces	Returns a numpy int8 array indicating whether cells are above, below or between K faces.
trajectory_grid_overlap	Returns True if there is some overlap of the xyz boxes for the trajectory and grid, False otherwise.

7.6.2 resqpy.grid_surface.bisector_from_faces

resqpy.grid_surface.**bisector_from_faces**(*grid_extent_kji: Tuple[int, int, int]*, *k_faces: ndarray*, *j_faces: ndarray*, *i_faces: ndarray*, *raw_bisector: bool*) → Tuple[ndarray, bool]

Creates a boolean array denoting the bisection of the grid by the face sets.

Parameters

- **grid_extent_kji** (*Tuple[int, int, int]*) – the shape of the grid.
- **k_faces** (*np.ndarray*) – a boolean array of which faces represent the surface in the k dimension.
- **j_faces** (*np.ndarray*) – a boolean array of which faces represent the surface in the j dimension.
- **i_faces** (*np.ndarray*) – a boolean array of which faces represent the surface in the i dimension.

Returns

Tuple containing –

- **array (np.ndarray): boolean bisectors array where values are True for cells on the side** of the surface that has a lower mean k index on average and False for cells on the other side.
- is_curtain (bool): True if the surface is a curtain (vertical), otherwise False.

Notes

The face sets must form a single 'sealed' cut of the grid (eg. not waving in and out of the grid). Any 'boxed in' parts of the grid (completely enclosed by bisecting faces) will be consistently assigned to either the True or False part.

7.6.3 resqpy.grid_surface.column_bisector_from_faces

resqpy.grid_surface.**column_bisector_from_faces**(*grid_extent_ji: Tuple[int, int]*, *j_faces: ndarray*, *i_faces: ndarray*) → ndarray

Returns a numpy bool array denoting the bisection of the top layer of the grid by the curtain face sets.

Parameters

- **grid_extent_ji** (*pair of int*) – the shape of a layer of the grid
- **j_faces** (*numpy bool arrays*) – True where an internal grid face forms part of the bisecting surface, shaped for a single layer
- **i_faces** (*numpy bool arrays*) – True where an internal grid face forms part of the bisecting surface, shaped for a single layer

Returns

numpy bool array of shape grid_extent_ji, set True for cells on one side of the face sets; set False for cells on othe side

Notes

the face sets must form a single 'sealed' cut of the grid (eg. not waving in and out of the grid); any 'boxed in' parts of the grid (completely enclosed by bisecting faces) will be consistently assigned to the False part; the resulting array is suitable for use as a grid property with indexable element of columns; the array is set True for the side of the curtain that contains cell [0, 0]

7.6.4 resqpy.grid_surface.create_column_face_mesh_and_surface

resqpy.grid_surface.**create_column_face_mesh_and_surface**(*grid*, *col_ji0*, *axis*, *polarity*, *quad_triangles=True*, *as_single_layer=False*)

Creates a Mesh and corresponding Surface representing a column face.

Parameters

- **grid** (*grid.Grid object*) –
- **col_ji0** (*int pair*) – the column indices, zero based
- **axis** (*int*) – 1 for J face, 2 fo I face
- **polarity** (*int*) – 0 for negative face, 1 for positive
- **quad_triangles** (*boolean, default True*) – if True, 4 triangles are used per cell face; if False, 2 triangles
- **as_single_layer** (*boolean, default False*) – if True, only the top and basal points are used, with the results being equivalent to the grid being treated as a single layer

Returns

surface.Mesh, surface.Surface (or None, surface.Surface if grid has k gaps)

7.6.5 resqpy.grid_surface.find_faces_to_represent_surface

resqpy.grid_surface.**find_faces_to_represent_surface**(*grid*, *surface*, *name*, *mode='auto'*, *feature_type='fault'*, *progress_fn=None*)

Returns a grid connection set containing those cell faces which are deemed to represent the surface.

Parameters

- **grid** (*RegularGrid*) – the grid for which to create a grid connection set representation of the surface
- **surface** (*Surface*) – the triangulated surface for which grid cell faces are required
- **name** (*str*) – the feature name to use in the grid connection set
- **mode** (*str, default 'auto'*) – one of 'auto', 'staffa', 'regular', 'regular_optimised', 'regular_cuda'; auto will translate to regular_optimised for regulat grids, and staffa for irregular grids; regular_cude required GPU hardware and the correct installation of numba.cuda and cupy
- **feature_type** (*str, default 'fault'*) – 'fault', 'horizon' or 'geobody boundary'
- **(f(x** (*progress_fn*) – float), optional): a callback function to be called at intervals by this function; the argument will progress from 0.0 to 1.0 in unspecified and uneven increments

Returns
a new GridConnectionSet with a single feature, not yet written to hdf5 nor xml created

---

Note: this is a wrapper function selecting between more specialised functions; use those directly for more options

---

7.6.6 resqpy.grid_surface.find_faces_to_represent_surface_regular

resqpy.grid_surface.**find_faces_to_represent_surface_regular**(*grid*, *surface*, *name*, *title=None*, *centres=None*, *agitate=False*, *random_agitation=False*, *feature_type='fault'*, *progress_fn=None*, *consistent_side=False*, *return_properties=None*)

Returns a grid connection set containing those cell faces which are deemed to represent the surface.

Parameters

- **grid** (*RegularGrid*) – the grid for which to create a grid connection set representation of the surface
- **surface** (*Surface*) – the surface to be intersected with the grid
- **name** (*str*) – the feature name to use in the grid connection set
- **title** (*str, optional*) – the citation title to use for the grid connection set; defaults to name
- **centres** (*numpy float array of shape (nk, nj, ni, 3), optional*) – precomputed cell centre points in local grid space, to avoid possible crs issues; required if grid's crs includes an origin (offset)?
- **agitate** (*bool, default False*) – if True, the points of the surface are perturbed by a small offset, which can help if the surface has been built from a regular mesh with a periodic resonance with the grid
- **random_agitation** (*bool, default False*) – if True, the agitation is by a small random distance; if False, a constant positive shift of 5.0e-6 is applied to x, y & z values; ignored if agitate is False
- **feature_type** (*str, default 'fault'*) – 'fault', 'horizon' or 'geobody boundary'
- **(f(x** (*progress_fn*) – float), optional): a callback function to be called at intervals by this function; the argument will progress from 0.0 to 1.0 in unspecified and uneven increments
- **consistent_side** (*bool, default False*) – if True, the cell pairs will be ordered so that all the first cells in each pair are on one side of the surface, and all the second cells on the other
- **return_properties** (*list of str, optional*) – if present, a list of property arrays to calculate and return as a dictionary; recognised values in the list are 'offset' and 'normal vector'; offset is a measure of the distance between the centre of the cell face and the intersection point of the inter-cell centre vector with a triangle in the surface; normal vector is a unit vector normal to the surface triangle; each array has an entry for each face in the gcs; the returned dictionary has the passed strings as keys and numpy arrays as values

Returns

gcs or (gcs, dict) where gcs is a new GridConnectionSet with a single feature, not yet written to hdf5 nor xml created; dict is a dictionary mapping from property name to numpy array; 'offset' will map to a numpy float array of shape (gcs.count,); 'normal vector' will map to a numpy float array of shape (gcs.count, 3) holding a unit vector normal to the surface for each of the faces in the gcs; the dict is only returned if a non-empty list has been passed as return_properties

Notes

use find_faces_to_represent_surface_regular_optimised() instead, for faster computation; this function can handle the surface and grid being in different coordinate reference systems, as long as the implicit parent crs is shared; no trimming of the surface is carried out here: for computational efficiency, it is recommended to trim first; organisational objects for the feature are created if needed; if grid has differing xy & z units, this is accounted for here when generating normal vectors, ie. true normal unit vectors are returned

7.6.7 resqpy.grid_surface.find_faces_to_represent_surface_regular_optimised

resqpy.grid_surface.**find_faces_to_represent_surface_regular_optimised**(*grid*, *surface*, *name*, *title=None*, *agitate=False*, *random_agitation=False*, *feature_type='fault'*, *is_curtain=False*, *progress_fn=None*, *return_properties=None*, *raw_bisector=False*)

Returns a grid connection set containing those cell faces which are deemed to represent the surface.

argumants:

grid (RegularGrid): the grid for which to create a grid connection set representation of the surface; must be aligned, ie. I with +x, J with +y, K with +z and local origin of (0.0, 0.0, 0.0)

surface (Surface): the surface to be intersected with the grid name (str): the feature name to use in the grid connection set title (str, optional): the citation title to use for the grid connection set; defaults to name agitate (bool, default False): if True, the points of the surface are perturbed by a small

offset, which can help if the surface has been built from a regular mesh with a periodic resonance with the grid

random_agitation (bool, default False): if True, the agitation is by a small random distance; if False, a constant positive shift of 5.0e-6 is applied to x, y & z values; ignored if agitate is False

feature_type (str, default 'fault'): 'fault', 'horizon' or 'geobody boundary' is_curtain (bool, default False): if True, only the top layer of the grid is processed and the bisector

property, if requested, is generated with indexable element columns

progress_fn (f(x: float), optional): a callback function to be called at intervals by this function; the argument will progress from 0.0 to 1.0 in unspecified and uneven increments

return_properties (List[str]): if present, a list of property arrays to calculate and return as a dictionary; recognised values in the list are 'triangle', 'depth', 'offset', 'flange bool', 'grid bisector', or 'grid shadow'; triangle is an index into the surface triangles of the triangle detected for the gcs face; depth is the z value of the intersection point of the inter-cell centre vector with a triangle in

the surface; offset is a measure of the distance between the centre of the cell face and the intersection point; grid bisector is a grid cell boolean property holding True for the set of cells on one side of the surface, deemed to be shallower; grid shadow is a grid cell int8 property holding 0: cell neither above nor below a K face of the gridded surface, 1 cell is above K face(s), 2 cell is below K face(s), 3 cell is between K faces; the returned dictionary has the passed strings as keys and numpy arrays as values

raw_bisector (bool, default False): if True and grid bisector is requested then it is left in a raw form without assessing which side is shallower (True values indicate same side as origin cell)

Returns

gcs or (gcs, gcs_props) where gcs is a new GridConnectionSet with a single feature, not yet written to hdf5 nor xml created; gcs_props is a dictionary mapping from requested return_properties string to numpy array

Notes

this function is designed for aligned regular grids only; this function can handle the surface and grid being in different coordinate reference systems, as long as the implicit parent crs is shared; no trimming of the surface is carried out here: for computational efficiency, it is recommended to trim first; organisational objects for the feature are created if needed; if the offset return property is requested, the implicit units will be the z units of the grid's crs

7.6.8 resqpy.grid_surface.find_faces_to_represent_surface_staffa

resqpy.grid_surface.**find_faces_to_represent_surface_staffa**(*grid*, *surface*, *name*, *feature_type='fault'*, *progress_fn=None*)

Returns a grid connection set containing those cell faces which are deemed to represent the surface.

Note: this version of the find faces function is designed for irregular grids

7.6.9 resqpy.grid_surface.find_first_intersection_of_trajectory_with_cell_surface

resqpy.grid_surface.**find_first_intersection_of_trajectory_with_cell_surface**(*trajectory*, *grid*, *kji0*, *start_knot*, *start_xyz=None*, *nudge=0.001*, *quad_triangles=True*)

Return first intersection with cell's surface found along a trajectory.

7.6.10 resqpy.grid_surface.find_first_intersection_of_trajectory_with_layer_interface

resqpy.grid_surface.**find_first_intersection_of_trajectory_with_layer_interface**(*trajectory*, *grid*, *k0=0*, *ref_k_faces='top'*, *start=0*, *heal_faults=False*, *quad_triangles=True*, *is_regular=False*)

Returns info about the first intersection of well trajectory(s) with layer interface.

Parameters

- **trajectory** (*well.Trajectory object; or list thereof*) – the wellbore trajectory object(s) to find the intersection for; if a list of trajectories is provided then the return value is a corresponding list
- **grid** (*grid.Grid object*) – the grid object from which a layer interface is to be converted to a surface
- **k0** (*int*) – the layer number (zero based) to be used
- **ref_k_faces** (*string*) – either 'top' (the default) or 'base', indicating whether the top or the base interface of the layer is to be used
- **start** (*int, default 0*) – an index into the trajectory knots to start the search from; is applied naively to all trajectories when a trajectory list is passed
- **heal_faults** (*boolean, default False*) – if True, faults will be 'healed' to give an untorn surface before looking for intersections; if False and the trajectory passes through a fault plane without intersecting the layer interface then no intersection will be identified; makes no difference if the grid is unfaulted
- **quad_triangles** (*boolean, optional, default True*) – if True, 4 triangles are used to represent each cell k face, which gives a unique solution with a shared node of the 4 triangles at the mean point of the 4 corners of the face; if False, only 2 triangles are used, which gives a non-unique solution
- **is_regular** (*boolean, default False*) – set True if grid is a RegularGrid with IJK axes aligned with xyz axes

Returns

(numpy float array of shape (3,), int, (int, int)) – being the (x, y, z) intersection point, and the trajectory segment number, and the (j0, i0) column number of the first intersection point; or None, None, (None, None) if no intersection found; if the trajectory argument is a list of trajectories, then correponding list of numpy array, list of int, list of int pair are returned

Notes

interseections are found based on straight line segments between the trajectory control points, this will result in errors where there is significant curvature between neighbouring control points

7.6.11 resqpy.grid_surface.find_first_intersection_of_trajectory_with_surface

resqpy.grid_surface.**find_first_intersection_of_trajectory_with_surface**(*trajectory*, *surface*, *start=0*, *start_xyz=None*, *nudge=None*, *return_second=False*)

Returns xyz and other info of the first intersection of well trajectory with surface.

Parameters

- **trajectory** (*well.Trajectory object*) – the wellbore trajectory object(s) to find the intersection for
- **surface** (*surface.Surface object*) – the triangulated surface with which to search for intersections
- **start** (*int, default 0*) – an index into the trajectory knots to start the search from
- **start_xyz** (*triple float, optional*) – if present, should lie on start segment and search starts from this point
- **nudge** (*float, optional*) – if present and positive, starting xyz is nudged forward this distance along segment; if present and negative, starting xyz is nudged backward along segment
- **return_second** (*boolean, default False*) – if True, a sextuplet is returned with the last 3 elements identifying the 'runner up' intersection in the same trajectory segment, or None, None, None if only one intersection found

Returns

a triplet if return_second is False; a sextuplet if return_second is True; the first triplet is – (numpy float array of shape (3,), int, int): being the (x, y, z) intersection point, and the trajectory segment number, and the triangle index of the first intersection point; or None, None, None if no intersection found; if return_second is True, the 4th, 5th & 6th return values are similar to the first three, conveying information about the second intersection of the same trajectory segment with the surface, or None, None, None if a no second intersection was found

Notes

interseections are found based on straight line segments between the trajectory control points, this will result in errors where there is significant curvature between neighbouring control points

7.6.12 resqpy.grid_surface.find_intersection_of_trajectory_interval_with_column_face

resqpy.grid_surface.**find_intersection_of_trajectory_interval_with_column_face**(*trajectory*, *grid*, *start_knot*, *col_ji0*, *axis*, *polarity*, *start_xyz=None*, *nudge=None*, *quad_triangles=True*)

Searches for intersection of a single trajectory segment with an I or J column face.

Returns
xyz, k0

Note: does not support k gaps

7.6.13 resqpy.grid_surface.find_intersections_of_trajectory_with_layer_interface

resqpy.grid_surface.**find_intersections_of_trajectory_with_layer_interface**(*trajectory*, *grid*, *k0=0*, *ref_k_faces='top'*, *heal_faults=True*, *quad_triangles=True*)

Returns an array of column indices and an array of xyz of intersections of well trajectory with layer interface.

Parameters

- **trajectory** (*well.Trajectory object; or list thereof*) – the wellbore trajectory object(s) to find the intersections for; if a list of trajectories is provided then the return value is a corresponding list
- **grid** (*grid.Grid object*) – the grid object from which a layer interface is to be converted to a surface
- **k0** (*int*) – the layer number (zero based) to be used
- **ref_k_faces** (*string*) – either 'top' (the default) or 'base', indicating whether the top or the base interface of the layer is to be used
- **heal_faults** (*boolean, default True*) – if True, faults will be 'healed' to give an untorn surface before looking for intersections; if False and the trajectory passes through a fault plane without intersecting the layer interface then no intersection will be identified; makes no difference if the grid is unfaulted
- **quad_triangles** (*boolean, optional, default True*) – if True, 4 triangles are used to represent each cell k face, which gives a unique solution with a shared node of the 4 triangles at the mean point of the 4 corners of the face; if False, only 2 triangles are used, which gives a non-unique solution

Returns
(numpy int array of shape (N, 2), numpy float array of shape (N, 3)) – the first array is a list of (j0, i0) indices of columns containing an intersection and the second array is the corresponding list of (x, y, z) intersection points; if the trajectory argument is a list of trajectories, then a correponding list of numpy array pairs is returned

Notes

interseections are found based on straight line segments between the trajectory control points, this will result in errors where there is significant curvature between neighbouring control points; a given (j0, i0) column might appear more than once in the first returned array

7.6.14 resqpy.grid_surface.find_intersections_of_trajectory_with_surface

resqpy.grid_surface.**find_intersections_of_trajectory_with_surface**(*trajectory*, *surface*)

Returns an array of triangle indices and an array of xyz of intersections of well trajectory with surface.

Parameters

- **trajectory** (*well.Trajectory object; or list thereof*) – the wellbore trajectory object(s) to find the intersections for; if a list of trajectories is provided then the return value is a corresponding list
- **surface** (*surface.Surface object*) – the triangulated surface with which to intersect the trajectory

Returns

(numpy int array of shape (N,), numpy float array of shape (N, 3)) – the first array is a list of surface triangle indices containing an intersection and the second array is the corresponding list of (x, y, z) intersection points; if the trajectory argument is a list of trajectories, then a correponding list of numpy array pairs is returned

Notes

interseections are found based on straight line segments between the trajectory control points, this will result in errors where there is significant curvature between neighbouring control points; a given triangle index might appear more than once in the first returned array

7.6.15 resqpy.grid_surface.generate_surface_for_blocked_well_cells

resqpy.grid_surface.**generate_surface_for_blocked_well_cells**(*blocked_well*, *combined=True*, *active_only=False*, *min_k0=0*, *max_k0=None*, *depth_limit=None*, *quad_triangles=True*)

Returns a surface or list of surfaces representing the faces of the cells visited by the well.

7.6.16 resqpy.grid_surface.generate_torn_surface_for_layer_interface

resqpy.grid_surface.**generate_torn_surface_for_layer_interface**(*grid*, *k0=0*, *ref_k_faces='top'*, *quad_triangles=True*)

Returns a Surface object generated from the grid layer interface points.

Parameters

- **grid** (*grid.Grid object*) – the grid object from which a layer interface is to be converted to a surface
- **k0** (*int*) – the layer number (zero based) to be used
- **ref_k_faces** (*string*) – either ‘top’ (the default) or ‘base’, indicating whether the top or the base interface of the layer is to be used
- **quad_triangles** (*boolean, optional, default True*) – if True, 4 triangles are used to represent each cell k face, which gives a unique solution with a shared node of the 4 triangles at the mean point of the 4 corners of the face; if False, only 2 triangles are used, which gives a non-unique solution

Returns
a resqml_surface.Surface object with a single triangulated patch

Notes

The resulting surface is assigned to the same model as grid, though xml is not generated and hdf5 is not written. Strictly, the RESQML business rules for a triangulated surface require a separate patch for areas of the surface which are not joined; therefore, if fault tears cut off one area of the surface (eg. a fault running fully across the grid), then more than one patch should be generated; however, at present the code uses a single patch regardless.

7.6.17 resqpy.grid_surface.generate_torn_surface_for_x_section

resqpy.grid_surface.**generate_torn_surface_for_x_section**(*grid*, *axis*, *ref_slice0=0*, *plus_face=False*, *quad_triangles=True*, *as_single_layer=False*)

Returns a Surface object generated from the grid cross section points.

Parameters

- **grid** (*grid.Grid object*) – the grid object from which a cross section is to be converted to a surface
- **axis** (*string*) – 'I' or 'J' being the axis of the cross-sectional slice (ie. dimension being dropped)
- **ref_slice0** (*int, default 0*) – the reference value for indices in I or J (as defined in axis)
- **plus_face** (*boolean, default False*) – if False, negative face is used; if True, positive
- **quad_triangles** (*boolean, default True*) – if True, 4 triangles are used to represent each cell face, which gives a unique solution with a shared node of the 4 triangles at the mean point of the 4 corners of the face; if False, only 2 triangles are used, which gives a non-unique solution
- **as_single_layer** (*boolean, default False*) – if True, the top points from the top layer are used together with the basal points from the base layer, to effect a single layer equivalent cross section surface

Returns
a resqml_surface.Surface object with a single triangulated patch

Notes

The resulting surface is assigned to the same model as grid, though xml is not generated and hdf5 is not written. Strictly, the RESQML business rules for a triangulated surface require a separate patch for areas of the surface which are not joined; therefore, a fault running down through the grid should result in separate patches; however, at present the code uses a single patch regardless.

7.6.18 resqpy.grid_surface.generate_untorn_surface_for_layer_interface

resqpy.grid_surface.**generate_untorn_surface_for_layer_interface**(*grid*, *k0=0*, *ref_k_faces='top'*, *quad_triangles=True*, *border=None*)

Returns a Surface object generated from the grid layer interface points after any faults are 'healed'.

Parameters

- **grid** (*grid.Grid object*) – the grid object from which a layer interface is to be converted to a surface
- **k0** (*int*) – the layer number (zero based) to be used
- **ref_k_faces** (*string*) – either 'top' (the default) or 'base', indicating whether the top or the base interface of the layer is to be used
- **quad_triangles** (*boolean, optional, default True*) – if True, 4 triangles are used to represent each cell k face, which gives a unique solution with a shared node of the 4 triangles at the mean point of the 4 corners of the face; if False, only 2 triangles are used, which gives a non-unique solution
- **border** (*float, optional*) – If given, an extra border row of quadrangles is added around the grid mesh

Returns

a resqml_surface.Surface object with a single triangulated patch

Notes

The resulting surface is assigned to the same model as grid, though xml is not generated and hdf5 is not written. If a border is specified and the outer grid cells have non-parallel edges, the resulting mesh might be messed up.

7.6.19 resqpy.grid_surface.generate_untorn_surface_for_x_section

resqpy.grid_surface.**generate_untorn_surface_for_x_section**(*grid*, *axis*, *ref_slice0=0*, *plus_face=False*, *quad_triangles=True*, *as_single_layer=False*)

Returns a Surface object generated from the grid cross section points for an unfaulted grid.

Parameters

- **grid** (*grid.Grid object*) – the grid object from which a cross section is to be converted to a surface
- **axis** (*string*) – 'I' or 'J' being the axis of the cross-sectional slice (ie. dimension being dropped)
- **ref_slice0** (*int, default 0*) – the reference value for indices in I or J (as defined in axis)
- **plus_face** (*boolean, default False*) – if False, negative face is used; if True, positive
- **quad_triangles** (*boolean, default True*) – if True, 4 triangles are used to represent each cell face, which gives a unique solution with a shared node of the 4 triangles at the mean point of the 4 corners of the face; if False, only 2 triangles are used, which gives a non-unique solution

- **as_single_layer** (*boolean, default False*) – if True, the top points from the top layer are used together with the basal points from the base layer, to effect a single layer equivalent cross section surface

Returns
a resqml_surface.Surface object with a single triangulated patch

Notes

The resulting surface is assigned to the same model as grid, though xml is not generated and hdf5 is not written. Strictly, the RESQML business rules for a triangulated surface require a separate patch for areas of the surface which are not joined; therefore, a fault running down through the grid should result in separate patches; however, at present the code uses a single patch regardless.

7.6.20 resqpy.grid_surface.get_boundary

resqpy.grid_surface.**get_boundary**(*k_faces: ndarray*, *j_faces: ndarray*, *i_faces: ndarray*, *grid_extent_kji: Tuple[int, int, int]*) → Dict[str, int]

Cretaes a dictionary of the indices that bound the surface (where the faces are True).

Parameters

- **k_faces** (*np.ndarray*) – a boolean array of which faces represent the surface in the k dimension
- **j_faces** (*np.ndarray*) – a boolean array of which faces represent the surface in the j dimension
- **i_faces** (*np.ndarray*) – a boolean array of which faces represent the surface in the i dimension
- **grid_extent_kji** (*Tuple[int, int, int]*) – the shape of the grid

Returns
boundary (Dict[str, int]) – a dictionary of the indices that bound the surface

---

Note: input faces arrays are for internal grid faces (ie. extent reduced by 1 in axis of faces); a buffer slice is included where the surface does not reach the edge of the grid

---

7.6.21 resqpy.grid_surface.intersect_numba

resqpy.grid_surface.**intersect_numba**(*axis: int*, *index1: int*, *index2: int*, *hits: ndarray*, *n_axis: int*, *points: ndarray*, *triangles: ndarray*, *grid_dxyz: Tuple[float]*, *faces: ndarray*, *return_depths: bool*, *depths: ndarray*, *return_offsets: bool*, *offsets: ndarray*, *return_triangles: bool*, *triangle_per_face: ndarray*) → Tuple[ndarray, ndarray, ndarray]

Finds the faces that intersect the surface in 3D.

Parameters

- **axis** (*int*) – axis number. Axis i is 0, j is 1, and k is 2.
- **index1** (*int*) – the first index. Axis i is 0, j is 0, and k is 1.
- **index2** (*int*) – the second index. Axis i is 1, j is 2, and k is 2.

- **hits** (*np.ndarray*) – boolean array of grid centres that intersected the surface for the given axis.
- **n_axis** (*int*) – number of cells in the axis.
- **points** (*np.ndarray*) – array of all the surface node points in 3D.
- **triangles** (*np.ndarray*) – array of all the points indices creating each triangle.
- **grid_dxyz** (*Tuple[float]*) – tuple of a cell's thickness in each axis.
- **faces** (*np.ndarray*) – boolean array of each cell face that can represent the surface.
- **return_depths** (*bool*) – if True, an array of the depths is populated.
- **depths** (*np.ndarray*) – array of the z values of the intersection point of the inter-cell centre vector with a triangle in the surface.
- **return_offsets** (*bool*) – if True, an array of the offsets is calculated.
- **offsets** (*np.ndarray*) – array of the distance between the centre of the cell face and the intersection point of the inter-cell centre vector with a triangle in the surface.
- **return_triangles** (*bool*) – if True, an array of triangle indices is returned.

Returns

Tuple containing –

- faces (np.ndarray): boolean array of each cell face that can represent the surface.
- **offsets (np.ndarray): array of the distance between the centre of the cell face and the** intersection point of the inter-cell centre vector with a triangle in the surface.
- triangle_per_face (np.ndarray): array of triangle numbers

7.6.22 resqpy.grid_surface.point_is_within_cell

resqpy.grid_surface.**point_is_within_cell**(*xyz*, *grid*, *kji0*, *cell_surface=None*, *false_on_pinchout=True*)

Returns True if point xyz is within cell kji0, but not on its surface.

7.6.23 resqpy.grid_surface.populate_blocked_well_from_trajectory

resqpy.grid_surface.**populate_blocked_well_from_trajectory**(*blocked_well*, *grid*, *active_only=False*, *quad_triangles=True*, *lazy=False*, *use_single_layer_tactics=True*, *check_for_reentry=True*)

Populate an empty blocked well object based on the intersection of its trajectory with a grid.

Parameters

- **blocked_well** (*resqpy.well.BlockedWell object*) – a largely empty blocked well object to be populated by this function; note that the trajectory attribute must be set before calling this function
- **grid** (*resqpy.grid.Grid object*) – the grid to intersect the well trajectory with
- **active_only** (*boolean, default False*) – if True, intervals which cover inactive cells will be set as unblocked intervals; if False, intervals covering any cell will be set as blocked intervals

- **quad_triangles** (*boolean, default True*) – if True, each cell face is represented by 4 triangles when calculating intersections; if False, only 2 triangles are used
- **lazy** (*boolean, default False*) – if True, initial penetration must be through a top K face and blocking will cease as soon as the trajectory first leaves the gridded volume; if False, initial entry may be through any external face or a fault face and re-entry will be handled
- **use_single_layer_tactics** (*boolean, default True*) – if True and not lazy and the grid does not have k gaps, fault planes and grid sidewall are initially treated as if the grid were a single layer, when looking for penetrations from outwith the grid
- **check_for_reentry** (*boolean, default True*) – if True, the trajectory is tracked after leaving the grid through the outer skin (eg. base reservoir) in case of re-entry; if False, blocking stops upon the first exit of the trajectory through the skin; ignored (treated as False) if lazy is True

Returns

the blocked well object (same object as passed in) if successful; None if unsuccessful

Notes

the blocked_well trajectory attribute must be set before calling this function; grids with k gaps might result in very slow processing; the function represents a cell face as 2 or 4 triangles rather than a bilinear patch; setting quad_triangles False is not recommended as the 2 triangle formulation gives a non-unique representation of a face (though the code is designed to use the same representation for a shared face between neighbouring cells); where non-planar faults exist, the triangulation of faces may result in a small misalignment of abutted faces between the opposing sides of the fault; this could potentially result in an extra, small, unblocked interval being introduced as an artefact between the exit point of one cell and the entry point into the abutted cell

7.6.24 resqpy.grid_surface.shadow_from_faces

resqpy.grid_surface.**shadow_from_faces**(*extent_kji*, *k_faces*)

Returns a numpy int8 array indicating whether cells are above, below or between K faces.

Parameters

- **extent_kji** (*triple int*) – the shape of the grid
- **k_faces** (*bool array*) – True where a K face is present; shaped (nk - 1, nj, ni)

Returns

numpy int8 array of shape extent_kji; values are –

0 neither above nor below a K face;
1: above any K faces in the column; 2 below any K faces in the column; 3: between K faces (one or more above and one or more below)

7.6.25 resqpy.grid_surface.trajectory_grid_overlap

resqpy.grid_surface.**trajectory_grid_overlap**(*trajectory*, *grid*, *lazy=False*)

Returns True if there is some overlap of the xyz boxes for the trajectory and grid, False otherwise.

Notes

overlap of the xyz boxes does not guarantee that the trajectory intersects the grid; a return value of False guarantees that the trajectory does not intersect the grid

7.7 resqpy.lines

Polyline and PolylineSet classes and associated functions.

Classes

Polyline	Class for RESQML polyline representation.
PolylineSet	Class for RESQML polyline set representation.

Functions

flatten_polyline	Returns a new polyline object, flattened (projected) on a chosen axis to a given value.
load_hdf5_array	Loads the property array data as an attribute of object, from the hdf5 referenced in xml node.
shift_polyline	Returns a new polyline object, shifted by given coordinates.
spline	Returns a numpy array containing resampled cubic spline of line defined by points.
tangents	Returns a numpy array of tangent unit vectors for an ordered list of points.

7.7.1 resqpy.lines.flatten_polyline

resqpy.lines.**flatten_polyline**(*parent_model*, *poly_root*, *axis='z'*, *value=0.0*, *title=''*)

Returns a new polyline object, flattened (projected) on a chosen axis to a given value.

7.7.2 resqpy.lines.load_hdf5_array

resqpy.lines.**load_hdf5_array**(*object*, *node*, *array_attribute*, *tag='Values'*, *dtype='float'*)

Loads the property array data as an attribute of object, from the hdf5 referenced in xml node.

7.7.3 resqpy.lines.shift_polyline

resqpy.lines.**shift_polyline**(*parent_model*, *poly_root*, *xyz_shift=(0, 0, 0)*, *title=''*)

Returns a new polyline object, shifted by given coordinates.

7.7.4 resqpy.lines.spline

resqpy.lines.**spline**(*points*, *tangent_vectors=None*, *tangent_weight='square'*, *min_subdivisions=1*, *max_segment_length=None*, *max_degrees_per_knot=5.0*, *closed=False*)

Returns a numpy array containing resampled cubic spline of line defined by points.

Parameters

- **points** (*numpy float array of shape (N, 3)*) – points defining a line
- **tangent_vectors** (*numpy float array of shape (N, 3), optional*) – vectors to use in the construction of the cubic spline; if None, tangents are calculated
- **tangent_weight** (*string, default 'linear'*) – one of 'linear', 'square' or 'cube', giving increased weight to relatively shorter of 2 line segments at each knot when computing tangent vectors; ignored if tangent_vectors is not None
- **min_subdivisions** (*int, default 1*) – the resulting line will have at least this number of segments per original line segment
- **max_segment_length** (*float, optional*) – if present, resulting line segments will not exceed this length by much (see notes)
- **max_degrees_per_knot** (*float, default 5.0*) – the change in direction at each resulting knot will not usually exceed this value (see notes)
- **closed** (*boolean, default False*) – if True, the points are treated as a closed polyline with regard to end point tangents, otherwise as an open line

Returns

numpy float array of shape (>=N, 3) being knots on a cubic spline defined by points; original points are a subset of returned knots

Notes

the max_segment_length argument, if present, is compared with the length of each original segment to give a lower bound on the number of derived line segments; as the splined line may have extra curvature, the length of individual segments in the returned line can exceed the argument value, though usually not by much similarly, the max_degrees_per_knot is compared to the original deviations to provide another lower bound on the number of derived line segments for each original segment; as the spline may divide the segment unequally and also sometimes add loops, the resulting deviations can exceed the argument value

7.7.5 resqpy.lines.tangents

resqpy.lines.**tangents**(*points*, *weight='linear'*, *closed=False*)

Returns a numpy array of tangent unit vectors for an ordered list of points.

Parameters

- **points** (*numpy float array of shape (N, 3)*) – points defining a line
- **weight** (*string, default 'linear'*) – one of 'linear', 'square' or 'cube', giving increased weight to relatively shorter of 2 line segments at each knot
- **closed** (*boolean, default False*) – if True, the points are treated as a closed polyline with regard to end point tangents, otherwise as an open line

Returns

numpy float array of the same shape as points, containing a unit length tangent vector for each knot (point)

Note: if two neighbouring points are identical, a divide by zero will occur

7.8 resqpy.multi_processing

Multiprocessing module for running specific functions concurrently.

Functions

blocked_well_from_trajectory_batch	Creates BlockedWell objects from a common grid and a list of trajectories' uuids, in parallel.
blocked_well_from_trajectory_wrapper	Multiprocessing wrapper function of the BlockedWell initialisation from a Trajectory.
find_faces_to_represent_surface_regular_wrapper	Multiprocessing wrapper function of find_faces_to_represent_surface_regular_optimised.
function_multiprocessing	Calls a function concurrently with the specfied arguments.
mesh_from_regular_grid_column_property_batch	Creates Mesh objects from a list of property uuids in parallel.
mesh_from_regular_grid_column_property_wrapper	Multiprocessing wrapper function of the Mesh from_regular_grid_column_property method.

7.8.1 resqpy.multi_processing.blocked_well_from_trajectory_batch

resqpy.multi_processing.**blocked_well_from_trajectory_batch**(*grid_epc: str*, *grid_uuid: Union[UUID, str]*, *trajectory_epc: str*, *trajectory_uuids: List[Union[UUID, str]]*, *recombined_epc: str*, *cluster*, *n_workers: int*, *require_success: bool = False*, *tmp_dir_path: Union[Path, str] = '.'*) → List[bool]

Creates BlockedWell objects from a common grid and a list of trajectories' uuids, in parallel.

Parameters

- **grid_epc** (*str*) – epc file path where the grid is saved
- **grid_uuid** (*UUID or str*) – UUID (universally unique identifier) of the grid object
- **trajectory_epc** (*str*) – epc file path where the trajectories are saved
- **trajectory_uuids** (*list of UUID or str*) – a list of the trajectory uuids used to create each Trajectory object
- **recombined_epc** (*Path or str*) – A pathlib Path or path string, where the combined epc will be saved
- **cluster** (*LocalCluster/JobQueueCluster*) – a LocalCluster is a Dask cluster on a local machine; if using a job queing system, a JobQueueCluster can be used such as an SGECluster, SLURMCluster, PBSCluster, LSFCluster etc
- **n_workers** (*int*) – the number of workers on the cluster
- **require_success** (*bool, default False*) – if True an exception is raised if any failures
- **tmp_dir_path** (*str or Path, default '.'*) – the directory within which temporary directories will reside

Returns

success_list (list of bool) – A boolean list of successful function calls

Notes

the returned success list contains one value per batch, set True if all blocked wells were successfully created in the batch, False if one or more failed in the batch

7.8.2 resqpy.multi_processing.blocked_well_from_trajectory_wrapper

resqpy.multi_processing.**blocked_well_from_trajectory_wrapper**(*index: int*, *parent_tmp_dir: str*, *grid_epc: str*, *grid_uuid: Union[UUID, str]*, *trajectory_epc: str*, *trajectory_uuids: List[Union[UUID, str]]*) → Tuple[int, bool, str, List[Union[UUID, str]]]

Multiprocessing wrapper function of the BlockedWell initialisation from a Trajectory.

Parameters

- **index** (*int*) – the index of the function call from the multiprocessing function
- **parent_tmp_dir** (*str*) – the parent temporary directory path from the multiprocessing function
- **grid_epc** (*str*) – epc file path where the grid is saved
- **grid_uuid** (*UUID or str*) – UUID (universally unique identifier) of the grid object
- **trajectory_epc** (*str*) – epc file path where the trajectories are saved
- **trajectory_uuids** (*list of UUID or str*) – a list of the trajectory uuids used to create each Trajectory object

Returns

Tuple containing –

- index (int): the index passed to the function;
- success (bool): True if all the BlockedWell objects could be created, False otherwise;
- epc_file (str): the epc file path where the objects are stored;
- uuid_list (List[UUID/str]): list of UUIDs of relevant objects

---

Note: used this wrapper when calling the multiprocessing function to initialise blocked wells from trajectories; it will create a new model that is saved in a temporary epc file and returns the required values, which are used in the multiprocessing function to recombine all the objects into a single epc file

---

7.8.3 resqpy.multi_processing.find_faces_to_represent_surface_regular_wrapper

resqpy.multi_processing.**find_faces_to_represent_surface_regular_wrapper**(*index: int*, *parent_tmp_dir: str*, *use_index_as_realisation: bool*, *grid_epc: str*, *grid_uuid: Union[UUID, str]*, *surface_epc: str*, *surface_uuid: Union[UUID, str]*, *name: str*, *title: Optional[str] = None*, *agitate: bool = False*, *random_agitation: bool = False*, *feature_type: str = 'fault'*, *trimmed: bool = False*, *is_curtain=False*, *extend_fault_representation: bool = False*, *flange_inner_ring=False*, *saucer_parameter=None*, *retriangulate: bool = False*, *related_uuid=None*, *progress_fn: Optional[Callable] = None*, *extra_metadata=None*, *return_properties: Optional[List[str]] = None*, *raw_bisector: bool = False*, *use_pack: bool = False*) → Tuple[int, bool, str, List[Union[UUID, str]]]

Multiprocessing wrapper function of find_faces_to_represent_surface_regular_optimised.

Parameters

- **index** (*int*) – the index of the function call from the multiprocessing function
- **parent_tmp_dir** (*str*) – the parent temporary directory path from the multiprocessing function
- **use_index_as_realisation** (*bool*) – if True, uses the index number as the realization number on the property collection
- **grid_epc** (*str*) – epc file path where the grid is saved
- **grid_uuid** (*UUID or str*) – UUID (universally unique identifier) of the grid object
- **surface_epc** (*str*) – epc file path where the surface (or point set) is saved
- **surface_uuid** (*UUID or str*) – UUID (universally unique identifier) of the surface (or point set) object.

- **name** (*str*) – the feature name to use in the grid connection set.
- **title** (*str*) – the citation title to use for the grid connection set; defaults to name
- **agitate** (*bool*) – if True, the points of the surface are perturbed by a small offset, which can help if the surface has been built from a regular mesh with a periodic resonance with the grid
- **random_agitation** (*bool, default False*) – if True, the agitation is by a small random distance; if False, a constant positive shift of 5.0e-6 is applied to x, y & z values; ignored if agitate is False
- **feature_type** (*str, default 'fault'*) – one of 'fault', 'horizon', or 'geobody boundary'
- **trimmed** (*bool, default True*) – if True the surface has already been trimmed
- **is_curtain** (*bool, default False*) – if True, only the top layer is intersected with the surface and bisector is generated as a column property if requested
- **extend_fault_representation** (*bool, default False*) – if True, the representation is extended with a flange
- **retriangulate** (*bool, default False*) – if True, a retriangulation is performed even if not needed otherwise
- **related_uuid** (*uuid, optional*) – if present, the uuid of an object to be softly related to the gcs (and to grid bisector and/or shadow property if requested)
- **progress_fn** (*Callable*) – a callback function to be called at intervals by this function; the argument will progress from 0.0 to 1.0 in unspecified and uneven increments
- **extra_metadata** (*dict, optional*) – extra metadata items to be added to the grid connection set
- **return_properties** (*list of str*) – if present, a list of property arrays to calculate and return as a dictionary; recognised values in the list are 'triangle', 'depth', 'offset', 'normal vector', 'flange bool', 'grid bisector' and 'grid shadow'; triangle is an index into the surface triangles of the triangle detected for the gcs face; depth is the z value of the intersection point of the inter-cell centre vector with a triangle in the surface; offset is a measure of the distance between the centre of the cell face and the intersection point; normal vector is a unit vector normal to the surface triangle; each array has an entry for each face in the gcs; grid bisector is a grid cell boolean property holding True for the set of cells on one side of the surface, deemed to be shallower; grid shadow is a grid cell int8 property holding 1 for cells neither above nor below a K face, 1 for above, 2 for beneath, 3 for between; the returned dictionary has the passed strings as keys and numpy arrays as values
- **raw_bisector** (*bool, default False*) – if True and grid bisector is requested then it is left in a raw form without assessing which side is shallower (True values indicate same side as origin cell)
- **use_pack** (*bool, default False*) – if True, boolean properties will be stored in numpy packed format, which will only be readable by resqpy based applications

Returns

Tuple containing – - index (int): the index passed to the function - success (bool): whether the function call was successful, whatever that definiton is - epc_file (str): the epc file path where the objects are stored - uuid_list (List[str]): list of UUIDs of relevant objects

Notes

Use this function as argument to the multiprocessing function; it will create a new model that is saved in a temporary epc file and returns the required values, which are used in the multiprocessing function to recombine all the objects into a single epc file

7.8.4 resqpy.multi_processing.function_multiprocessing

resqpy.multi_processing.**function_multiprocessing**(*function: Callable*, *kwargs_list: List[Dict[str, Any]]*, *recombined_epc: Union[Path, str]*, *cluster*, *consolidate: bool = True*, *require_success=False*, *tmp_dir_path: Union[Path, str] = '.'*, *backend: str = 'dask'*, *clean_up: bool = True*) → List[bool]

Calls a function concurrently with the specfied arguments.

Parameters

- **function** (*Callable*) – the wrapper function to be called; needs to return: - index (int): the index of the kwargs in the kwargs_list; - success (bool): whether the function call was successful, however that is defined; - epc_file (Path or str): the epc file path where the objects are stored; - uuid_list (list of str): list of UUIDs of relevant objects;
- **kwargs_list** (*list of dict*) – A list of keyword argument dictionaries that are used when calling the function
- **recombined_epc** (*Path or str*) – A pathlib Path or path string of where the combined epc will be saved
- **cluster** – if using the Dask backend, a LocalCluster is a Dask cluster on a local machine. If using a job queing system, a JobQueueCluster can be used such as an SGECluster, SLURMCluster, PBSCluster, LSFCluster etc
- **consolidate** (*bool*) – if True and an equivalent part already exists in a model, it is not duplicated and the uuids are noted as equivalent
- **require_success** (*bool*) – if True and any instance fails, then an exception is raised
- **tmp_dir_path** (*str*) – path where the temporary directory is saved; defaults to the calling code directory
- **backend** (*str*) – the joblib parallel backend used. Dask is used by default so a Dask cluster must be passed to the cluster argument
- **clean_up** (*bool, default True*) – if True, the temporary directory used during multi processing is deleted; if False, it is left in place with its contents (to facilitate debugging)

Returns

success_list (List[bool]) – A boolean list of successful function calls

Notes

a multiprocessing pool is used to call the function multiple times in parallel; once all results are returned, they are combined into a single epc file; this function uses the Dask backend by default to run the given function in parallel, so a Dask cluster must be setup and passed as an argument if Dask is used; Dask will need to be installed in the Python environment because it is not a dependency of the project; more info can be found at https://resqpy.readthedocs.io/en/latest/tutorial/multiprocessing.html

7.8.5 resqpy.multi_processing.mesh_from_regular_grid_column_property_batch

resqpy.multi_processing.**mesh_from_regular_grid_column_property_batch**(*grid_epc: str*, *grid_uuid: Union[UUID, str]*, *prop_uuids: List[Union[UUID, str]]*, *recombined_epc: str*, *cluster*, *n_workers: int*, *require_success: bool = False*, *tmp_dir_path: Union[Path, str] = '.'*) → List[bool]

Creates Mesh objects from a list of property uuids in parallel.

Parameters

- **grid_epc** (*str*) – epc file path where the grid is saved
- **grid_uuid** (*UUID/str*) – UUID (universally unique identifier) of the grid object
- **prop_uuids** (*List[UUID/str]*) – a list of the column property uuids used to create each Mesh and their relationship
- **recombined_epc** (*Path/str*) – A pathlib Path or path string of where the combined epc will be saved
- **cluster** (*LocalCluster/JobQueueCluster*) – a LocalCluster is a Dask cluster on a local machine. If using a job queing system, a JobQueueCluster can be used such as an SGECluster, SLURMCluster, PBSCluster, LSFCluster etc
- **n_workers** (*int*) – the number of workers on the cluster
- **require_success** (*bool, default False*) – if True an exception is raised if any failures
- **tmp_dir_path** (*str or Path, default '.'*) – the directory within which temporary directories will reside

Returns

success_list (list of bool) – A boolean list of successful function calls; one value per batch

7.8.6 resqpy.multi_processing.mesh_from_regular_grid_column_property_wrapper

resqpy.multi_processing.**mesh_from_regular_grid_column_property_wrapper**(*index: int*, *parent_tmp_dir: str*, *grid_epc: str*, *grid_uuid: Union[UUID, str]*, *prop_uuids: List[Union[UUID, str]]*) → Tuple[int, bool, str, List[Union[UUID, str]]]

Multiprocessing wrapper function of the Mesh from_regular_grid_column_property method.

Parameters

- **index** (*int*) – the index of the function call from the multiprocessing function
- **parent_tmp_dir** (*str*) – the parent temporary directory path from the multiprocessing function
- **grid_epc** (*str*) – epc file path where the grid is saved
- **grid_uuid** (*UUID or str*) – UUID (universally unique identifier) of the regular grid object
- **prop_uuids** (*list of UUID or str*) – a list of the property uuids used to create each Mesh and their relationship

Returns

Tuple containing – - index (int): the index passed to the function - success (bool): True if all the Mesh objects could be created, False otherwise - epc_file (str): the epc file path where the objects are stored - uuid_list (List[UUID/str]): list of UUIDs of relevant objects

Notes

Use this function as argument to multiprocessing function; it will create a new model that is saved in a temporary epc file and returns the required values, which are used in the multiprocessing function to recombine all the objects into a single epc file; applications should generally call mesh_from_regular_grid_column_property_batch() which makes use of this wrapper

resqpy.multi_processing.wrappers	Wrapper module for the wrapper functions used in multiprocessing.

7.8.7 resqpy.multi_processing.wrappers

Wrapper module for the wrapper functions used in multiprocessing.

resqpy.multi_processing.wrappers.blocked_well_mp	Multiprocessing wrapper functions for the well/BlockedWell class.
resqpy.multi_processing.wrappers.grid_surface_mp	Multiprocessing wrapper functions for the grid_surface module.
resqpy.multi_processing.wrappers.mesh_mp	Multiprocessing wrapper functions for the Mesh class.

resqpy.multi_processing.wrappers.blocked_well_mp

Multiprocessing wrapper functions for the well/BlockedWell class.

resqpy.multi_processing.wrappers.grid_surface_mp

Multiprocessing wrapper functions for the grid_surface module.

Functions

inherit_interpretation_relationship	Inherit a relationship to an interpretation object if present for old representation.

resqpy.multi_processing.wrappers.grid_surface_mp.inherit_interpretation_relationship

resqpy.multi_processing.wrappers.grid_surface_mp.**inherit_interpretation_relationship**(*model*, *old_repr_uuid*, *new_repr_uuid*)

Inherit a relationship to an interpretation object if present for old representation.

resqpy.multi_processing.wrappers.mesh_mp

Multiprocessing wrapper functions for the Mesh class.

7.9 resqpy.organize

Organizational object classes: features and interpretations.

Classes

BoundaryFeature	Class for RESQML Boundary Feature organizational objects.
BoundaryFeatureInterpretation	Class for RESQML Horizon Interpretation organizational objects.
EarthModelInterpretation	Class for RESQML Earth Model Interpretation organizational objects.
FaultInterpretation	Class for RESQML Fault Interpretation organizational objects.
FluidBoundaryFeature	Class for RESQML Fluid Boundary Feature (contact) organizational objects.
FrontierFeature	Class for RESQML Frontier Feature organizational objects.
GenericInterpretation	Class for RESQML Generic Feature Interpretation objects.
GeneticBoundaryFeature	Class for RESQML Genetic Boundary Feature (horizon) organizational objects.
GeobodyBoundaryInterpretation	Class for RESQML Geobody Boundary Interpretation organizational objects.
GeobodyFeature	Class for RESQML Geobody Feature objects (note: definition may be incomplete in RESQML 2.0.1).
GeobodyInterpretation	Class for RESQML Geobody Interpretation objects.
GeologicUnitFeature	Class for RESQML Geologic Unit Feature organizational objects.
HorizonInterpretation	Class for RESQML Horizon Interpretation organizational objects.
OrganizationFeature	Class for generic RESQML Organization Feature objects.
RockFluidUnitFeature	Class for RESQML Rock Fluid Unit Feature organizational objects.
TectonicBoundaryFeature	Class for RESQML Tectonic Boundary Feature (fault) organizational objects.
WellboreFeature	Class for RESQML Wellbore Feature organizational objects.
WellboreInterpretation	Class for RESQML Wellbore Interpretation organizational objects.

Functions

alias_for_attribute	Return an attribute that is a direct alias for an existing attribute.
create_xml_has_occurred_during	Creates XML sub-tree 'HasOccuredDuring' node
equivalent_chrono_pairs	Returns True if the two chronostratigraphic pairs are equivalent
equivalent_extra_metadata	Returns True if the two objects have identical extra metadata
extract_has_occurred_during	Extracts UUIDs of chrono bottom and top from xml for has occurred during sub-node, or (None, None).

7.9.1 resqpy.organize.alias_for_attribute

resqpy.organize.**alias_for_attribute**(*attribute_name*)

Return an attribute that is a direct alias for an existing attribute.

7.9.2 resqpy.organize.create_xml_has_occurred_during

resqpy.organize.**create_xml_has_occurred_during**(*model*, *parent_node*, *hod_pair*, *tag='HasOccuredDuring'*)

Creates XML sub-tree 'HasOccuredDuring' node

7.9.3 resqpy.organize.equivalent_chrono_pairs

resqpy.organize.**equivalent_chrono_pairs**(*pair_a*, *pair_b*, *model=None*)

Returns True if the two chronostratigraphic pairs are equivalent

7.9.4 resqpy.organize.equivalent_extra_metadata

resqpy.organize.**equivalent_extra_metadata**(*a*, *b*)

Returns True if the two objects have identical extra metadata

7.9.5 resqpy.organize.extract_has_occurred_during

resqpy.organize.**extract_has_occurred_during**(*parent_node*, *tag='HasOccuredDuring'*)

Extracts UUIDs of chrono bottom and top from xml for has occurred during sub-node, or (None, None).

resqpy.organize.boundary_feature	Class for RESQML Boundary Feature organizational objects.
resqpy.organize.boundary_feature_interpretation	Class for RESQML Boundary Feature Interpretation organizational objects.
resqpy.organize.earth_model_interpretation	Class for RESQML Earth Model Interpretation organizational objects.
resqpy.organize.fault_interpretation	Class for RESQML Fault Interpretation organizational objects.
resqpy.organize.fluid_boundary_feature	Class for RESQML Fluid Boundary Feature (contact) organizational objects.
resqpy.organize.frontier_feature	Class for RESQML Frontier Feature organizational objects.
resqpy.organize.generic_interpretation	Class for RESQML Generic Feature Interpretation objects.
resqpy.organize.genetic_boundary_feature	Class for RESQML Genetic Boundary Feature (horizon) organizational objects.
resqpy.organize.geobody_boundary_interpretation	Class for RESQML Geobody Boudary Interpretation organizational objects.
resqpy.organize.geobody_feature	RESQML Feature and Interpretation classes.
resqpy.organize.geobody_interpretation	Class for RESQML Geobody Interpretation objects.
resqpy.organize.geologic_unit_feature	Class for RESQML Geologic Unit Feature organizational objects.
resqpy.organize.horizon_interpretation	Class for RESQML Horizon Interpretation organizational objects.
resqpy.organize.organization_feature	Class for generic RESQML Organization Feature objects.
resqpy.organize.rock_fluid_unit_feature	Class for RESQML Rock Fluid Unit Feature organizational objects.
resqpy.organize.structural_organization_interpretation	Class for RESQML Structural Organization Interpretation organizational objects.
resqpy.organize.tectonic_boundary_feature	Class for RESQML Tectonic Boundary Feature (fault) organizational objects.
resqpy.organize.wellbore_feature	Class for RESQML Wellbore Feature organizational objects.
resqpy.organize.wellbore_interpretation	Class for RESQML Wellbore Interpretation organizational objects.

7.9.6 resqpy.organize.boundary_feature

Class for RESQML Boundary Feature organizational objects.

7.9.7 resqpy.organize.boundary_feature_interpretation

Class for RESQML Boundary Feature Interpretation organizational objects.

7.9.8 resqpy.organize.earth_model_interpretation

Class for RESQML Earth Model Interpretation organizational objects.

7.9.9 resqpy.organize.fault_interpretation

Class for RESQML Fault Interpretation organizational objects.

7.9.10 resqpy.organize.fluid_boundary_feature

Class for RESQML Fluid Boundary Feature (contact) organizational objects.

7.9.11 resqpy.organize.frontier_feature

Class for RESQML Frontier Feature organizational objects.

7.9.12 resqpy.organize.generic_interpretation

Class for RESQML Generic Feature Interpretation objects.

7.9.13 resqpy.organize.genetic_boundary_feature

Class for RESQML Genetic Boundary Feature (horizon) organizational objects.

7.9.14 resqpy.organize.geobody_boundary_interpretation

Class for RESQML Geobody Boudary Interpretation organizational objects.

7.9.15 resqpy.organize.geobody_feature

RESQML Feature and Interpretation classes.

7.9.16 resqpy.organize.geobody_interpretation

Class for RESQML Geobody Interpretation objects.

7.9.17 resqpy.organize.geologic_unit_feature

Class for RESQML Geologic Unit Feature organizational objects.

7.9.18 resqpy.organize.horizon_interpretation

Class for RESQML Horizon Interpretation organizational objects.

7.9.19 resqpy.organize.organization_feature

Class for generic RESQML Organization Feature objects.

7.9.20 resqpy.organize.rock_fluid_unit_feature

Class for RESQML Rock Fluid Unit Feature organizational objects.

7.9.21 resqpy.organize.structural_organization_interpretation

Class for RESQML Structural Organization Interpretation organizational objects.

Classes

`StructuralOrganizationInterpretation`	Class for RESQML Structural Organization Interpretation organizational objects.

7.9.22 resqpy.organize.tectonic_boundary_feature

Class for RESQML Tectonic Boundary Feature (fault) organizational objects.

7.9.23 resqpy.organize.wellbore_feature

Class for RESQML Wellbore Feature organizational objects.

7.9.24 resqpy.organize.wellbore_interpretation

Class for RESQML Wellbore Interpretation organizational objects.

7.10 resqpy.property

Collections of properties for grids, wellbore frames, grid connection sets etc.

Classes

GridPropertyCollection	Class for RESQML Property collection for an IJK Grid, inheriting from PropertyCollection.
Property	Class for an individual property object; uses a single element PropertyCollection behind the scenes.
PropertyCollection	Class for RESQML Property collection for any supporting representation (or mix of supporting representations).
PropertyKind	Class catering for RESQML bespoke PropertyKind objects.
StringLookup	Class catering for RESQML obj_StringLookupTable objects.
WellIntervalProperty	Thin wrapper class around interval properties for a Wellbore Frame or Blocked Wellbore.
WellIntervalPropertyCollection	Class for RESQML property collection for a WellboreFrame for interval or blocked well logs
WellLog	Thin wrapper class around RESQML properties for well logs.
WellLogCollection	Class for RESQML Property collection for a Wellbore Frame (ie well logs), inheriting from PropertyCollection.

7.10.1 resqpy.property.PropertyCollection

class resqpy.property.**PropertyCollection**(*support=None*, *property_set_root=None*, *realization=None*)

Bases: object

Class for RESQML Property collection for any supporting representation (or mix of supporting representations).

Notes

this is a base class inherited by GridPropertyCollection and WellLogCollection (and others to follow), application code usually works with the derived classes; RESQML caters for three simple types of numerical property: Continuous (ie. real data, aka floating point); Discrete (ie. integer data, or boolean); Categorical (integer data, usually non-negative, with an associated look-up table to convert to a string); Points properties are for storing xyz values; resqpy does not currently support Comment properties

Commonly Used Methods:

__init__([support, property_set_root, ...])	Initialise an empty Property Collection, optionally populate properties from a supporting representation.
parts()	Return list of parts in this collection.
uuids()	Return list of uuids in this collection.
selective_parts_list([realization, ...])	Returns a list of parts filtered by those arguments which are not None.
singleton([realization, support_uuid, uuid, ...])	Returns a single part selected by those arguments which are not None.
single_array_ref([realization, ...])	Returns the array of data for a single part selected by those arguments which are not None.
number_of_parts()	Returns the number of parts (properties) in this collection.
part_str(part[, include_citation_title])	Returns a human-readable string identifying the part.
realization_for_part(part)	Returns realization number (within ensemble) that the property relates to.
uuid_for_part(part)	Returns UUID object for the property part.
continuous_for_part(part)	Returns True if the property is continuous (including points); False if it is discrete (or categorical).
indexable_for_part(part)	Returns the text of the IndexableElement for the property part; usually 'cells' for grid properties.
property_kind_for_part(part)	Returns the resqml property kind for the property part.
facet_type_for_part(part)	If relevant, returns the resqml Facet Facet for the property part, eg.
facet_for_part(part)	If relevant, returns the resqml Facet Value for the property part, eg.
citation_title_for_part(part)	Returns the citation title for the property part.
time_index_for_part(part)	If the property has an associated time series (is not static), returns the time index within the time series.
minimum_value_for_part(part)	Returns the minimum value for the property part, as stored in the xml.
maximum_value_for_part(part)	Returns the maximum value for the property part, as stored in the xml.
uom_for_part(part)	Returns the resqml units of measure for the property part.
part_is_categorical(part)	Returns True if the property is categorical (not continuous and has an associated string lookup).
has_multiple_realizations()	Returns the has multiple realizations flag based on whether properties belong to more than one realization.
cached_part_array_ref(part[, dtype, masked, ...])	Returns a numpy array containing the data for the property part; the array is cached in this collection.
realizations_array_ref([use_32_bit, ...])	Returns a +1D array of all parts with first axis being over realizations.
time_series_array_ref([use_32_bit, ...])	Returns a +1D array of all parts with first axis being over time indices.
add_cached_array_to_imported_list(...[, ...])	Caches array and adds to the list of imported properties (but not to the collection dict).

continues on next page

Table 2 – continued from previous page

add_similar_to_imported_list(similar_uuid, ...)	Caches array and adds to the list of imported properties using default metadata from a similar property.
write_hdf5_for_imported_list([file_name, ...])	Create or append to an hdf5 file, writing datasets for the imported arrays.
create_xml_for_imported_list_and_add_parts_to_model([...])	Add imported or generated grid property arrays as parts in parent model, creating xml.
basic_static_property_parts([realization, ...])	Returns five parts: net to gross ratio, porosity, permeability rock I, J & K; each returned part may be None.
basic_static_property_uuids([realization, ...])	Returns five uuids: net to gross ratio, porosity, permeability rock I, J & K; each returned uuid may be None.

Methods:

set_support([support_uuid, support, model, ...])	Sets the supporting object associated with this collection if not done so at initialisation.
supporting_shape([indexable_element, direction])	Return the shape of the supporting representation with respect to the given indexable element
populate_from_property_set(property_set_root)	Populates this (newly created) collection based on xml members of property set.
set_realization(realization)	Sets the model realization number (within an ensemble) for this collection.
add_part_to_dict(part[, continuous, ...])	Add the named part to the dictionary for this collection.
add_parts_list_to_dict(parts_list)	Add all the parts named in the parts list to the dictionary for this collection.
remove_part_from_dict(part)	Remove the named part from the dictionary for this collection.
remove_parts_list_from_dict(parts_list)	Remove all the parts named in the parts list from the dictionary for this collection.
inherit_imported_list_from_other_collection(other)	Extends this collection's imported list with items from other's imported list.
inherit_parts_from_other_collection(other[, ...])	Adds all the parts in the other PropertyCollection to this one.
add_to_imported_list_sampling_other_collection(...)	Makes cut down copies of parts from other collection, using indices, and adds to imported list.
inherit_parts_selectively_from_other_collection(other)	Adds (some) parts from the other PropertyCollection which match all arguments that are not None.
inherit_similar_parts_for_time_series_from_other_collection(...)	Adds the example part from other collection and any other parts for the same property at different times.
inherit_similar_parts_for_facets_from_other_collection(...)	Adds the example part from other collection and any other parts for same property with different facets.
inherit_similar_parts_for_realizations_from_other_collection(...)	Add the example part from other collection and any other parts for same property with different realizations.
number_of_imports()	Returns the number of property arrays in the imported list for this collection.

continues on next page

Table 3 – continued from previous page

part_in_collection(part)	Returns True if named part is member of this collection; otherwise False.
element_for_part(part, index)	Returns a particular element by index from the tuple of metadata for the specified part.
unique_element_list(index[, sort_list])	Returns an optionally sorted list of unique values (excluding None) of an element identified by index.
part_filename(part)	Returns a string which can be used as the starting point of a filename relating to part.
realization_list([sort_list])	Returns a list of unique realization numbers present in the collection.
support_uuid_for_part(part)	Returns supporting representation object's uuid that the property relates to.
grid_for_part(part)	Returns grid object that the property relates to.
node_for_part(part)	Returns the xml node for the property part.
extra_metadata_for_part(part)	Returns the extra_metadata dictionary for the part.
null_value_for_part(part)	Returns the null value for the (discrete) property part; np.NaN for continuous parts.
points_for_part(part)	Returns True if the property is a points property; False otherwise.
all_continuous()	Returns True if all the parts are for continuous (real) properties (includes points).
all_discrete()	Returns True if all the parts are for discrete or categorical (integer) properties.
count_for_part(part)	Returns the Count value for the property part; usually 1.
all_count_one()	Returns True if the low level Count value is 1 for all the parts in the collection.
unique_indexable_element_list([sort_list])	Returns a list of unique values for the IndexableElement of the property parts in the collection.
property_kind_list([sort_list])	Returns a list of unique property kinds found amongst the parts of the collection.
local_property_kind_uuid(part)	Returns the uuid of the bespoke (local) property kind for this part, or None for a standard property kind.
facet_type_list([sort_list])	Returns a list of unique facet types found amongst the parts of the collection.
facet_list([sort_list])	Returns a list of unique facet values found amongst the parts of the collection.
title_for_part(part)	Synonymous with citation_title_for_part().
titles()	Returns a list of citation titles for the parts in the collection.
time_series_uuid_for_part(part)	If the property has an associated time series (is not static), returns the uuid for the time series.
time_series_uuid_list([sort_list])	Returns a list of unique time series uuids found amongst the parts of the collection.
time_index_list([sort_list])	Returns a list of unique time indices found amongst the parts of the collection.
patch_min_max_for_part(part[, minimum, ...])	Updates the minimum and/ox maximum values stored in the metadata, optionally updating xml tree too.
uom_list([sort_list])	Returns a list of unique units of measure found amongst the parts of the collection.

continues on next page

Table 3 – continued from previous page

string_lookup_uuid_for_part(part)	If the property has an associated string lookup (is categorical), return the uuid.
string_lookup_for_part(part)	Returns a StringLookup object for the part, if it has a string lookup uuid, otherwise None.
string_lookup_uuid_list([sort_list])	Returns a list of unique string lookup uuids found amongst the parts of the collection.
constant_value_for_part(part)	Returns the value (float or int) of a constant array part, or None for an hdf5 array.
override_min_max(part, min_value, max_value)	Sets the minimum and maximum values in the metadata for the part.
establish_time_set_kind()	Re-evaulates the time set kind attribute.
time_set_kind()	Returns the time set kind attribute.
establish_has_single_property_kind()	Re-evaluates the has single property kind attribute.
has_single_property_kind()	Return the has single property kind flag depending on whether all properties are of the same kind.
establish_has_single_indexable_element()	Re-evaluate the has single indexable element attribute.
has_single_indexable_element()	Returns the has single indexable element flag depending on whether all properties have the same.
establish_has_multiple_realizations()	Re-evaluates the has multiple realizations attribute.
establish_has_single_uom()	Re-evaluates the has single uom attribute depending on whether all properties have the same units of measure.
has_single_uom()	Returns the has single uom flag depending on whether all properties have the same units of measure.
assign_realization_numbers()	Assigns a distinct realization number to each property, after checking for compatibility.
masked_array(simple_array[, ...])	Returns a masked version of simple_array, using inactive mask associated with support for this property collection.
h5_key_pair_for_part(part)	Return hdf5 key pair (ext uuid, internal path) for the part.
h5_slice(part, slice_tuple)	Returns a subset of the array for part, without loading the whole array.
h5_overwrite_slice(part, slice_tuple, ...[, ...])	Overwrites a subset of the array for part, in the hdf5 file.
shape_and_type_of_part(part)	Returns shape tuple and element type of cached or hdf5 array for part.
facets_array_ref([use_32_bit, ...])	Returns a +1D array of all parts with first axis being over facet values; Use facet_list() for lookup.
combobulated_face_array(resqml_a)	Returns a logically ordered copy of RESQML faces-per-cell property array resqml_a.
discombobulated_face_array(resqpy_a)	Return logical face property array a, re-ordered and reshaped regarding the six facial directions.
normalized_part_array(part[, masked, ...])	Return data normalised to between 0 and 1, along with min and max value.
uncache_part_array(part)	Removes the cached copy of the array of data for the named property part.
remove_cached_imported_arrays()	Removes any cached arrays that are mentioned in imported list.

continues on next page

Table 3 – continued from previous page

remove_cached_part_arrays()	Removes any cached arrays for parts of the collection.
remove_all_cached_arrays()	Removes any cached arrays for parts or mentioned in imported list.
write_hdf5_for_part(part[, file_name, mode, ...])	Create or append to an hdf5 file, writing dataset for the specified part.
create_xml(ext_uuid, property_array, title, ...)	Create a property xml node for a single property related to a given supporting representation node.
create_property_set_xml(title[, ps_uuid, ...])	Creates an xml node for a property set to represent this collection of properties.
basic_static_property_parts_dict([...])	Same as basic_static_property_parts() method but returning a dictionary with 5 items.
basic_static_property_uuids_dict([...])	Same as basic_static_property_uuids() method but returning a dictionary with 5 items.

__init__(*support=None*, *property_set_root=None*, *realization=None*)

Initialise an empty Property Collection, optionally populate properties from a supporting representation.

Parameters

- **support** (*optional*) – a grid.Grid object, or a well.BlockedWell, or a well.WellboreFrame object which belongs to a resqpy.Model which includes associated properties; if this argument is given, and property_set_root is None, the properties in the support's parent model which are for this representation (ie. have this object as the supporting representation) are added to this collection as part of the initialisation
- **property_set_root** (*optional*) – if present, the collection is populated with the properties defined in the xml tree of the property set
- **realization** (*integer, optional*) – if present, the single realisation (within an ensemble) that this collection is for; if None, then the collection is either covering a whole ensemble (individual properties can each be flagged with a realisation number), or is for properties that do not have multiple realizations

Note: at present, if the collection is being initialised from a property set, the support argument must also be specified; also for now, if not initialising from a property set, all properties related to the support are included, whether the relationship is supporting representation or some other relationship; the full handling of RESQML property sets and property series is still under development

add_cached_array_to_imported_list(*cached_array*, *source_info*, *keyword*, *discrete=False*, *uom=None*, *time_index=None*, *null_value=None*, *property_kind=None*, *local_property_kind_uuid=None*, *facet_type=None*, *facet=None*, *realization=None*, *indexable_element=None*, *count=1*, *const_value=None*, *points=False*, *time_series_uuid=None*, *string_lookup_uuid=None*)

Caches array and adds to the list of imported properties (but not to the collection dict).

Parameters

- **cached_array** – a numpy array to be added to the imported list for this collection (prior to being added as a part); for a constant array set cached_array to None (and use const_value)

- **source_info** (*string*) – typically the name of a file from which the array has been read but can be any information regarding the source of the data
- **keyword** (*string*) – this will be used as the citation title when a part is generated for the array
- **discrete** (*boolean, optional, default False*) – if True, the array should contain integer (or boolean) data; if False, float
- **uom** (*string, optional, default None*) – the resqml units of measure for the data
- **time_index** (*integer, optional, default None*) – if not None, the time index to be used when creating a part for the array
- **null_value** (*int or float, optional, default None*) – if present, this is used in the metadata to indicate that this value is to be interpreted as a null value wherever it appears in the data
- **property_kind** (*string*) – resqml property kind, or None
- **local_property_kind_uuid** (*uuid.UUID or string*) – uuid of local property kind, or None
- **facet_type** (*string*) – resqml facet type, or None
- **facet** (*string*) – resqml facet, or None
- **realization** (*int*) – realization number, or None
- **indexable_element** (*string, optional*) – the indexable element in the supporting representation
- **count** (*int, default 1*) – the number of values per indexable element; if greater than one then this must be the fastest cycling axis in the cached array, ie last index
- **const_value** (*int, float or bool, optional*) – the value with which a constant array is filled; required if cached_array is None, must be None otherwise
- **points** (*bool, default False*) – if True, this is a points property with an extra dimension of extent 3
- **time_series_uuid** (*UUID, optional*) – should be provided if time_index is not None, though can alternatively be provided when writing hdf5 and creating xml for the imported list
- **string_lookup_uuid** (*UUID, optional*) – should be provided for categorical properties, though can alternatively be specified when creating xml

Returns

uuid of nascent property object

Note: the process of importing property arrays follows these steps: 0. create any time series, string lookup and local property kinds that will be needed; 1. read (or generate) array of data into a numpy array in memory (cache); 2. add to the imported list using this method, or add_similar_to_imported_list(); when a batch of arrays have been added to the imported list: 3. write imported list arrays to hdf5 file using write_hdf5_for_imported_list(); 4. create xml for the new properties using create_xml_for_imported_list_and_add_parts_to_model(); step 4 also adds the new properties to the collection and to the model, and clears the imported list; after step 4, the whole sequence may be repeated for further new properties;

---

add_part_to_dict(*part*, *continuous=None*, *realization=None*, *trust_uom=True*)

Add the named part to the dictionary for this collection.

Parameters

- **part** (*string*) – the name of a part (which exists in the support's parent model) to be added to this collection
- **continuous** (*boolean, optional*) – whether the property is of a continuous (real) kind; if not None, is checked against the property's type and an assertion error is raised if there is a mismatch; should be None or True for Points properties
- **realization** (*integer, optional*) – if present, must match this collection's realization number if that is not None; if this argument is None then the part is assigned the realization number associated with this collection as a whole; if the xml for the part includes a realization index then that overrides these other sources to become the realization number
- **trust_uom** (*boolean, default True*) – if True, the uom stored in the part's xml is used as the part's uom in this collection; if False and the uom in the xml is an empty string or 'Euc', then the part's uom in this collection is set to a guess based on the property kind and min & max values; note that this guessed value is not used to overwrite the value in the xml

add_parts_list_to_dict(*parts_list*)

Add all the parts named in the parts list to the dictionary for this collection.

argument:
parts_list: a list of strings, each being the name of a part in the support's parent model

---

Note: the add_part_to_dict() function is called for each part in the list

---

add_similar_to_imported_list(*similar_uuid*, *cached_array*, *source_info=None*, *keyword=None*, *discrete=None*, *uom=None*, *time_index=None*, *null_value=None*, *property_kind=None*, *local_property_kind_uuid=None*, *facet_type=None*, *facet=None*, *realization=None*, *indexable_element=None*, *count=None*, *const_value=None*, *points=None*, *time_series_uuid=None*, *string_lookup_uuid=None*, *similar_model=None*, *title=None*)

Caches array and adds to the list of imported properties using default metadata from a similar property.

Parameters

- **similar_uuid** (*UUID*) – the uuid of a similar property from which any unspecified arguments will be fetched
- **cached_array** – a numpy array to be added to the imported list for this collection (prior to being added as a part); for a constant array set cached_array to None (and use const_value); the array is never inherited from the similar property
- **source_info** (*string, optional*) – typically the name of a file from which the array has been read but can be any information regarding the source of the data
- **keyword** (*string, optional*) – this will be used as the citation title when a part is generated for the array
- **discrete** (*boolean, optional*) – True for discrete and categorical properties, False for continuous data
- **uom** (*string, optional*) – the resqml units of measure for the data

- **time_index** (*integer, optional*) – the time index to be used when creating a part for the array
- **null_value** (*int or float, optional*) – the null value in the case of a discrete property
- **property_kind** (*string, optional*) – resqml property kind; may be a local property kind
- **local_property_kind_uuid** (*uuid.UUID or string, optional*) – uuid of local property kind
- **facet_type** (*string, optional*) – resqml facet type
- **facet** (*string, optional*) – resqml facet value
- **realization** (*int, optional*) – realization number
- **indexable_element** (*string, optional*) – the indexable element in the supporting representation
- **count** (*int, optional*) – the number of values per indexable element; if greater than one then this must be the fastest cycling axis in the cached array, ie last index
- **const_value** (*int, float or bool, optional*) – the value with which a constant array is filled; required if cached_array is None, must be None otherwise; this value is never inherited from similar
- **points** (*bool, optional*) – if True, this is a points property with an extra dimension of extent 3
- **time_series_uuid** (*UUID, optional*) – should be provided if time_index is not None, though can alternatively be provided when writing hdf5 and creating xml for the imported list
- **string_lookup_uuid** (*UUID, optional*) – should be provided for categorical properties, though can alternatively be specified when creating xml
- **similar_model** (*Model, optional*) – the model where the similar property resides, if not the same as this property collection
- **title** (*str, optional*) – synonym for keyword argument

Returns

uuid of nascent property object

Notes

this is a convenience method that avoids having to specify all the metadata when adding a property that is similar to an existing one; passing a string value of 'none' forces the argument to None; note that the cached_array and const_value arguments are never inherited from the similar property; the only extra metadata item that may be inherited is 'source'; the process of importing property arrays follows these steps: 0. create any time series, string lookup and local property kinds that will be needed; 1. read (or generate) array of data into a numpy array in memory (cache); 2. add to the imported list using this method, or add_cached_array_to_imported_list(); steps 1 & 2 may be repeated; when a batch of arrays have been added to the imported list: 3. write imported list arrays to hdf5 file using write_hdf5_for_imported_list(); 4. create xml for the new properties using create_xml_for_imported_list_and_add_parts_to_model(); step 4 also adds the new properties to the collection and to the model, and clears the imported list; after step 4, the whole sequence may be repeated for further new properties;

add_to_imported_list_sampling_other_collection(*other*, *flattened_indices*)

Makes cut down copies of parts from other collection, using indices, and adds to imported list.

Parameters

- **other** (PropertyCollection) – the source collection whose arrays will be sampled
- **flattened_indices** (*1D numpy int array*) – the indices (in flattened space) of the elements to be copied

Notes

the values in flattened_indices refer to the source (other) array elements, after flattening; the size of flattened_indices must match the size of the target (self) supporting shape; where different indexable elements are at play, with different implicit sizes, make selective copies of other and call this method once for each group of differently sized properties; for very large collections it might also be necessary to divide the work into smaller groups to reduce memory usage; this method does not write to hdf5 nor create xml – use the usual methods for further processing of the imported list

all_continuous()

Returns True if all the parts are for continuous (real) properties (includes points).

all_count_one()

Returns True if the low level Count value is 1 for all the parts in the collection.

all_discrete()

Returns True if all the parts are for discrete or categorical (integer) properties.

assign_realization_numbers()

Assigns a distinct realization number to each property, after checking for compatibility.

Note: this method does not modify realization information in any established xml; it is intended primarily as a convenience to allow realization based processing of any collection of compatible properties

---

basic_static_property_parts(*realization=None*, *share_perm_parts=False*, *perm_k_mode=None*, *perm_k_ratio=1.0*)

Returns five parts: net to gross ratio, porosity, permeability rock I, J & K; each returned part may be None.

Parameters

- **realization** – (int, optional): if present, only properties with the given realization are considered; if None, all properties in the collection are considered
- **share_perm_parts** (*boolean, default False*) – if True, the permeability I part will also be returned for J and/or K if no other properties are found for those directions; if False, None will be returned for such parts
- **perm_k_mode** (*string, optional*) – if present, indicates what action to take when no K direction permeability is found; valid values are: 'none': same as None, perm K part return value will be None 'shared': if share_perm_parts is True, then perm I value will be used for perm K, else same as None 'ratio': multiply IJ permeability by the perm_k_ratio argument 'ntg': multiply IJ permeability by ntg and by perm_k_ratio 'ntg squared': multiply IJ permeability by square of ntg and by perm_k_ratio
- **perm_k_ratio** (*float, default 1.0*) – a Kv:Kh ratio, typically in the range zero to one, applied if generating a K permeability array (perm_k_mode is 'ratio', 'ntg' or 'ntg squared' and no existing K permeability found); ignored otherwise

Returns

tuple of 5 strings being part names for – net to gross ratio, porosity, perm i, perm j, perm k (respectively); any of the returned elements may be None if no appropriate property was identified

---

Note: if generating a K permeability array, the data is appended to the hdf5 file and the xml is created, however the epc re-write must be carried out by the calling code

---

basic_static_property_parts_dict(*realization=None*, *share_perm_parts=False*, *perm_k_mode=None*, *perm_k_ratio=1.0*)

Same as basic_static_property_parts() method but returning a dictionary with 5 items.

Note: returned dictionary contains following keys: 'NTG', 'PORO', 'PERMI', 'PERMJ', 'PERMK'

---

basic_static_property_uuids(*realization=None*, *share_perm_parts=False*, *perm_k_mode=None*, *perm_k_ratio=1.0*)

Returns five uuids: net to gross ratio, porosity, permeability rock I, J & K; each returned uuid may be None.

Note: see basic_static_property_parts() method for argument documentation

---

basic_static_property_uuids_dict(*realization=None*, *share_perm_parts=False*, *perm_k_mode=None*, *perm_k_ratio=1.0*)

Same as basic_static_property_uuids() method but returning a dictionary with 5 items.

Note: returned dictionary contains following keys: 'NTG', 'PORO', 'PERMI', 'PERMJ', 'PERMK'

---

cached_part_array_ref(*part*, *dtype=None*, *masked=False*, *exclude_null=False*, *use_pack=True*)

Returns a numpy array containing the data for the property part; the array is cached in this collection.

Parameters

- **part** (*string*) – the part name for which the array reference is required
- **dtype** (*optional, default None*) – the element data type of the array to be accessed, eg 'float' or 'int'; if None (recommended), the dtype of the returned numpy array matches that in the hdf5 dataset
- **masked** (*boolean, default False*) – if True, a masked array is returned instead of a simple numpy array; the mask is set to the inactive array attribute of the support object if present
- **exclude_null** (*boolean, default False*) – if True, and masked is also True, then elements of the array holding the null value will also be masked out
- **use_pack** (*boolean, default True*) – if True, and the property is a boolean array, the hdf5 data will be unpacked if its shape indicates that it has been packed into bits for storage

Returns
reference to a cached numpy array containing the actual property data; multiple calls will return the same cached array so calling code should copy if duplication is needed

Notes

this function is the usual way to get at the actual property array; at present, the funtion only works if the entire array is stored as a single patch in the hdf5 file (resqml allows multiple patches per array); the masked functionality can be used to apply a common mask, stored in the supporting representation object with the attribute name 'inactive', to multiple properties (this will only work if the indexable element is set to the typical value for the class of supporting representation, eg. 'cells' for grid objects); if exclude_null is set True then null value elements will also be masked out (as long as masked is True); however, it is recommended simply to use np.NaN values in floating point property arrays if the commonality is not needed; set use_pack True if the hdf5 data may have been written with a similar setting

`citation_title_for_part`(*part*)

Returns the citation title for the property part.

Parameters
part (*string*) – the part name for which the citation title is required

Returns
citation title (string) for this part

Note: for simulation grid properties, the citation title is often a property keyword specific to a simulator

---

`combobulated_face_array`(*resqml_a*)

Returns a logically ordered copy of RESQML faces-per-cell property array resqml_a.

argument:
resqml_a (numpy array of shape (..., 6): a RESQML property array with indexable element faces per cell

Returns

numpy array of shape (..., 3, 2) where the 3 covers K,J,I and the 2 the -/+ face polarities being a resqpy logically
arranged copy of resqml_a

Notes

this method is for properties of IJK grids only; RESQML documentation is not entirely clear about the required ordering of -I, +I, -J, +J faces; current implementation assumes count = 1 for the property; does not currently support points properties

`constant_value_for_part`(*part*)

Returns the value (float or int) of a constant array part, or None for an hdf5 array.

Note: a constant array can optionally be expanded and written to the hdf5, in which case it will not have a constant value assigned when the dataset is read from file

continuous_for_part(*part*)

Returns True if the property is continuous (including points); False if it is discrete (or categorical).

Parameters

part (*string*) – the part name for which the continuous versus discrete flag is required

Returns

True if the part is representing a continuous (or points) property, ie. the array elements are real numbers (float); False if the part is representing a discrete property or a categorical property, ie the array elements are integers (or boolean)

Note: RESQML differentiates between discrete and categorical properties; discrete properties are unbounded integers where the values have numerical significance (eg. could be added together), whilst categorical properties have an associated dictionary mapping from a finite set of integer key values onto strings (eg. {1: 'background', 2: 'channel sand', 3: 'mud drape'}); however, this module treats categorical properties as a special case of discrete properties

count_for_part(*part*)

Returns the Count value for the property part; usually 1.

Parameters

part (*string*) – the part name for which the count is required

Returns

integer reflecting the count attribute for the part (usually one); if greater than one, the array has an extra axis, cycling fastest, having this extent

Note: this mechanism allows a vector of values to be associated with a single indexable element in the supporting representation

create_property_set_xml(*title*, *ps_uuid=None*, *originator=None*, *add_as_part=True*, *add_relationships=True*)

Creates an xml node for a property set to represent this collection of properties.

Parameters

- **title** (*string*) – to be used as citation title
- **ps_uuid** (*string, optional*) – if present, used as the uuid for the property set, otherwise a new uuid is generated
- **originator** (*string, optional*) – if present, used as the citation creator (otherwise login name is used)
- **add_as_part** (*boolean, default True*) – if True, the property set is added to the model as a part

- **add_relationships** (*boolean, default True*) – if True, the relationships to the member properties are added

Note: xml for individual properties should exist before calling this method

create_xml(*ext_uuid*, *property_array*, *title*, *property_kind*, *support_uuid=None*, *p_uuid=None*, *facet_type=None*, *facet=None*, *discrete=False*, *time_series_uuid=None*, *time_index=None*, *uom=None*, *null_value=None*, *originator=None*, *source=None*, *add_as_part=True*, *add_relationships=True*, *add_min_max=True*, *min_value=None*, *max_value=None*, *realization=None*, *string_lookup_uuid=None*, *property_kind_uuid=None*, *find_local_property_kinds=True*, *indexable_element=None*, *count=1*, *points=False*, *extra_metadata={}*, *const_value=None*, *expand_const_arrays=False*)

Create a property xml node for a single property related to a given supporting representation node.

Parameters

- **ext_uuid** (*uuid.UUID*) – the uuid of the hdf5 external part
- **property_array** (*numpy array*) – the actual property array (used to populate xml min & max values); may be None if min_value and max_value are passed or add_min_max is False
- **title** (*string*) – used for the citation Title text for the property; often set to a simulator keyword for grid properties
- **property_kind** (*string*) – the resqml property kind of the property; in the case of a bespoke (local) property kind, this is used as the title in the local property kind reference and the property_kind_uuid argument must also be passed or find_local_property_kinds set True
- **support_uuid** (*uuid.UUID, optional*) – if None, the support for the collection is used
- **p_uuid** (*uuid.UUID, optional*) – if None, a new uuid is generated for the property; otherwise this uuid is used
- **facet_type** (*string, optional*) – if present, a resqml facet type whose value is supplied in the facet argument
- **facet** (*string, optional*) – required if facet_type is supplied; the value of the facet
- **discrete** (*boolean, default False*) – if True, a discrete or categorical property node is created (depending on whether string_lookup_uuid is None or present); if False (default), a continuous property node is created
- **time_series_uuid** (*uuid.UUID, optional*) – if present, the uuid of the time series that this (recurrent) property relates to
- **time_index** (*int, optional*) – if time_series_uuid is not None, this argument is required and provides the time index into the time series for this property array
- **uom** (*string*) – the resqml unit of measure for the property (only used for continuous properties)
- **null_value** (*optional*) – the value that is to be interpreted as null if it appears in the property array
- **originator** (*string, optional*) – the name of the human being who created the property object; default is to use the login name
- **source** (*string, optional*) – if present, an extra metadata node is added as a child to the property node, with this string indicating the source of the property data

- **add_as_part** (*boolean, default True*) – if True, the newly created xml node is added as a part in the model
- **add_relationships** (*boolean, default True*) – if True, relationship xml parts are created relating the new property part to: the support, the hdf5 external part; and the time series part (if applicable)
- **add_min_max** (*boolean, default True*) – if True, min and max values are included as children in the property node
- **min_value** (*optional*) – if present and add_min_max is True, this is used as the minimum value (otherwise it is calculated from the property array)
- **max_value** (*optional*) – if present and add_min_max is True, this is used as the maximum value (otherwise it is calculated from the property array)
- **realization** (*int, optional*) – if present, is used as the realization number in the property node; if None, no realization child is created
- **string_lookup_uuid** (*optional*) – if present, and discrete is True, a categorical property node is created which refers to this string table lookup
- **property_kind_uuid** (*optional*) – if present, the property kind is a local property kind; must be None for a standard property kind
- **find_local_property_kinds** (*boolean, default True*) – if True and property_kind is not in standard supported property kind list and property_kind_uuid is None, the citation titles of PropertyKind objects in the model are compared with property_kind and if a match is found, that local property kind is used; if no match is found, a new local property kind is created; the same logic is applied if the specified property kind is abstract ('continuous', 'discrete', 'categorical') in which case the property title is also used as the property kind title
- **indexable_element** (*string, optional*) – if present, is used as the indexable element in the property node; if None, 'cells' are used for grid properties and 'nodes' for wellbore frame properties
- **count** (*int, default 1*) – the number of values per indexable element; if greater than one then this axis must cycle fastest in the array, ie. be the last index
- **points** (*bool, default False*) – if True, this is a points property
- **extra_metadata** (*dictionary, optional*) – if present, adds extra metadata in the xml
- **const_value** (*float or int, optional*) – if present, create xml for a constant array filled with this value
- **expand_const_arrays** (*boolean, default False*) – if True, the hdf5 write must also have been called with the same argument and the xml will treat a constant array as a normal array

Returns

the newly created property xml node

Notes

this function doesn't write the actual array data to the hdf5 file: that should be done before calling this function; this code (and elsewhere) only supports at most one facet per property, though the RESQML standard allows for multiple facets; RESQML does not allow facets for points properties; if the xml has not been created for the support object, then xml will not be created for relationships between the properties and the supporting representation

create_xml_for_imported_list_and_add_parts_to_model(*ext_uuid=None*, *support_uuid=None*, *time_series_uuid=None*, *selected_time_indices_list=None*, *string_lookup_uuid=None*, *property_kind_uuid=None*, *find_local_property_kinds=True*, *expand_const_arrays=False*, *extra_metadata={}*)

Add imported or generated grid property arrays as parts in parent model, creating xml.

hdf5 should already have been written.

Parameters

- **ext_uuid** – uuid for the hdf5 external part, which must be known to the model's hdf5 dictionary
- **support_uuid** (*optional*) – the uuid of the supporting representation that the imported properties relate to
- **time_series_uuid** (*optional*) – the uuid of the full or reduced time series for which any recurrent properties' timestep numbers can be used as a time index; in the case of a reduced series, the selected_time_indices_list argument must be passed and the properties timestep numbers are found in the list with the position yielding the time index for the reduced list; time_series_uuid should be present if there are any recurrent properties in the imported list, unless it has been specified when adding the array to the imported list
- **selected_time_indices_list** (*list of int, optional*) – if time_series_uuid is for a reduced time series then this argument must be present and its length must match the number of timestamps in the reduced series; the values in the list are indices in the full time series
- **string_lookup_uuid** (*optional*) – if present, the uuid of the string table lookup which any non-continuous properties relate to (ie. they are all taken to be categorical); this info can altarnatively be supplied on an individual property basis when adding the array to the imported list
- **property_kind_uuid** (*optional*) – if present, the uuid of the bespoke (local) property kind for all the property arrays in the imported list (except those with an individual local property kind uuid); this info can altarnatively be supplied on an individual property basis when adding the array to the imported list, or left as None if the following argument is True
- **find_local_property_kinds** (*boolean, default True*) – if True, local property kind uuids need not be provided as long as the property kinds are set to match the titles of the appropriate local property kind objects
- **expand_const_arrays** (*boolean, default False*) – if True, the hdf5 write must also have been called with the same argument and the xml will treat the constant arrays as normal arrays
- **extra_metadata** (*optional*) – if present, a dictionary of extra metadata to be added to each of the properties

Returns
list of uuid.UUID, being the uuids of the newly added property parts

Notes

the imported list should have been built up, and associated hdf5 arrays written, before calling this method; the imported list is cleared as a deliberate side-effect of this method (so a new set of imports can be started hereafter); if importing properties of a bespoke (local) property kind, ensure the property kind objects exist as parts in the model before calling this method

`discombobulated_face_array`(*resqpy_a*)

Return logical face property array a, re-ordered and reshaped regarding the six facial directions.

argument:

resqpy_a (numpy array of shape (..., 3, 2)): the penultimate array axis represents K,J,I and the final axis is -/+ face
polarity; the resqpy logically arranged property array to be converted to illogical RESQML ordering and shape

Returns

numpy array of shape (..., 6) being a copy of resqpy_a with slices reordered before collapsing the last 2 axes into 1;
ready to be stored as a RESQML property array with indexable element faces per cell

Notes

this method is for properties of IJK grids only; RESQML documentation is not entirely clear about the required ordering of -I, +I, -J, +J faces; current implementation assumes count = 1 for the property; does not currently support points properties

`establish_has_multiple_realizations`()

Re-evaluates the has multiple realizations attribute.

Based on whether properties belong to more than one realization.

`establish_has_single_indexable_element`()

Re-evaluate the has single indexable element attribute.

Depends on whether all properties have the same.

`establish_has_single_property_kind`()

Re-evaluates the has single property kind attribute.

Depends on whether all properties are of the same kind.

`establish_has_single_uom`()

Re-evaluates the has single uom attribute depending on whether all properties have the same units of measure.

`establish_time_set_kind`()

Re-evaulates the time set kind attribute.

Based on all properties having same time index in the same time series.

extra_metadata_for_part(*part*)

Returns the extra_metadata dictionary for the part.

Parameters
part (*string*) – the part name for which the xml node is required

Returns
dictionary containing extra_metadata for part

facet_for_part(*part*)

If relevant, returns the resqml Facet Value for the property part, eg. 'I'; otherwise None.

Parameters
part (*string*) – the part name for which the facet value is required

Returns
facet value for this part (string), for the facet type returned by the facet_type_for_part() function, or None

see notes for facet_type_for_part()

facet_list(*sort_list=True*)

Returns a list of unique facet values found amongst the parts of the collection.

facet_type_for_part(*part*)

If relevant, returns the resqml Facet Facet for the property part, eg. 'direction'; otherwise None.

Parameters
part (*string*) – the part name for which the facet type is required

Returns
standard resqml facet type for this part (string), or None

Notes

resqml refers to Facet Facet and Facet Value; the equivalents in this module are facet_type and facet; the resqml standard allows a property to have any number of facets; this module currently limits a property to having at most one facet; the facet_type and facet should be either both None or both not None

facet_type_list(*sort_list=True*)

Returns a list of unique facet types found amongst the parts of the collection.

facets_array_ref(*use_32_bit=False*, *indexable_element=None*, *use_pack=True*)

Returns a +1D array of all parts with first axis being over facet values; Use facet_list() for lookup.

Parameters

- **use_32_bit** (*boolean, default False*) – if True, the resulting numpy array will use a 32 bit dtype; if False, 64 bit
- **indexable_element** (*string, optional*) – the indexable element for the properties in the collection; if None, will be determined from the data
- **use_pack** (*boolean, default True*) – if True, and the property is a boolean array, the hdf5 data will be unpacked if its shape indicates that it has been packed into bits

Returns
numpy array containing all the data in the collection, the first axis being over facet values and the rest of the axes matching the shape of the individual property arrays

Notes

the property collection should be constructed so as to hold a suitably coherent set of properties before calling this method; the facet_list() method will return the facet values that correspond to slices in the first axis of the resulting array

`grid_for_part`(*part*)

Returns grid object that the property relates to.

Parameters
part (*string*) – the part name for which the related grid object is required

Returns
grid.Grid object reference

Note: this method maintained for backward compatibility and kept in base PropertyClass for pragmatic reasons (rather than being method in GridPropertyCollection)

`h5_key_pair_for_part`(*part*)

Return hdf5 key pair (ext uuid, internal path) for the part.

`h5_overwrite_slice`(*part*, *slice_tuple*, *array_slice*, *update_cache=True*)

Overwrites a subset of the array for part, in the hdf5 file.

Parameters

- **part** (*string*) – the part name for which the array slice is to be overwritten
- **slice_tuple** (*tuple of slice objects*) – each element should be constructed using the python built-in function slice()
- **array_slice** (*numpy array of shape to match slice_tuple*) – the data to be written
- **update_cache** (*boolean, default True*) – if True and the part is currently cached within this PropertyCollection, then the cached array is also updated; if False, the part is uncached

Notes

this method naively writes the slice to hdf5 without using mpi to look after parallel writes; if a cached copy of the array is updated, this is in an unmasked form; if calling code has a reterence to a masked version of the array then the mask will not be updated by this method; if the part is not currently cached, this method will not cause it to become cached, regardless of the update_cache argument

`h5_slice`(*part*, *slice_tuple*)

Returns a subset of the array for part, without loading the whole array.

Parameters

- **part** (*string*) – the part name for which the array slice is required

- **slice_tuple** (*tuple of slice objects*) – each element should be constructed using the python built-in function slice()

Returns
numpy array that is a hyper-slice of the hdf5 array, with the same ndim as the source hdf5 array

---

Note: this method always fetches from the hdf5 file and does not attempt local caching; the whole array is not loaded; all axes continue to exist in the returned array, even where the sliced extent of an axis is 1

---

has_multiple_realizations()

Returns the has multiple realizations flag based on whether properties belong to more than one realization.

has_single_indexable_element()

Returns the has single indexable element flag depending on whether all properties have the same.

has_single_property_kind()

Return the has single property kind flag depending on whether all properties are of the same kind.

has_single_uom()

Returns the has single uom flag depending on whether all properties have the same units of measure.

indexable_for_part(*part*)

Returns the text of the IndexableElement for the property part; usually 'cells' for grid properties.

Parameters
part (*string*) – the part name for which the indexable element is required

Returns
string, usually 'cells' when the supporting representation is a grid or 'nodes' when a wellbore frame

---

Note: see tail of Representations.xsd for overview of indexable elements usable for other object classes

---

inherit_imported_list_from_other_collection(*other*, *copy_cached_arrays=True*, *exclude_inactive=False*)

Extends this collection's imported list with items from other's imported list.

Parameters

- **other** – another PropertyCollection object with some imported arrays
- **copy_cached_arrays** (*boolean, default True*) – if True, arrays cached with the other collection are copied and cached with this collection
- **exclude_inactive** (*boolean, default False*) – if True, any item in the other imported list which has INACTIVE or ACTIVE as the keyword is excluded from the inheritance

Note: the imported list is a list of cached imported arrays with basic metadata for each array; it is used as a staging post before fully incorporating the imported arrays as parts of the support's parent model and writing the arrays to the hdf5 file

inherit_parts_from_other_collection(*other*, *ignore_clashes=False*)

Adds all the parts in the other PropertyCollection to this one.

Parameters

- **other** – another PropertyCollection object related to the same support as this collection
- **ignore_clashes** (*boolean, default False*) – if False, any part in other which is already in this collection will result in an assertion error; if True, such duplicates are simply skipped without modifying the existing part in this collection

inherit_parts_selectively_from_other_collection(*other*, *realization=None*, *support_uuid=None*, *grid=None*, *uuid=None*, *continuous=None*, *count=None*, *points=None*, *indexable=None*, *property_kind=None*, *facet_type=None*, *facet=None*, *citation_title=None*, *citation_title_match_mode=False*, *time_series_uuid=None*, *time_index=None*, *uom=None*, *string_lookup_uuid=None*, *categorical=None*, *related_uuid=None*, *const_value=None*, *extra=None*, *ignore_clashes=False*)

Adds those parts from the other PropertyCollection which match all arguments that are not None.

Parameters

- **other** – another PropertyCollection object related to the same support as this collection
- **citation_title_match_mode** (*str, optional*) – if present, one of 'is', 'starts', 'ends', 'contains', 'is not', 'does not start', 'does not end', 'does not contain'; None is the same as 'is'
- **ignore_clashes** (*boolean, default False*) – if False, any part in other which passes the filters yet is already in this collection will result in an assertion error; if True, such duplicates are simply skipped without modifying the existing part in this collection

Other optional arguments (realization, grid, uuid, continuous, count, points, indexable, property_kind, facet_type, facet, citation_title, time_series_uuid, time_index, uom, string_lookup_uuid, categorical):

For each of these arguments: if None, then all members of collection pass this filter; if not None then only those members with the given value pass this filter; finally, the filters for all the attributes must be passed for a given member (part) to be inherited; a soft relationship is sufficient for related_uuid to pass.

Note: the grid argument is maintained for backward compatibility; it is treated synonymously with support which takes precendence; the categorical boolean argument can be used to filter only Categorical (or non-Categorical) properties

inherit_similar_parts_for_facets_from_other_collection(*other*, *example_part*, *citation_title_match_mode=None*, *ignore_clashes=False*)

Adds the example part from other collection and any other parts for same property with different facets.

Parameters

- **other** – another PropertyCollection object related to the same grid as this collection, from which to inherit
- **example_part** (*string*) – the part name of an example member of other
- **citation_title_match_mode** (*str, optional*) – if present, one of 'is', 'starts', 'ends', 'contains', 'is not', 'does not start', 'does not end', 'does not contain'; None is the same as 'is'
- **ignore_clashes** (*boolean, default False*) – if False, any part in other which passes the filters yet is already in this collection will result in an assertion error; if True, such duplicates are simply skipped without modifying the existing part in this collection

---

Note: at present, the citation title must match (as well as the other identifying elements) for a part to be inherited

---

inherit_similar_parts_for_realizations_from_other_collection(*other*, *example_part*, *citation_title_match_mode=None*, *ignore_clashes=False*)

Add the example part from other collection and any other parts for same property with different realizations.

Parameters

- **other** – another PropertyCollection object related to the same support as this collection, from which to inherit
- **example_part** (*string*) – the part name of an example member of other
- **citation_title_match_mode** (*str, optional*) – if present, one of 'is', 'starts', 'ends', 'contains', 'is not', 'does not start', 'does not end', 'does not contain'; None is the same as 'is'
- **ignore_clashes** (*boolean, default False*) – if False, any part in other which passes the filters yet is already in this collection will result in an assertion error; if True, such duplicates are simply skipped without modifying the existing part in this collection

---

Note: at present, the citation title must match (as well as the other identifying elements) for a part to be inherited

---

inherit_similar_parts_for_time_series_from_other_collection(*other*, *example_part*, *citation_title_match_mode=None*, *ignore_clashes=False*)

Adds the example part from other collection and any other parts for the same property at different times.

Parameters

- **other** – another PropertyCollection object related to the same grid as this collection, from which to inherit
- **example_part** (*string*) – the part name of an example member of other (which has an associated time_series)
- **citation_title_match_mode** (*str, optional*) – if present, one of 'is', 'starts', 'ends', 'contains', 'is not', 'does not start', 'does not end', 'does not contain'; None is the same as 'is'

- **ignore_clashes** (*boolean, default False*) – if False, any part in other which passes the filters yet is already in this collection will result in an assertion error; if True, such duplicates are simply skipped without modifying the existing part in this collection

---

Note: at present, the citation title must match (as well as the other identifying elements) for a part to be inherited

---

local_property_kind_uuid(*part*)

Returns the uuid of the bespoke (local) property kind for this part, or None for a standard property kind.

masked_array(*simple_array*, *exclude_inactive=True*, *exclude_value=None*, *points=False*)

Returns a masked version of simple_array, using inactive mask associated with support for this property collection.

Parameters

- **simple_array** (*numpy array*) – an unmasked numpy array with the same shape as property arrays for the support (and indexable element) associated with this collection
- **exclude_inactive** (*boolean, default True*) – elements which are flagged as inactive in the supporting representation are masked out if this argument is True
- **exclude_value** (*float or int, optional*) – if present, elements which match this value are masked out; if not None then usually set to np.NaN for continuous data or null_value_for_part() for discrete data
- **points** (*boolean, default False*) – if True, the simple array is expected to have an extra dimension of extent 3, relative to the inactive attribute of the support

Returns

a masked version of the array, with the mask set to exclude cells which are inactive in the support

Notes

when requesting a reference to a cached copy of a property array (using other functions), a masked argument can be used to apply the inactive mask; this function is therefore rarely needed by calling code (it is used internally by this module); the simple_array need not be part of this collection

maximum_value_for_part(*part*)

Returns the maximum value for the property part, as stored in the xml.

Parameters

part (*string*) – the part name for which the maximum value is required

Returns

maximum value (as float ir int) for this part, or None if metadata item is not set

---

Note: this method merely returns the maximum value recorded in the xml for the property, it does not check the array data

---

minimum_value_for_part(*part*)

Returns the minimum value for the property part, as stored in the xml.

Parameters

part (*string*) – the part name for which the minimum value is required

Returns

minimum value (as float or int) for this part, or None if metadata item is not set

---

Note: this method merely returns the minimum value recorded in the xml for the property, it does not check the array data

---

node_for_part(*part*)

Returns the xml node for the property part.

Parameters

part (*string*) – the part name for which the xml node is required

Returns

xml Element object reference for the main xml node for the part

normalized_part_array(*part*, *masked=False*, *use_logarithm=False*, *discrete_cycle=None*, *trust_min_max=False*, *fix_zero_at=None*)

Return data normalised to between 0 and 1, along with min and max value.

Parameters

- **part** (*string*) – the part name for which the normalized array reference is required
- **masked** (*boolean, optional, default False*) – if True, the masked version of the property array is used to determine the range of values to map onto the normalized range of 0 to 1 (the mask removes inactive cells from having any impact); if False, the values of inactive cells are included in the operation; the returned normalized array is masked or not depending on this argument
- **use_logarithm** (*boolean, optional, default False*) – if False, the property values are linearly mapped to the normalized range; if True, the logarithm (base 10) of the property values are mapped to the normalized range
- **discrete_cycle** (*positive integer, optional, default None*) – if a value is supplied and the property array contains integer data (discrete or categorical), the modulus of the property values are calculated against this value before conversion to floating point and mapping to the normalized range
- **trust_min_max** (*boolean, optional, default False*) – if True, the minimum and maximum values from the property's metadata is used as the range of the property values; if False, the values are determined using numpy min and max operations
- **fix_zero_at** (*float, optional*) – if present, a value between 0.0 and 1.0 (typically 0.0 or 0.5) to pin zero at

Returns

(normalized_array, min_value, max_value) where – normalized_array is a numpy array of floats, masked or unmasked depending on the masked argument, with values ranging between 0 and 1; in the case of a masked array the values for excluded cells are meaningless and may

lie outside the range 0 to 1 min_value and max_value: the property values that have been mapped to 0 and 1 respectively

Notes

this function is typically used to map property values onto the range required for colouring in; in case of failure, (None, None, None) is returned; if use_logarithm is True, the min_value and max_value returned are the log10 values, not the original property values; also, if use logarithm is True and the minimum property value is not greater than zero, then values less than 0.0001 are set to 0.0001, prior to taking the logarithm; fix_zero_at is mutually incompatible with use_logarithm; to force the normalised data to have a true zero, set fix_zero_at to 0.0; for divergent data fixing zero at 0.5 will often be appropriate; fixing zero at 0.0 or 1.0 may result in normalised values being clipped; for floating point data, NaN values will be handled okay; if all data are NaN, (None, NaN, NaN) is returned; for integer data, null values are not currently supported (though the RESQML metadata can hold a null value); the masked argument is most applicable to properties for grid objects; note that NaN values are excluded when determining the min and max regardless of the value of the masked argument; not applicable to points properties

`null_value_for_part`(*part*)

Returns the null value for the (discrete) property part; np.NaN for continuous parts.

Parameters
part (*string*) – the part name for which the null value is required

Returns
int or np.NaN

`number_of_imports`()

Returns the number of property arrays in the imported list for this collection.

Returns
count of number of cached property arrays in the imported list for this collection (non-negative integer)

Note: the importation list is cleared after creation of xml trees for the imported properties, so this function will return zero at that point, until a new list of imports is built up

---

`number_of_parts`()

Returns the number of parts (properties) in this collection.

Returns
count of the number of parts (members) in this collection; there is one part per property array (non-negative integer)

`override_min_max`(*part*, *min_value*, *max_value*)

Sets the minimum and maximum values in the metadata for the part.

Parameters

- **part** (*string*) – the part name for which the minimum and maximum values are to be set
- **min_value** (*float or int or string*) – the minimum value to be stored in the metadata
- **max_value** (*float or int or string*) – the maximum value to be stored in the metadata

Note: this function is typically called if the existing min & max metadata is missing or distrusted; the min and max values passed in are typically the result of numpy min and max function calls (possibly skipping NaNs) on the property array or a version of it masked for inactive cells

part_filename(*part*)

Returns a string which can be used as the starting point of a filename relating to part.

Parameters
part (*string*) – the part name for which a partial filename is required

Returns
a string suitable as the basis of a filename for the part (typically used when exporting)

part_in_collection(*part*)

Returns True if named part is member of this collection; otherwise False.

Parameters
part (*string*) – part name to be tested for membership of this collection

Returns
boolean

part_is_categorical(*part*)

Returns True if the property is categorical (not conintuous and has an associated string lookup).

part_str(*part*, *include_citation_title=True*)

Returns a human-readable string identifying the part.

Parameters

- **part** (*string*) – the part name for which a displayable string is required
- **include_citation_title** (*boolean, default True*) – if True, the citation title for the part is included in parenthesis at the end of the returned string; otherwise it does not appear

Returns
a human readable string consisting of the property kind, the facet (if present), the time index (if applicable), and the citation title (if requested)

Note: the time index is labelled 'timestep' in the returned string; however, resqml differentiates between the simulator timestep number and a time index into a time series; at present this module conflates the two

parts()

Return list of parts in this collection.

Returns
list of part names (strings) being the members of this collection; there is one part per property array

patch_min_max_for_part(*part*, *minimum=None*, *maximum=None*, *model=None*)

Updates the minimum and/ox maximum values stored in the metadata, optionally updating xml tree too.

Parameters

- **part** (*str*) – the part name of the property
- **minimum** (*float or int, optional*) – the new minimum value to be set in the metadata (unchanged if None)
- **maximum** (*float or int, optional*) – the new maximum value to be set in the metadata (unchanged if None)
- **model** (model.Model, *optional*) – if present and containing xml for the part, that xml is also patched

Notes

this method is rarely needed: only if a property array is being re-populated after being initialised with temporary values; the xml tree for the part in the model will only be updated where the minimum and/or maximum nodes already exist in the tree

points_for_part(*part*)

Returns True if the property is a points property; False otherwise.

Parameters

part (*string*) – the part name for which the points flag is required

Returns

True if the part is representing a points property, ie. the array has an extra dimension of extent 3 covering the xyz axes; False if the part is representing a non-points property

populate_from_property_set(*property_set_root*)

Populates this (newly created) collection based on xml members of property set.

property_kind_for_part(*part*)

Returns the resqml property kind for the property part.

Parameters

part (*string*) – the part name for which the property kind is required

Returns

standard resqml property kind or local property kind for this part, as a string, eg. 'porosity'

Notes

see attributes of this module named supported_property_kind_list and supported_local_property_kind_list for the property kinds which this module can relate to simulator keywords (Nexus); however, other property kinds should be handled okay in a generic way; for bespoke (local) property kinds, this is the property kind title as stored in the xml reference node

property_kind_list(*sort_list=True*)

Returns a list of unique property kinds found amongst the parts of the collection.

realization_for_part(*part*)

Returns realization number (within ensemble) that the property relates to.

Parameters

part (*string*) – the part name for which the realization number (realization index) is required

Returns

integer or None

realization_list(*sort_list=True*)

Returns a list of unique realization numbers present in the collection.

realizations_array_ref(*use_32_bit=False*, *fill_missing=True*, *fill_value=None*, *indexable_element=None*)

Returns a +1D array of all parts with first axis being over realizations.

Parameters

- **use_32_bit** (*boolean, default False*) – if True, the resulting numpy array will use a 32 bit dtype; if False, 64 bit
- **fill_missing** (*boolean, default True*) – if True, the first axis of the resulting numpy array will range from 0 to the maximum realization number present and slices for any missing realizations will be filled with fill_value; if False, the extent of the first axis will only cpver the number pf realizations actually present (see also notes)
- **fill_value** (*int or float, optional*) – the value to use for missing realization slices; if None, will default to np.NaN if data is continuous, -1 otherwise; irrelevant if fill_missing is False
- **indexable_element** (*string, optional*) – the indexable element for the properties in the collection; if None, will be determined from the data

Returns

numpy array containing all the data in the collection, the first axis being over realizations and the rest of the axes matching the shape of the individual property arrays

Notes

the property collection should be constructed so as to hold a suitably coherent set of properties before calling this method; if fill_missing is False, the realization axis indices range from zero to the number of realizations present; if True, the realization axis indices range from zero to the maximum realization number and slices for missing realizations will be filled with the fill_value

remove_all_cached_arrays()

Removes any cached arrays for parts or mentioned in imported list.

remove_cached_imported_arrays()

Removes any cached arrays that are mentioned in imported list.

remove_cached_part_arrays()

Removes any cached arrays for parts of the collection.

remove_part_from_dict(*part*)

Remove the named part from the dictionary for this collection.

argument:
part (string): the name of a part which might be in this collection, to be removed

Note: if the part is not in the collection, no action is taken and no exception is raised

`remove_parts_list_from_dict`(*parts_list*)

Remove all the parts named in the parts list from the dictionary for this collection.

argument:
parts_list: a list of strings, each being the name of a part which might be in the collection

Note: the remove_part_from_dict() function is called for each part in the list

`selective_parts_list`(*realization=None*, *support_uuid=None*, *continuous=None*, *points=None*, *count=None*, *indexable=None*, *property_kind=None*, *facet_type=None*, *facet=None*, *citation_title=None*, *title_mode=None*, *time_series_uuid=None*, *time_index=None*, *uom=None*, *string_lookup_uuid=None*, *categorical=None*, *related_uuid=None*, *title=None*, *const_value=None*)

Returns a list of parts filtered by those arguments which are not None.

All arguments are optional.

For each of these arguments: if None, then all members of collection pass this filter; if not None then only those members with the given value pass this filter; finally, the filters for all the attributes must be passed for a given member (part) to be included in the returned list of parts; title is a synonym for citation_title

Returns
list of part names (strings) of those parts which match any selection arguments which are not None

`set_realization`(*realization*)

Sets the model realization number (within an ensemble) for this collection.

argument:

realization (non-negative integer): the realization number of the whole collection within an ensemble of
collections

Note: the resqml Property classes allow for a realization index to be associated with an individual property array; this module supports this by associating a realization number (equivalent to realization index) for each part (ie. for each property array); however, the collection can be given a realization number which is then applied to each member of the collection as it is added, if no part-specific realization number is provided

`set_support`(*support_uuid=None*, *support=None*, *model=None*, *modify_parts=True*)

Sets the supporting object associated with this collection if not done so at initialisation.

Does not load properties.

Parameters

- **support_uuid** – the uuid of the supporting representation which the properties in this collection are for

- **support** – a grid.Grid, unstructured.UnstructuredGrid (or derived class), well.WellboreFrame, well.BlockedWell, surface.Mesh, well.WellboreMarkerFrame or fault.GridConnectionSet object which the properties in this collection are for
- **model** (*model.Model object, optional*) – if present, the model associated with this collection is set to this; otherwise the model is assigned from the supporting object
- **modify_parts** (*boolean, default True*) – if True, any parts already in this collection have their individual support_uuid set

shape_and_type_of_part(*part*)

Returns shape tuple and element type of cached or hdf5 array for part.

single_array_ref(*realization=None*, *support_uuid=None*, *uuid=None*, *continuous=None*, *points=None*, *count=None*, *indexable=None*, *property_kind=None*, *facet_type=None*, *facet=None*, *citation_title=None*, *time_series_uuid=None*, *time_index=None*, *uom=None*, *string_lookup_uuid=None*, *categorical=None*, *dtype=None*, *masked=False*, *exclude_null=False*, *multiple_handling='exception'*, *title=None*, *title_mode=None*, *related_uuid=None*, *use_pack=True*)

Returns the array of data for a single part selected by those arguments which are not None.

Parameters

- **dtype** (*optional, default None*) – the element data type of the array to be accessed, eg 'float' or 'int'; if None (recommended), the dtype of the returned numpy array matches that in the hdf5 dataset
- **masked** (*boolean, optional, default False*) – if True, a masked array is returned instead of a simple numpy array; the mask is set to the inactive array attribute of the grid object
- **exclude_null** (*boolean, default False*) – it True and masked is True, elements holding the null value will also be masked out
- **multiple_handling** (*string, default 'exception'*) – one of 'exception', 'none', 'first', 'oldest', 'newest'
- **title** (*string, optional*) – synonym for citation_title argument
- **use_pack** (*boolean, default True*) – if True, and the property is a boolean array, the hdf5 data will be unpacked if its shape indicates that it has been packed into bits

Other optional arguments: realization, support_uuid, continuous, points, count, indexable, property_kind, facet_type, facet, citation_title, time_series_uuid, time_index, uom, string_lookup_id, categorical, related_uuid:

For each of these arguments: if None, then all members of collection pass this filter; if not None then only those members with the given value pass this filter; finally, the filters for all the attributes must pass for a given member (part) to be selected

Returns

reference to a cached numpy array containing the actual property data for the part which matches all selection arguments which are not None

Notes

returns None if no parts match; if more than one part matches multiple_handling argument determines behaviour; multiple calls will return the same cached array so calling code should copy if duplication is needed;

singleton(*realization=None*, *support_uuid=None*, *uuid=None*, *continuous=None*, *points=None*, *count=None*, *indexable=None*, *property_kind=None*, *facet_type=None*, *facet=None*, *citation_title=None*, *time_series_uuid=None*, *time_index=None*, *uom=None*, *string_lookup_uuid=None*, *categorical=None*, *multiple_handling='exception'*, *title=None*, *title_mode=None*, *related_uuid=None*, *const_value=None*)

Returns a single part selected by those arguments which are not None.

multiple_handling (string, default 'exception'): one of 'exception', 'none', 'first', 'oldest', 'newest'
title (string, optional): synonym for citation_title argument

For each argument (other than multiple_handling): if None, then all members of collection pass this filter; if not None then only those members with the given value pass this filter; finally, the filters for all the attributes must pass for a given member (part) to be selected

multiple_handling (string, default 'exception'): one of 'exception', 'none', 'first', 'oldest', 'newest'

Returns
part name (string) of the part which matches all selection arguments which are not None; returns None if no parts match; if more than one part matches multiple_handling argument determines behaviour

string_lookup_for_part(*part*)

Returns a StringLookup object for the part, if it has a string lookup uuid, otherwise None.

string_lookup_uuid_for_part(*part*)

If the property has an associated string lookup (is categorical), return the uuid.

Parameters
part (*string*) – the part name for which the string lookup uuid is required

Returns
string lookup uuid (uuid.UUID) for this part

string_lookup_uuid_list(*sort_list=True*)

Returns a list of unique string lookup uuids found amongst the parts of the collection.

support_uuid_for_part(*part*)

Returns supporting representation object's uuid that the property relates to.

Parameters
part (*string*) – the part name for which the related support object uuid is required

Returns
uuid.UUID object (or string representation thereof)

supporting_shape(*indexable_element=None*, *direction=None*)

Return the shape of the supporting representation with respect to the given indexable element

Parameters

- **indexable_element** (*string, optional*) – if None, a hard-coded default depending on the supporting representation class will be used

- **direction** (*string, optional*) – must be passed if required for the combination of support class and indexable element; currently only used for Grid faces.

Returns
list of int, being required shape of numpy array, or None if not coded for

---

Note: individual property arrays will only match this shape if they have the same indexable element and a count of one

---

time_index_for_part(*part*)

If the property has an associated time series (is not static), returns the time index within the time series.

Parameters
part (*string*) – the part name for which the time index is required

Returns
time index (integer) for this part

time_index_list(*sort_list=True*)

Returns a list of unique time indices found amongst the parts of the collection.

time_series_array_ref(*use_32_bit=False*, *fill_missing=True*, *fill_value=None*, *indexable_element=None*)

Returns a +1D array of all parts with first axis being over time indices.

Parameters

- **use_32_bit** (*boolean, default False*) – if True, the resulting numpy array will use a 32 bit dtype; if False, 64 bit
- **fill_missing** (*boolean, default True*) – if True, the first axis of the resulting numpy array will range from 0 to the maximum time index present and slices for any missing indices will be filled with fill_value; if False, the extent of the first axis will only cpver the number pf time indices actually present (see also notes)
- **fill_value** (*int or float, optional*) – the value to use for missing time index slices; if None, will default to np.NaN if data is continuous, -1 otherwise; irrelevant if fill_missing is False
- **indexable_element** (*string, optional*) – the indexable element for the properties in the collection; if None, will be determined from the data

Returns
numpy array containing all the data in the collection, the first axis being over time indices and the rest of the axes matching the shape of the individual property arrays

Notes

the property collection should be constructed so as to hold a suitably coherent set of properties before calling this method; if fill_missing is False, the time axis indices range from zero to the number of time indices present, with the list of tine index values available by calling the method time_index_list(sort_list = True); if fill_missing is True, the time axis indices range from zero to the maximum time index and slices for missing time indices will be filled with the fill_value

time_series_uuid_for_part(*part*)

If the property has an associated time series (is not static), returns the uuid for the time series.

Parameters
part (*string*) – the part name for which the time series uuid is required

Returns
time series uuid (uuid.UUID) for this part

time_series_uuid_list(*sort_list=True*)

Returns a list of unique time series uuids found amongst the parts of the collection.

time_set_kind()

Returns the time set kind attribute.

Based on all properties having same time index in the same time series.

title_for_part(*part*)

Synonymous with citation_title_for_part().

titles()

Returns a list of citation titles for the parts in the collection.

uncache_part_array(*part*)

Removes the cached copy of the array of data for the named property part.

argument:
part (string): the part name for which the cached array is to be removed

---

Note: this function applies a python delattr() which will mark the array as no longer being in use here; however, actual freeing of the memory only happens when all other references to the array are released

---

unique_indexable_element_list(*sort_list=False*)

Returns a list of unique values for the IndexableElement of the property parts in the collection.

uom_for_part(*part*)

Returns the resqml units of measure for the property part.

Parameters
part (*string*) – the part name for which the units of measure is required

Returns
resqml units of measure (string) for this part

uom_list(*sort_list=True*)

Returns a list of unique units of measure found amongst the parts of the collection.

uuid_for_part(*part*)

Returns UUID object for the property part.

Parameters
part (*string*) – the part name for which the UUID is required

Returns
uuid.UUID object reference; use str(uuid_for_part()) to convert to string

uuids()

Return list of uuids in this collection.

Returns

list of uuids being the members of this collection; there is one uuid per property array

write_hdf5_for_imported_list(*file_name=None*, *mode='a'*, *expand_const_arrays=False*, *dtype=None*, *use_int32=None*, *use_pack=False*, *chunks=None*, *compression=None*)

Create or append to an hdf5 file, writing datasets for the imported arrays.

Parameters

- **file_name** (*str, optional*) – if present, this hdf5 filename will override the default
- **mode** (*str, default 'a'*) – the mode to open the hdf5 file in, either 'a' (append), or 'w' (overwrite)
- **expand_const_arrays** (*boolean, default False*) – if True, constant arrays will be written in full to the hdf5 file and the same argument should be used when creating the xml
- **dtype** (*numpy dtype, optional*) – the required numpy element type to use when writing to hdf5; eg. np.float16, np.float32, np.float64, np.uint8, np.int16, np.int32, np.int64 etc.; defaults to the dtype of each individual numpy array in the imported list except for int64 for which the use_int32 controls whether to write as 32 bit data
- **use_int32** (*bool, optional*) – if dtype is None, this controls whether 64 bit int arrays are written as 32 bit; if None, the system default is to write as 32 bit; if True, 32 bit is used; if False, 64 bit data is written; ignored if dtype is not None
- **use_pack** (*bool, default False*) – if True, bool arrays will be packed along their last axis; this will generally result in hdf5 data that is not readable by non-resqpy applications
- **chunks** (*str, optional*) – if not None, one of 'auto', 'all', or 'slice', controlling hdf5 chunks
- **compression** (*str, optional*) – if not None, one of 'gzip' or 'lzf' being the hdf5 compression algorithm to be used; gzip gives better compression ratio but is slower

write_hdf5_for_part(*part*, *file_name=None*, *mode='a'*, *use_pack=False*, *chunks=None*, *compression=None*)

Create or append to an hdf5 file, writing dataset for the specified part.

7.10.2 resqpy.property.WellIntervalProperty

class resqpy.property.**WellIntervalProperty**(*collection*, *part*)

Bases: object

Thin wrapper class around interval properties for a Wellbore Frame or Blocked Wellbore.

ie, interval or cell well logs.

Methods:

__init__(collection, part)	Create an interval log or blocked well log from a part name.
values()	Return interval log or blocked well log as numpy array.

__init__(*collection*, *part*)

Create an interval log or blocked well log from a part name.

values()

Return interval log or blocked well log as numpy array.

7.10.3 resqpy.property.WellLog

class resqpy.property.**WellLog**(*collection*, *uuid*)

Bases: object

Thin wrapper class around RESQML properties for well logs.

Methods:

__init__(collection, uuid)	Create a well log from a part name.
values()	Return log data as numpy array.

__init__(*collection*, *uuid*)

Create a well log from a part name.

values()

Return log data as numpy array.

Note: may return 2D numpy array with shape (num_depths, num_columns).

Functions

create_transmisibility_multiplier_property_kind	Create a local property kind 'transmisibility multiplier' for a given model.
guess_uom	Returns a guess at the units of measure for the given kind of property.
infer_property_kind	Guess a valid property kind.
property_collection_for_keyword	Returns a new PropertyCollection with parts that match the property kind and facet deduced for the keyword.
property_kind_and_facet_from_keyword	If keyword is recognised, returns equivalent resqml PropertyKind and Facet info.
property_over_time_series_from_collection	Returns a new PropertyCollection with parts like the example part, over all indices in its time series.
property_part	Returns individual property part from model matching filters.
property_parts	Returns list of property parts from model matching filters.
reformat_column_edges_from_resqml_format	Converts an array of shape (nj,ni,4) in RESQML edge ordering to shape (nj,ni,2,2)
reformat_column_edges_to_resqml_format	Converts an array of shape (nj,ni,2,2) to shape (nj,ni,4) in RESQML edge ordering.
same_property_kind	Returns True if the two property kinds are the same, or pseudonyms.
selective_version_of_collection	Returns a new PropertyCollection with those parts which match all arguments that are not None.
write_hdf5_and_create_xml_for_active_property	Writes hdf5 data and creates xml for an active cell property; returns uuid.

7.10.4 resqpy.property.create_transmisibility_multiplier_property_kind

resqpy.property.**create_transmisibility_multiplier_property_kind**(*model*)

Create a local property kind 'transmisibility multiplier' for a given model.

argument:
model: resqml model object

Returns
property kind uuid

7.10.5 resqpy.property.guess_uom

resqpy.property.**guess_uom**(*property_kind*, *minimum*, *maximum*, *support*, *facet_type=None*, *facet=None*)

Returns a guess at the units of measure for the given kind of property.

Parameters

- **property_kind** (*string*) – a valid resqml property kind, lowercase
- **minimum** – the minimum value in the data for which the units are being guessed
- **maximum** – the maximum value in the data for which the units are being guessed
- **support** – the grid.Grid or well.WellboreFrame object which the property data relates to

- **facet_type** (*string, optional*) – a valid resqml facet type, lowercase, one of: 'direction', 'what', 'netgross', 'qualifier', 'conditions', 'statistics'
- **facet** – (string, present if facet_type is present): the value relating to the facet_type, eg. 'I' for direction, or 'oil' for 'what'

Returns

a valid resqml unit of measure (uom) for the property_kind, or None

Notes

this function is tailored towards Nexus unit systems; the resqml standard allows a property to have any number of facets; however, this module currently only supports zero or one facet per property

7.10.6 resqpy.property.infer_property_kind

resqpy.property.**infer_property_kind**(*name*, *unit*)

Guess a valid property kind.

7.10.7 resqpy.property.property_collection_for_keyword

resqpy.property.**property_collection_for_keyword**(*collection*, *keyword*)

Returns a new PropertyCollection with parts that match the property kind and facet deduced for the keyword.

Parameters

- **collection** – an existing PropertyCollection from which a subset will be returned as a new object; the existing collection might often be the 'main' collection holding all the properties for a supporting representation (grid or wellbore frame)
- **keyword** (*string*) – a simulator keyword for which the property kind (and facet, if any) can be deduced

Returns

a new PropertyCollection containing those memners of collection which have the property kind (and facet, if any) as that deduced for the keyword

---

Note: this function is particularly relevant to grid property collections for simulation models; the handling of simulator keywords in this module is based on the main grid property keywords for Nexus; if the resqml dataset was generated from simulator data using this module then the result of this function should be reliable; resqml data sets from other sources might use facets if a different way, leading to an omission in the results of this function

---

7.10.8 resqpy.property.property_kind_and_facet_from_keyword

resqpy.property.**property_kind_and_facet_from_keyword**(*keyword*)

If keyword is recognised, returns equivalent resqml PropertyKind and Facet info.

argument:
keyword (string): Nexus grid property keyword

Returns
(property_kind, facet_type, facet) as defined in resqml standard; some or all may be None

---

Note: this function may now return the local property kind 'transmissibility multiplier'; calling code must ensure that the local property kind object is created if not already present

---

7.10.9 resqpy.property.property_over_time_series_from_collection

resqpy.property.**property_over_time_series_from_collection**(*collection*, *example_part*)

Returns a new PropertyCollection with parts like the example part, over all indices in its time series.

Parameters

- **collection** – an existing PropertyCollection from which a subset will be returned as a new object; the existing collection might often be the 'main' collection holding all the properties for a grid
- **example_part** (*string*) – the part name of an example member of collection (which has an associated time_series)

Returns
a new PropertyCollection containing those memners of collection which have the same property kind (and facet, if any) as the example part and which have the same associated time series

7.10.10 resqpy.property.property_part

resqpy.property.**property_part**(*model*, *obj_type*, *parts_list=None*, *property_kind=None*, *facet_type=None*, *facet=None*, *related_uuid=None*)

Returns individual property part from model matching filters.

7.10.11 resqpy.property.property_parts

resqpy.property.**property_parts**(*model*, *obj_type*, *parts_list=None*, *property_kind=None*, *facet_type=None*, *facet=None*, *related_uuid=None*)

Returns list of property parts from model matching filters.

7.10.12 resqpy.property.reformat_column_edges_from_resqml_format

resqpy.property.**reformat_column_edges_from_resqml_format**(*array*)

Converts an array of shape (nj,ni,4) in RESQML edge ordering to shape (nj,ni,2,2)

7.10.13 resqpy.property.reformat_column_edges_to_resqml_format

resqpy.property.**reformat_column_edges_to_resqml_format**(*array*)

Converts an array of shape (nj,ni,2,2) to shape (nj,ni,4) in RESQML edge ordering.

7.10.14 resqpy.property.same_property_kind

resqpy.property.**same_property_kind**(*pk_a*, *pk_b*)

Returns True if the two property kinds are the same, or pseudonyms.

7.10.15 resqpy.property.selective_version_of_collection

resqpy.property.**selective_version_of_collection**(*collection*, *realization=None*, *support_uuid=None*, *uuid=None*, *continuous=None*, *points=None*, *count=None*, *indexable=None*, *property_kind=None*, *facet_type=None*, *facet=None*, *citation_title=None*, *time_series_uuid=None*, *time_index=None*, *uom=None*, *string_lookup_uuid=None*, *categorical=None*, *title=None*, *title_mode=None*, *related_uuid=None*, *const_value=None*, *extra=None*)

Returns a new PropertyCollection with those parts which match all arguments that are not None.

Parameters

- **collection** (PropertyCollection) – an existing collection from which a subset will be returned as a new object
- **realization** (*int, optional*) – realization number to filter on
- **support_uuid** (*UUID or str, optional*) – UUID of supporting representation, to filter on
- **uuid** (*UUID or str, optional*) – a property uuid to select a singleton property from the collection
- **continuous** (*bool, optional*) – if True, continuous properties are selected; if False, discrete and categorical
- **points** (*bool, optional*) – if True, points properties are selected; if False, they are excluded
- **count** (*int, optional*) – a count value to filter on
- **indexable** (*str, optional*) – indexable elements flavour to filter on
- **property_kind** (*str, optional*) – property kind to filter on (commonly used)
- **facet_type** (*str, optional*) – a facet_type to filter on (must be present for property to be selected)
- **facet** (*str, optional*) – a facet value to filter on (if used, facet_type should be specified)

- **citation_title** (*str, optional*) – citation to title to filter on; see also title_mode argument
- **time_series_uuid** (*UUID or str, optional*) – UUID of a TimeSeries to filter on
- **time_index** (*int, optional*) – a time series time index to filter on
- **uom** (*str, optional*) – unit of measure to filter on
- **string_lookup_uuid** (*UUID or str, optional*) – UUID of a string lookup table to filter on
- **categorical** (*bool, optional*) – if True, only categorical properties are selected; if False they are excluded
- **title** (*str, optional*) – synonymous with citation_title argument
- **title_mode** (*str, optional*) – if present, one of 'is', 'starts', 'ends', 'contains', 'is not', 'does not start', 'does not end', 'does not contain'; None is the same as 'is'; will default to 'is' if not specified and title or citation_title argument is present
- **related_uuid** (*UUID or str, optional*) – only properties with direct relationship to this uuid are selected
- **const_value** (*float or int, optional*) – only properties flagged as constant, with given value, are selected
- **extra** (*dict, optional*) – only properties where the extra metadata includes all the items of this dict are selected

Returns

a new PropertyCollection containing those properties which match the filter parameters that are not None

Notes

the existing collection might often be the 'main' collection holding all the properties for a supporting representation (eg. grid, blocked well or wellbore frame); for each of the filtering arguments: if None, then all members of collection pass this filter; if not None then only those members with the given value pass this filter; special values: '*' any non-None value passes; 'none' only None passes; citation_title (or its synonym title) uses string filtering in association with title_mode argument; finally, the filters for all the attributes must be passed for a given member to be included in the returned collection; title is a synonym for the citation_title argument; related_uuid will pass if any relationship exists ('hard' or 'soft'); the categorical boolean argument can be used to select only categorical (or non-categorical) properties, even though this is not explicitly held as a field in the internal dictionary

7.10.16 resqpy.property.write_hdf5_and_create_xml_for_active_property

resqpy.property.**write_hdf5_and_create_xml_for_active_property**(*model*, *active_property_array*, *support_uuid*, *title='ACTIVE'*, *realization=None*, *time_series_uuid=None*, *time_index=None*)

Writes hdf5 data and creates xml for an active cell property; returns uuid.

resqpy.property.grid_property_collection	Class for a collection of grid properties
resqpy.property.property_collection	Class handling collections of RESQML properties for grids, wellbore frames, grid connection sets etc.
resqpy.property.property_common	Module containing common methods for properties
resqpy.property.property_kind	Containing resqml propertykind class
resqpy.property.string_lookup	Class containing resqml stringlookup class.
resqpy.property.well_interval_property	Class for wellintervalproperty, for resqml wellbore frame of blocked wellbore supports.
resqpy.property.well_interval_property_collection	Class for a collection of well interval properties
resqpy.property.well_log	Class for a welllog, representing resqml properties for well logs
resqpy.property.well_log_collection	Class for a collection of well logs

7.10.17 resqpy.property.grid_property_collection

Class for a collection of grid properties

7.10.18 resqpy.property.property_collection

Class handling collections of RESQML properties for grids, wellbore frames, grid connection sets etc.

7.10.19 resqpy.property.property_common

Module containing common methods for properties

Functions

check_and_warn_property_kind	Check property kind and warn if one of three main abstract kinds.
dtype_flavour	Returns the numpy elemental data type depending on the two boolean flags.
return_cell_indices	Returns the i'th entry in the cell_indices array, or NaN if i has null value of -1.

resqpy.property.property_common.check_and_warn_property_kind

resqpy.property.property_common.**check_and_warn_property_kind**(*pk*, *activity*)

Check property kind and warn if one of three main abstract kinds.

resqpy.property.property_common.dtype_flavour

resqpy.property.property_common.**dtype_flavour**(*continuous*, *use_32_bit*)

Returns the numpy elemental data type depending on the two boolean flags.

resqpy.property.property_common.return_cell_indices

resqpy.property.property_common.**return_cell_indices**(*i*, *cell_indices*)

Returns the i'th entry in the cell_indices array, or NaN if i has null value of -1.

7.10.20 resqpy.property.property_kind

Containing resqml propertykind class

Functions

establish_zone_property_kind	Returns zone local property kind object, creating the xml and adding as part if not found in model.

resqpy.property.property_kind.establish_zone_property_kind

resqpy.property.property_kind.**establish_zone_property_kind**(*model*)

Returns zone local property kind object, creating the xml and adding as part if not found in model.

7.10.21 resqpy.property.string_lookup

Class containing resqml stringlookup class.

7.10.22 resqpy.property.well_interval_property

Class for wellintervalproperty, for resqml wellbore frame of blocked wellbore supports.

7.10.23 resqpy.property.well_interval_property_collection

Class for a collection of well interval properties

7.10.24 resqpy.property.well_log

Class for a welllog, representing resqml properties for well logs

7.10.25 resqpy.property.well_log_collection

Class for a collection of well logs

7.11 resqpy.rq_import

Miscellaneous functions for importing from other formats.

Functions

add_ab_properties	Import a list of pure binary property array files as grid properties.
add_surfaces	Process a list of surface files, adding each surface as a new part in the resqml model.
grid_from_cp	Create a resqpy.grid.Grid object from a 7D corner point array.
import_nexus	Read a simulation grid geometry and optionally grid properties.
import_vdb_all_grids	Creates a RESQML dataset containing grids and grid properties, including LGRs, for a single realisation.
import_vdb_ensemble	Adds properties from all vdb's within an ensemble directory tree to a single RESQML dataset.

7.11.1 resqpy.rq_import.add_ab_properties

resqpy.rq_import.**add_ab_properties**(*epc_file*, *grid_uuid=None*, *ext_uuid=None*, *ab_property_list=None*)

Import a list of pure binary property array files as grid properties.

Parameters

- **epc_file** (*str*) – path of existing resqml epc to be added to
- **grid_uuid** (*UUID, optional*) – the uuid of the grid to receive the properties; required if more than one grid present
- **ext_uuid** (*UUID, optional*) – the uuid of the hdf5 extension part to use for the arrays; recommended to leave as None
- **ab_property_list** (*list of tuples*) – each entry contains: (file_name, keyword, property_kind, facet_type, facet, uom, time_index, null_value, discrete, realization)

Returns

Model, with the new properties added, with hdf5 and epc fully updated

7.11.2 resqpy.rq_import.add_surfaces

resqpy.rq_import.**add_surfaces**(*epc_file*, *crs_uuid=None*, *surface_file_format='zmap'*, *rq_class='surface'*, *surface_role='map'*, *quad_triangles=False*, *surface_file_list=None*, *make_horizon_interpretations_and_features=True*)

Process a list of surface files, adding each surface as a new part in the resqml model.

Parameters

- **epc_file** (*str*) – file name and path to an existing resqml model
- **crs_uuid** (*uuid.UUID, default None*) – uuid for a coordinate reference system. Defaults to crs associated with model (usually the main grid crs)
- **surface_file_format** (*str, default 'zmap'*) – 'zmap', 'rms', 'roxar' or 'GOCAD-Tsurf'. The format of the input file
- **rq_class** (*str, default 'surface'*) – 'surface' or 'mesh'. The class of object ot be
- **surface_role** (*str, default 'map'*) – 'map' or 'pick'
- **quad_triangles** (*bool, default False*) – if True, 4 triangles per quadrangle will be used for mesh formats, otherwise 2
- **surface_file_list** (*list, default None*) – list of full file names (paths), each holding one surface
- **make_horizon_interpretations_and_features** (*bool, default True*) – if True, feature and interpretation objects are created

Returns

resqml model object with added surfaces

7.11.3 resqpy.rq_import.grid_from_cp

resqpy.rq_import.**grid_from_cp**(*model*, *cp_array*, *crs_uuid*, *active_mask=None*, *geometry_defined_everywhere=True*, *treat_as_nan=None*, *dot_tolerance=1.0*, *morse_tolerance=5.0*, *max_z_void=0.1*, *split_pillars=True*, *split_tolerance=0.01*, *ijk_handedness='right'*, *known_to_be_straight=False*)

Create a resqpy.grid.Grid object from a 7D corner point array.

Parameters

- **model** (resqpy.model.Model) – model to which the grid will be added
- **cp_array** (*numpy float array*) – 7 dimensional numpy array of nexus corner point data, in nexus ordering
- **crs_uuid** (*uuid.UUID*) – uuid for the coordinate reference system
- **active_mask** (*3d numpy bool array*) – array indicating which cells are active
- **geometry_defined_everywhere** (*bool, default True*) – if False then inactive cells are marked as not having geometry
- **treat_as_nan** (*float, default None*) – if a value is provided corner points with this value will be assigned nan
- **dot_tolerance** (*float, default 1.0*) – minimum manhatten distance of primary diagonal of cell, below which cell is treated as inactive

- **morse_tolerance** (*float, default 5.0*) – maximum ratio of i and j face vector lengths, beyond which cells are treated as inactive
- **max_z_void** (*float, default 0.1*) – maximum z gap between vertically neighbouring corner points. Vertical gaps greater than this will introduce k gaps into resqml grid. Units are corp z units
- **split_pillars** (*bool, default True*) – if False an unfaulted grid will be generated
- **split_tolerance** (*float, default 0.01*) – maximum distance between neighbouring corner points before a pillar is considered 'split'. Applies to each of x, y, z differences
- **ijk_handedness** (*str, default 'right'*) – 'right' or 'left'
- **known_to_be_straight** (*bool, default False*) – if True pillars are forced to be straight

Notes

this function sets up all the geometry arrays in memory but does not write to hdf5 nor create xml: use Grid methods; geometry_defined_everywhere is deprecated, use treat_as_nan instead

7.11.4 resqpy.rq_import.import_nexus

resqpy.rq_import.**import_nexus**(*resqml_file_root*, *extent_ijk=None*, *vdb_file=None*, *vdb_case=None*, *corp_file=None*, *corp_bin_file=None*, *corp_xy_units='m'*, *corp_z_units='m'*, *corp_z_inc_down=True*, *ijk_handedness='right'*, *corp_eight_mode=False*, *geometry_defined_everywhere=True*, *treat_as_nan=None*, *active_mask_file=None*, *use_binary=False*, *resqml_xy_units='m'*, *resqml_z_units='m'*, *resqml_z_inc_down=True*, *shift_to_local=False*, *local_origin_place='centre'*, *max_z_void=0.1*, *split_pillars=True*, *split_tolerance=0.01*, *property_array_files=None*, *summary_file=None*, *vdb_static_properties=True*, *vdb_recurrent_properties=False*, *timestep_selection='all'*, *use_compressed_time_series=True*, *decoarsen=True*, *ab_property_list=None*, *create_property_set=False*, *ensemble_case_dirs_root=None*, *ensemble_property_dictionary=None*, *ensemble_size_limit=None*, *grid_title='ROOT'*, *mode='w'*, *progress_fn=None*)

Read a simulation grid geometry and optionally grid properties.

Input may be from nexus ascii input files, or nexus vdb output.

Parameters

- **resqml_file_root** (*str*) – output path and file name without .epc or .h5 extension
- **extent_ijk** (*triple float, optional*) – ijk extents (fortran ordering)
- **vdb_file** (*str, optional*) – vdb input file, either this or corp_file should be not None. Required if importing from a vdb
- **vdb_case** (*str, optional*) – required if the vdb contains more than one case. If None, first case in vdb is used
- **corp_file** (*str, optional*) – required if importing from corp ascii file. corp ascii input file: nexus corp data without keyword
- **corp_bin_file** (*str, optional*) – required if importing from corp binary file

- **corp_xy_units** (*str, default 'm'*) – xy length units
- **corp_z_units** (*str, default 'm'*) – z length units
- **corp_z_inc_down** (*bool, default True*) – if True z values increase with depth
- **ijk_handedness** (*str, default 'right'*) – 'right' or 'left'
- **corp_eight_mode** (*bool, default False*) – if True the ordering of corner point data is in nexus EIGHT mode
- **geometry_defined_everywhere** (*bool, default True*) – if False then inactive cells are marked as not having geometry
- **treat_as_nan** (*float, default None*) – if a value is provided corner points with this value will be assigned nan
- **active_mask_file** (*str, default None*) – ascii property file holding values 0 or 1, with 1 indicating active cells
- **use_binary** (*bool, default False*) – if True a cached binary version of ascii files will be used (pure binary, not corp bin format)
- **resqml_xy_units** (*str, default 'm'*) – output xy units for resqml file
- **resqml_z_units** (*str, default 'm'*) – output z units for resqml file
- **resqml_z_inc_down** (*bool, default True*) – if True z values increase with depth for output resqml file
- **shift_to_local** (*bool, default False*) – if True then a local origin will be used in the CRS
- **local_origin_place** (*str, default 'centre'*) – 'centre' or 'minimum'. If 'centre' the local origin is placed at the centre of the grid; ignored if shift_to_local is False
- **max_z_void** (*float, default 0.1*) – maximum z gap between vertically neighbouring corner points. Vertical gaps greater than this will introduce k gaps into resqml grid. Units are corp z units
- **split_pillars** (*bool, default True*) – if False an unfaulted grid will be generated
- **split_tolerance** (*float, default 0.01*) – maximum distance between neighbouring corner points before a pillar is considered 'split'. Applies to each of x, y, z differences
- **property_array_files** (*list, default None*) – list of (filename, keyword, uom, time_index, null_value, discrete)
- **summary_file** (*str, default None*) – nexus output summary file, used to extract timestep dates when loading recurrent data from vdb
- **vdb_static_properties** (*bool, default True*) – if True, static vdb properties are imported (only relevant if vdb_file is not None)
- **vdb_recurrent_properties** (*bool, default False*) – # if True, recurrent vdb properties are imported (only relevant if vdb_file is not None)
- **timestep_selection** (*str, default 'all*) – 'first', 'last', 'first and last', 'all', or list of ints being reporting timestep numbers. Ignored if vdb_recurrent_properties is False
- **use_compressed_time_series** (*bool, default True*) – generates reduced time series containing timesteps with recurrent properties from vdb, rather than full nexus summary time series

- **decoarsen** (*bool, default True*) – where ICOARSE is present, redistribute data to uncoarse cells
- **ab_property_list** (*list, default None*) – list of (file_name, keyword, property_kind, facet_type, facet, uom, time_index, null_value, discrete)
- **create_property_set** (*bool, default False*) – if True a resqml PropertySet is created
- **ensemble_case_dirs_root** (*str, default None*) – path up to but excluding realisation number
- **ensemble_property_dictionary** (*str, default None*) – dictionary mapping title (or keyword) to (filename, property_kind, facet_type, facet, uom, time_index, null_value, discrete)
- **ensemble_size_limit** (*int, default None*) – if present processing of ensemble will terminate after this number of cases is reached
- **grid_title** (*str, default 'ROOT'*) – grid citation title
- **mode** (*str, default 'w'*) – 'w' or 'a', mode to write or append to hdf5
- **progress_fn** (*function, default None*) – if present function must have one floating argument with value increasing from 0 to 1, and is called at intervals to indicate progress

Returns

resqml model in memory & written to disc

7.11.5 resqpy.rq_import.import_vdb_all_grids

resqpy.rq_import.**import_vdb_all_grids**(*resqml_file_root*, *extent_ijk=None*, *vdb_file=None*, *vdb_case=None*, *corp_xy_units='m'*, *corp_z_units='m'*, *corp_z_inc_down=True*, *ijk_handedness='right'*, *geometry_defined_everywhere=True*, *treat_as_nan=None*, *resqml_xy_units='m'*, *resqml_z_units='m'*, *resqml_z_inc_down=True*, *shift_to_local=False*, *local_origin_place='centre'*, *max_z_void=0.1*, *split_pillars=True*, *split_tolerance=0.01*, *vdb_static_properties=True*, *vdb_recurrent_properties=False*, *decoarsen=True*, *timestep_selection='all'*, *create_property_set=False*)

Creates a RESQML dataset containing grids and grid properties, including LGRs, for a single realisation.

Parameters

- **resqml_file_root** (*str*) – output path and file name without .epc or .h5 extension
- **extent_ijk** (*triple float, optional*) – ijk extents (fortran ordering)
- **vdb_file** (*str, optional*) – vdb input file, either this or corp_file should be not None. Required if importing from a vdb
- **vdb_case** (*str, optional*) – required if the vdb contains more than one case. If None, first case in vdb is used
- **corp_xy_units** (*str, default 'm'*) – xy length units
- **corp_z_units** (*str, default 'm'*) – z length units
- **corp_z_inc_down** (*bool, default True*) – if True z values increase with depth
- **ijk_handedness** (*str, default 'right'*) – 'right' or 'left'

- **geometry_defined_everywhere** (*bool, default True*) – if False then inactive cells are marked as not having geometry
- **treat_as_nan** (*float, default None*) – if a value is provided corner points with this value will be assigned nan
- **resqml_xy_units** (*str, default 'm'*) – output xy units for resqml file
- **resqml_z_units** (*str, default 'm'*) – output z units for resqml file
- **resqml_z_inc_down** (*bool, default True*) – if True z values increase with depth for output resqml file
- **shift_to_local** (*bool, default False*) – if True then a local origin will be used in the CRS
- **local_origin_place** (*str, default 'centre'*) – 'centre' or 'minimum'. If 'centre' the local origin is placed at the centre of the grid; ignored if shift_to_local is False
- **max_z_void** (*float, default 0.1*) – maximum z gap between vertically neighbouring corner points. Vertical gaps greater than this will introduce k gaps into resqml grid. Units are corp z units
- **split_pillars** (*bool, default True*) – if False an unfaulted grid will be generated
- **split_tolerance** (*float, default 0.01*) – maximum distance between neighbouring corner points before a pillar is considered 'split'. Applies to each of x, y, z differences
- **vdb_static_properties** (*bool, default True*) – if True, static vdb properties are imported (only relevant if vdb_file is not None)
- **vdb_recurrent_properties** (*bool, default False*) – # if True, recurrent vdb properties are imported (only relevant if vdb_file is not None)
- **decoarsen** (*bool, default True*) – where ICOARSE is present, redistribute data to uncoarse cells
- **timestep_selection** (*str, default 'all'*) – 'first', 'last', 'first and last', 'all', or list of ints being reporting timestep numbers. Ignored if vdb_recurrent_properties is False
- **create_property_set** (*bool, default False*) – if True a resqml PropertySet is created

7.11.6 resqpy.rq_import.import_vdb_ensemble

resqpy.rq_import.**import_vdb_ensemble**(*epc_file*, *ensemble_run_dir*, *existing_epc=False*, *keyword_list=None*, *property_kind_list=None*, *vdb_static_properties=True*, *vdb_recurrent_properties=True*, *decoarsen=True*, *timestep_selection='all'*, *create_property_set_per_realization=True*, *create_property_set_per_timestep=True*, *create_complete_property_set=False*, *extent_ijk=None*, *corp_xy_units='m'*, *corp_z_units='m'*, *corp_z_inc_down=True*, *ijk_handedness='right'*, *geometry_defined_everywhere=True*, *treat_as_nan=None*, *resqml_xy_units='m'*, *resqml_z_units='m'*, *resqml_z_inc_down=True*, *shift_to_local=True*, *local_origin_place='centre'*, *max_z_void=0.1*, *split_pillars=True*, *split_tolerance=0.01*, *progress_fn=None*)

Adds properties from all vdb's within an ensemble directory tree to a single RESQML dataset.

Referencing a shared grid.

Parameters

- **epc_file** (*string*) – filename of epc file to be extended with ensemble properties
- **ensemble_run_dir** (*string*) – path of main ensemble run directory; vdb's within this directory tree are source of import
- **existing_epc** (*boolean, default False*) – if True, the epc_file must already exist and contain the compatible grid
- **keyword_list** (*list of strings, optional*) – if present, only properties for keywords within the list are included
- **property_kind_list** (*list of strings, optional*) – if present, only properties which are mapped to these resqml property kinds are included in the import
- **vdb_static_properties** (*boolean, default True*) – if False, no static properties are included, regardless of keyword and/or property kind matches
- **vdb_recurrent_properties** (*boolean, default True*) – if False, no recurrent properties are included, regardless of keyword and/or property kind matches
- **decoarsen** (*boolean, default True*) – if True and ICOARSE property exists for a grid in a case, the associated property data is decoarsened; if False, the property data is as stored in the vdb
- **timestep_selection** (*string, default 'all'*) – may be 'first', 'last', 'first and last', or 'all', controlling which reporting timesteps are included when loading recurrent data
- **create_property_set_per_realization** (*boolean, default True*) – if True, a property set object is created for each realization
- **create_property_set_per_timestep** (*boolean, default True*) – if True, a property set object is created for each timestep included in the recurrent data import
- **create_complete_property_set** (*boolean, default False*) – if True, a property set object is created containing all the properties imported; only really useful to differentiate from other properties related to the grid
- **extent_ijk** (*triple int, optional*) – this and remaining arguments are only used if existing_epc is False; the extent is only needed in case automatic determination of the extent fails
- **corp_xy_units** (*string, default 'm'*) – the units of x & y values in the vdb corp data; typically 'm' (metres), 'ft' (feet) or 'cm' (centimetres, for lab scale models)
- **corp_z_units** (*string, default 'm'*) – the units of z values in the vdb corp data; typically 'm' (metres), 'ft' (feet) or 'cm' (centimetres, for lab scale models)
- **corp_z_inc_down** (*boolean, default True*) – set to True if corp z values are depth; False if elevation
- **ijk_handedness** (*string, default 'right'*) – set to the handedness of the IJK axes in the Nexus model; 'right' or 'left'
- **geometry_defined_everywhere** (*boolean, default True*) – set to False if inactive cells do not have valid geometry; deprecated - use treat_as_nan argument instead
- **treat_as_nan** (*string, optional*) – if not None, one of 'dots', 'ij_dots', 'inactive'; controls which inactive cells have their geometry set to undefined

- **resqml_xy_units** (*string, default 'm'*) – the units of x & y values to use in the generated resqml grid; typically 'm' (metres), 'ft' (feet) or 'cm' (centimetres, for lab scale models)
- **resqml_z_units** (*string, default 'm'*) – the units of z values to use in the generated resqml grid; typically 'm' (metres), 'ft' (feet) or 'cm' (centimetres, for lab scale models)
- **resqml_z_inc_down** (*boolean, default True*) – set to True if resqml z values are to be depth; False for elevations
- **shift_to_local** (*boolean, default True*) – if True, the resqml coordinate reference system will use a local origin
- **local_origin_place** (*string, default 'centre'*) – where to place the local origin; 'centre' or 'minimum'; only relevant if shift_to_local is True
- **max_z_void** (*float, default 0.1*) – the tolerance of voids between layers, in z direction; voids greater than this will cause the grid import to fail
- **split_pillars** (*boolean, default True*) – if False, a grid is generated without split pillars
- **split_tolerance** (*float, default 0.01*) – the tolerance applied to each of x, y, & z values, beyond which a corner point (and hence pillar) will be split
- **progress_fn** (*function(float), optional*) – if present, this function is called at intervals during processing; it must accept one floating point argument which will range from 0.0 to 1.0

Returns

resqpy.Model object containing properties for all the realisations; hdf5 and epc files having been updated

---

Note: if existing_epc is True, the epc file must already exist and contain one grid (or one grid named ROOT) which must have the correct extent for all realisations within the ensemble; if existing_epc is False, the resqml dataset is created afresh with a grid extracted from the first realisation in the ensemble; either way, the single grid is used as the representative grid in the ensemble resqml dataset being generated; all vdb directories within the directory tree headed by ensemble_run_dir are included in the import; by default all properties will be imported; the keyword_list, property_kind_list, vdb_static_properties, vdb_recurrent_properties and timestep_selection arguments can be used to filter the required properties; if both keyword_list and property_kind_list are provided, a property must match an item in both lists in order to be included; if recurrent properties are being included then all vdb's should contain the same number of reporting steps in their recurrent data and these should relate to the same set of timestamps; timestamp data is extracted from a summary file for the first realisation; no check is made to ensure that reporting timesteps in different realisations are actually for the same date.

7.12 resqpy.strata

Stratigraphy related classes and valid values.

Classes

BinaryContactInterpretation	Internal class for contact between 2 geological entities; not a high level class but used by others.
GeologicUnitInterpretation	Class for RESQML Geologic Unit Interpretation objects.
StratigraphicColumn	Class for RESQML stratigraphic column objects.
StratigraphicColumnRank	Class for RESQML StratigraphicColumnRankInterpretation objects.
StratigraphicUnitFeature	Class for RESQML Stratigraphic Unit Feature objects.
StratigraphicUnitInterpretation	Class for RESQML Stratigraphic Unit Interpretation objects.

7.12.1 resqpy.strata.BinaryContactInterpretation

class resqpy.strata.**BinaryContactInterpretation**(*model*, *existing_xml_node=None*, *index=None*, *contact_relationship: Optional[str] = None*, *verb: Optional[str] = None*, *subject_uuid=None*, *direct_object_uuid=None*, *subject_contact_side=None*, *subject_contact_mode=None*, *direct_object_contact_side=None*, *direct_object_contact_mode=None*, *part_of_uuid=None*)

Bases: object

Internal class for contact between 2 geological entities; not a high level class but used by others.

Methods:

__init__(model[, existing_xml_node, index, ...])	Creates a new binary contact interpretation internal object.
create_xml([parent_node])	Generates xml sub-tree for this contact interpretation, for inclusion as element of high level interpretation.

__init__(*model*, *existing_xml_node=None*, *index=None*, *contact_relationship: Optional[str] = None*, *verb: Optional[str] = None*, *subject_uuid=None*, *direct_object_uuid=None*, *subject_contact_side=None*, *subject_contact_mode=None*, *direct_object_contact_side=None*, *direct_object_contact_mode=None*, *part_of_uuid=None*)

Creates a new binary contact interpretation internal object.

Note: if an existing xml node is present, then all the later arguments are ignored

create_xml(*parent_node=None*)

Generates xml sub-tree for this contact interpretation, for inclusion as element of high level interpretation.

Parameters
parent_node (*lxml.etree._Element, optional*) – if present, the created sub-tree is added as a child to this node

Returns
lxml.etree._Element – the root node of the newly created xml sub-tree for the contact interpretation

7.13 resqpy.surface

Classes for RESQML objects related to surfaces.

Classes

BaseSurface	Base class to implement shared methods for other classes in this module.
CombinedSurface	Class allowing a collection of Surface objects to be treated as a single surface.
Mesh	Class covering meshes (lattices: surfaces where points form a 2D grid; RESQML obj_Grid2dRepresentation).
PointSet	Class for RESQML Point Set Representation within resqpy model object.
Surface	Class for RESQML triangulated set surfaces.
TriMesh	Class of mesh using equilateral triangles in the xy plane.
TriangulatedPatch	Class for RESQML TrianglePatch objects (used by Surface objects inter alia).

7.13.1 resqpy.surface.CombinedSurface

class resqpy.surface.**CombinedSurface**(*surface_list*, *crs_uuid=None*)

Bases: object

Class allowing a collection of Surface objects to be treated as a single surface.

Not a RESQML class in its own right.

Methods:

__init__(surface_list[, crs_uuid])	Initialise a CombinedSurface object from a list of Surface (and/or CombinedSurface) objects.
surface_index_for_triangle_index(tri_index)	Return the index of the surface containing the triangle and local triangle index.
triangles_and_points()	Returns the composite triangles and points for the combined surface.

__init__(*surface_list*, *crs_uuid=None*)

Initialise a CombinedSurface object from a list of Surface (and/or CombinedSurface) objects.

Parameters

- **surface_list** (*list of Surface and/or CombinedSurface objects*) – the new object is the combination of these surfaces
- **crs_uuid** (*uuid.UUID, optional*) – if present, all contributing surfaces must refer to this crs

---

Note: all contributing surfaces should be established before initialising this object; all contributing surfaces must refer to the same crs; this class of object is not part of the RESQML standard and cannot be saved in a RESQML dataset - it is a high level derived object class

---

surface_index_for_triangle_index(*tri_index*)

Return the index of the surface containing the triangle and local triangle index.

Parameters

tri_index – triangle index in the combined surface

triangles_and_points()

Returns the composite triangles and points for the combined surface.

7.13.2 resqpy.surface.TriangulatedPatch

class resqpy.surface.**TriangulatedPatch**(*parent_model*, *patch_index=None*, *patch_node=None*, *crs_uuid=None*)

Bases: object

Class for RESQML TrianglePatch objects (used by Surface objects inter alia).

Methods:

__init__(parent_model[, patch_index, ...])	Create an empty TriangulatedPatch (TrianglePatch) node and optionally load from xml.
extract_crs_root_and_uuid()	Caches uuid for coordinate reference system, as stored in geometry xml sub-tree.
triangles_and_points()	Returns arrays representing the patch.
set_to_trimmed_patch(larger_patch[, ...])	Populate this (empty) patch with triangles and points that overlap with a trimming volume.
set_to_horizontal_plane(depth, box_xyz[, ...])	Populate this (empty) patch with two triangles defining a flat, horizontal plane at a given depth.
set_to_triangle(corners)	Populate this (empty) patch with a single triangle.
set_to_triangle_pair(corners)	Populate this (empty) patch with a pair of triangles.
set_from_triangles_and_points(triangles, points)	Populate this (empty) patch from triangle node indices and points from elsewhere.
set_to_sail(n, centre, radius, azimuth, ...)	Populate this (empty) patch with triangles for a big triangle wrapped on a sphere.
set_from_irregular_mesh(mesh_xyz[, ...])	Populate this (empty) patch from an untorn mesh array of shape (N, M, 3).
set_from_sparse_mesh(mesh_xyz)	Populate this (empty) patch from a mesh array of shape (N, M, 3), with some NaNs in z.
get_indices_from_sparse_meshxyz(mesh_xyz)	Update self.points and self.node_count with non-nan points in a given mesh_xyz array.
set_from_torn_mesh(mesh_xyz[, quad_triangles])	Populate this (empty) patch from a torn mesh array of shape (nj, ni, 2, 2, 3).
column_from_triangle_index(triangle_index)	For patch freshly built from fully defined mesh, returns (j, i) for given triangle index.
set_to_cell_faces_from_corner_points(cp[, ...])	Populates this (empty) patch to represent faces of a cell, from corner points of shape (2, 2, 2, 3).
get_triangles_for_cell_faces_quad_false(cp)	Returns the triangles for corner points representing cell faces, where quad_triangles is False.
get_triangles_for_cell_faces_quad_true(cp)	Returns the triangles for corner points representing cell faces, where quad_triangles is True.
face_from_triangle_index(triangle_index)	For patch freshly built for cell faces, returns (axis, polarity) for given triangle index.
vertical_rescale_points(ref_depth, ...)	Rescale points along vertical direction.

__init__(*parent_model*, *patch_index=None*, *patch_node=None*, *crs_uuid=None*)

Create an empty TriangulatedPatch (TrianglePatch) node and optionally load from xml.

Note: not usually instantiated directly by application code

column_from_triangle_index(*triangle_index*)

For patch freshly built from fully defined mesh, returns (j, i) for given triangle index.

argument:

triangle_index (int or numpy int array): the triangle index (or array of indices) for which column(s) are being sought

Returns

pair of ints or pair of numpy int arrays – the (j0, i0) indices of the column(s) which the triangle(s) is/are part of

Notes

this function will only work if the surface has been freshly constructed with data from a mesh without NaNs, otherwise (None, None) will be returned; if triangle_index is a numpy int array, a pair of similarly shaped numpy arrays is returned

extract_crs_root_and_uuid()

Caches uuid for coordinate reference system, as stored in geometry xml sub-tree.

face_from_triangle_index(*triangle_index*)

For patch freshly built for cell faces, returns (axis, polarity) for given triangle index.

get_indices_from_sparse_meshxyz(*mesh_xyz*)

Update self.points and self.node_count with non-nan points in a given mesh_xyz array.

Returns the indices of these non_nan points.

get_triangles_for_cell_faces_quad_false(*cp*)

Returns the triangles for corner points representing cell faces, where quad_triangles is False.

get_triangles_for_cell_faces_quad_true(*cp*)

Returns the triangles for corner points representing cell faces, where quad_triangles is True.

set_from_irregular_mesh(*mesh_xyz*, *quad_triangles=False*)

Populate this (empty) patch from an untorn mesh array of shape (N, M, 3).

set_from_sparse_mesh(*mesh_xyz*)

Populate this (empty) patch from a mesh array of shape (N, M, 3), with some NaNs in z.

set_from_torn_mesh(*mesh_xyz*, *quad_triangles=False*)

Populate this (empty) patch from a torn mesh array of shape (nj, ni, 2, 2, 3).

set_from_triangles_and_points(*triangles*, *points*)

Populate this (empty) patch from triangle node indices and points from elsewhere.

set_to_cell_faces_from_corner_points(*cp*, *quad_triangles=True*)

Populates this (empty) patch to represent faces of a cell, from corner points of shape (2, 2, 2, 3).

set_to_horizontal_plane(*depth*, *box_xyz*, *border=0.0*, *quad_triangles=False*)

Populate this (empty) patch with two triangles defining a flat, horizontal plane at a given depth.

Parameters

- **depth** (*float*) – z value to use in all points in the triangulated patch
- **box_xyz** (*float[2, 3]*) – the min, max values of x, y (&z) giving the area to be covered (z ignored)
- **border** (*float*) – an optional border width added around the x,y area defined by box_xyz
- **quad_triangles** (*bool, default False*) – if True, 4 triangles are used instead of 2

set_to_sail(*n*, *centre*, *radius*, *azimuth*, *delta_theta*)

Populate this (empty) patch with triangles for a big triangle wrapped on a sphere.

set_to_triangle(*corners*)

Populate this (empty) patch with a single triangle.

set_to_triangle_pair(*corners*)

Populate this (empty) patch with a pair of triangles.

set_to_trimmed_patch(*larger_patch*, *xyz_box=None*, *xy_polygon=None*, *internal=False*)

Populate this (empty) patch with triangles and points that overlap with a trimming volume.

Parameters

- **larger_patch** (TriangulatedPatch) – the larger patch, a copy of which is to be trimmed
- **xyz_box** (*numpy float array of shape (2, 3), optional*) – if present, a cuboid in xyz space against which to trim the patch
- **xy_polygon** (*closed convex resqpy.lines.Polyline, optional*) – if present, an xy boundary against which to trim
- **internal** (*bool, default False*) – if True, only those triangles where all three vertices are wtihin the trimming space are kept; if False, triangles with at least one vertex within the space are kept

Notes

at least one of xyz_box or xy_polygon must be present; if both are present, a triangle must be within both boundaries to survive the trimming; xyz_box and xy_polygon must be in the same crs as the larger patch

triangles_and_points()

Returns arrays representing the patch.

Returns

Tuple (triangles, points) –

- triangles (int array of shape[:, 3]): integer indices into points array, being the nodes of the corners of the triangles
- points (float array of shape[:, 3]): flat array of xyz points, indexed by triangles

vertical_rescale_points(*ref_depth*, *scaling_factor*)

Rescale points along vertical direction.

Modifies the z values of points for this patch by stretching the distance from reference depth by scaling factor.

Functions

distill_triangle_points	Returns a (triangles, points) pair with points distilled as only those used from p.

7.13.3 resqpy.surface.distill_triangle_points

resqpy.surface.**distill_triangle_points**(*t*, *p*)

Returns a (triangles, points) pair with points distilled as only those used from p.

7.14 resqpy.time_series

Time series classes and functions.

Classes

AnyTimeSeries	Abstract class for a RESQML Time Series; use resqpy TimeSeries or GeologicTimeSeries.
GeologicTimeSeries	Class for RESQML Time Series using only year offsets (for geological time frames).
TimeDuration	A thin wrapper around python's datetime timedelta objects (not a RESQML class).
TimeSeries	Class for RESQML Time Series without year offsets.

7.14.1 resqpy.time_series.TimeDuration

class resqpy.time_series.**TimeDuration**(*days=None*, *hours=None*, *minutes=None*, *seconds=None*, *earlier_timestamp=None*, *later_timestamp=None*)

Bases: object

A thin wrapper around python's datetime timedelta objects (not a RESQML class).

Methods:

__init__([days, hours, minutes, seconds, ...])	Create a TimeDuration object either from days and seconds or from a pair of timestamps.
timestamp_after_duration(earlier_timestamp)	Create a new timestamp from this duration and an earlier timestamp.
timestamp_before_duration(later_timestamp)	Create a new timestamp from this duration and a later timestamp.

__init__(*days=None*, *hours=None*, *minutes=None*, *seconds=None*, *earlier_timestamp=None*, *later_timestamp=None*)

Create a TimeDuration object either from days and seconds or from a pair of timestamps.

timestamp_after_duration(*earlier_timestamp*)

Create a new timestamp from this duration and an earlier timestamp.

timestamp_before_duration(*later_timestamp*)

Create a new timestamp from this duration and a later timestamp.

Functions

any_time_series	Returns a resqpy TimeSeries or GeologicTimeSeries object for an existing RESQML time series with a given uuid.
check_timestamp	Check format of timestamp and raise ValueError if badly formed.
cleaned_timestamp	Return a cleaned version of the timestamp.
colloquial_date	Returns date string in format DD/MM/YYYY (or MM/DD/YYYY if usa_date_format is True).
geologic_time_str	Returns a string representing a geological time for a large int representing number of years before present.
merge_timeseries_from_uuid	Create a TimeSeries object from an iterable object of existing timeseries UUIDs of timeseries.
selected_time_series	Returns a new TimeSeries or GeologicTimeSeries object with timestamps selected from the full series by a list of indices.
simplified_timestamp	Return a more readable version of the timestamp.
time_series_from_list	Create a TimeSeries object from a list of timestamps (model and node set to None).
time_series_from_nexus_summary	Create a TimeSeries object based on time steps reported in a Nexus summary file (.sum).
timeframe_for_time_series_uuid	Returns string 'human' or 'geologic' indicating timeframe of the RESQML time series with a given uuid.

7.14.2 resqpy.time_series.any_time_series

resqpy.time_series.**any_time_series**(*parent_model*, *uuid*)

Returns a resqpy TimeSeries or GeologicTimeSeries object for an existing RESQML time series with a given uuid.

7.14.3 resqpy.time_series.check_timestamp

resqpy.time_series.**check_timestamp**(*timestamp*)

Check format of timestamp and raise ValueError if badly formed.

7.14.4 resqpy.time_series.cleaned_timestamp

resqpy.time_series.**cleaned_timestamp**(*timestamp*)

Return a cleaned version of the timestamp.

7.14.5 resqpy.time_series.colloquial_date

resqpy.time_series.**colloquial_date**(*timestamp*, *usa_date_format=False*)

Returns date string in format DD/MM/YYYY (or MM/DD/YYYY if usa_date_format is True).

7.14.6 resqpy.time_series.geologic_time_str

resqpy.time_series.**geologic_time_str**(*years*)

Returns a string representing a geological time for a large int representing number of years before present.

7.14.7 resqpy.time_series.merge_timeseries_from_uuid

resqpy.time_series.**merge_timeseries_from_uuid**(*model*, *timeseries_uuid_iter*)

Create a TimeSeries object from an iterable object of existing timeseries UUIDs of timeseries.

iterable can be a list, array, or iterable generator (model must be provided). The new timeseries is sorted in ascending order. Returns the new time series, the new time series uuid, and the list of timeseries objects used to generate the list

7.14.8 resqpy.time_series.selected_time_series

resqpy.time_series.**selected_time_series**(*full_series*, *indices_list*, *title=None*)

Returns a new TimeSeries or GeologicTimeSeries object with timestamps selected from the full series by a list of indices.

7.14.9 resqpy.time_series.simplified_timestamp

resqpy.time_series.**simplified_timestamp**(*timestamp*)

Return a more readable version of the timestamp.

7.14.10 resqpy.time_series.time_series_from_list

resqpy.time_series.**time_series_from_list**(*timestamp_list*, *parent_model=None*, *title=None*)

Create a TimeSeries object from a list of timestamps (model and node set to None).

Note: timestamps in the list should be in the correct string format for human timeframe series, or large negative integers for geologic timeframe series

---

7.14.11 resqpy.time_series.time_series_from_nexus_summary

resqpy.time_series.**time_series_from_nexus_summary**(*summary_file*, *parent_model=None*, *start_date=None*)

Create a TimeSeries object based on time steps reported in a Nexus summary file (.sum).

Parameters

- **summary_file** (*str*) – path of Nexus summary file
- **parent_model** (Model) – the model to which the new time series will be attached
- **start_date** (*str, optional*) – if the summary file does not contain dates, this will be used as the start date; required format is 'YYYY-MM-DD'

Returns

newly created TimeSeries

Note: this function does not create the xml for the new TimeSeries, nor add it as a part to the parent model

7.14.12 resqpy.time_series.timeframe_for_time_series_uuid

resqpy.time_series.**timeframe_for_time_series_uuid**(*model*, *uuid*)

Returns string 'human' or 'geologic' indicating timeframe of the RESQML time series with a given uuid.

7.15 resqpy.unstructured

Unstructured grid and derived classes.

Classes

HexaGrid	Class for unstructured grids where every cell is hexahedral (faces may be degenerate).
PrismGrid	Class for unstructured grids where every cell is a triangular prism.
PyramidGrid	Class for unstructured grids where every cell is a quadrilateral pyramid.
TetraGrid	Class for unstructured grids where every cell is a tetrahedron.
UnstructuredGrid	Class for RESQML Unstructured Grid objects.
VerticalPrismGrid	Class for unstructured grids where every cell is a vertical triangular prism.

7.16 resqpy.weights_and_measures

Weights and measures valid units and unit conversion functions.

Functions

convert	Convert value between two compatible units.
convert_flow_rates	Converts values in numpy array (or a scalar) from one volume flow rate unit to another, in situ if array.
convert_lengths	Converts values in numpy array (or a scalar) from one length unit to another, in situ if array.
convert_pressures	Converts values in numpy array (or a scalar) from one pressure unit to another, in situ if array.
convert_times	Converts values in numpy array (or a scalar) from one time unit to another, in situ if array.
convert_transmissibilities	Converts values in numpy array (or a scalar) from one transmissibility unit to another, in situ if array.
convert_volumes	Converts values in numpy array (or a scalar) from one volume unit to another, in situ if array.
get_conversion_factors	Return base unit and conversion factors (A, B, C, D) for a given uom.
nexus_uom_for_quantity	Returns RESQML uom string expected by Nexus for given quantity class and unit system.
rq_length_unit	Returns length units string as expected by resqml.
rq_time_unit	Returns time units string as expected by resqml.
rq_uom	Returns RESQML uom string equivalent to units.
rq_uom_list	Returns a list of RESQML uom equivalents for units in list.
valid_property_kinds	Return set of valid property kinds.
valid_quantities	Return set of valid RESQML quantities.
valid_uoms	Return set of valid RESQML units of measure.

7.16.1 resqpy.weights_and_measures.convert

resqpy.weights_and_measures.**convert**(*x*, *unit_from*, *unit_to*, *quantity=None*, *inplace=False*)

Convert value between two compatible units.

Parameters

- **x** (*numeric or np.array*) – value(s) to convert
- **unit_from** (*str*) – resqml uom
- **unit_to** (*str*) – resqml uom
- **quantity** (*str, optional*) – If provided, raise an exception if units are not supported by this quantity
- **inplace** (*bool*) – if True, convert arrays in-place. Else, return new value

Returns

Converted value(s)

Raises

- *InvalidUnitError* – if units cannot be coerced into RESQML units
- *IncompatibleUnitsError* – if units do not have compatible base units

7.16.2 resqpy.weights_and_measures.convert_flow_rates

resqpy.weights_and_measures.**convert_flow_rates**(*a*, *from_units*, *to_units*)

Converts values in numpy array (or a scalar) from one volume flow rate unit to another, in situ if array.

Parameters

- **a** (*numpy float array, or float*) – array of volume flow rate values to undergo unit conversion in situ, or a scalar
- **from_units** (*string*) – units of the data before conversion, eg. 'm3/d'; see notes for acceptable units
- **to_units** (*string*) – required units of the data after conversion, eg. 'ft3/d'; see notes for acceptable units

Returns

a after unit conversion

Note: To see supported units, use: *valid_uoms(quantity='volume per time')*

7.16.3 resqpy.weights_and_measures.convert_lengths

resqpy.weights_and_measures.**convert_lengths**(*a*, *from_units*, *to_units*)

Converts values in numpy array (or a scalar) from one length unit to another, in situ if array.

Parameters

- **a** (*numpy float array, or float*) – array of length values to undergo unit conversion in situ, or a scalar
- **from_units** (*string*) – the units of the data before conversion
- **to_units** (*string*) – the required units

Returns

a after unit conversion

Note: To see supported units, use: *valid_uoms(quantity='length')*

7.16.4 resqpy.weights_and_measures.convert_pressures

resqpy.weights_and_measures.**convert_pressures**(*a*, *from_units*, *to_units*)

Converts values in numpy array (or a scalar) from one pressure unit to another, in situ if array.

Parameters

- **a** (*numpy float array, or float*) – array of pressure values to undergo unit conversion in situ, or a scalar
- **from_units** (*string*) – the units of the data before conversion
- **to_units** (*string*) – the required units

Returns

a after unit conversion

Note: To see supported units, use: *valid_uoms(quantity='pressure')*

7.16.5 resqpy.weights_and_measures.convert_times

resqpy.weights_and_measures.**convert_times**(*a*, *from_units*, *to_units*, *invert=False*)

Converts values in numpy array (or a scalar) from one time unit to another, in situ if array.

Note: To see supported units, use: *valid_uoms(quantity='time')*

7.16.6 resqpy.weights_and_measures.convert_transmissibilities

resqpy.weights_and_measures.**convert_transmissibilities**(*a*, *from_units*, *to_units*)

Converts values in numpy array (or a scalar) from one transmissibility unit to another, in situ if array.

Parameters

- **a** (*numpy float array, or float*) – array of transmissibility values to undergo unit conversion in situ, or a scalar
- **from_units** (*string*) – units of the data before conversion, eg. 'm3.cP/(kPa.d)'; see notes for acceptable units
- **to_units** (*string*) – required units of the data after conversion, eg. 'bbl.cP/(psi.d)'; see notes for acceptable units

Returns

a after unit conversion

Note: for transmissibility data, resqpy expects a viscosity term in the units of measure (unlike the Energistics standard at the time of RESQML 2.0.1); examples are: 'm3.cP/(kPa.d)' or 'bbl.cP/(psi.d)'; the general form of the units strings must be A.B/(C.D) where A is a valid unit of volume, B is a valid unit of dynamic viscosty, C is a valid unit of pressure, and D is a valid unit of time; use wam.valid_uoms(quantity = 'dynamic viscosity') etc. to discover the allowed components of the unit strings

7.16.7 resqpy.weights_and_measures.convert_volumes

resqpy.weights_and_measures.**convert_volumes**(*a*, *from_units*, *to_units*)

Converts values in numpy array (or a scalar) from one volume unit to another, in situ if array.

Parameters

- **a** (*numpy float array, or float*) – array of volume values to undergo unit conversion in situ, or a scalar
- **from_units** (*string*) – units of the data before conversion; see note for accepted units
- **to_units** (*string*) – the required units; see note for accepted units

Returns

a after unit conversion

Note: To see supported units, use: *valid_uoms(quantity='volume')*

7.16.8 resqpy.weights_and_measures.get_conversion_factors

resqpy.weights_and_measures.**get_conversion_factors**(*uom*)

Return base unit and conversion factors (A, B, C, D) for a given uom.

The formula "y=(A + Bx)/(C + Dx)" where "y" represents a value in the base unit.

Returns

3-tuple of (base_unit, dimension, factors). Factors is a 4-tuple of conversion factors

Raises

ValueError if either uom is not a valid resqml uom –

7.16.9 resqpy.weights_and_measures.nexus_uom_for_quantity

resqpy.weights_and_measures.**nexus_uom_for_quantity**(*nexus_unit_system*, *quantity*, *english_volume_flavour=None*)

Returns RESQML uom string expected by Nexus for given quantity class and unit system.

Parameters

- **nexus_unit_system** (*str*) – one of 'METRIC', 'METKG/CM2', 'METBAR', 'LAB', or 'ENGLISH'
- **quantity** (*str*) – the RESQML quantity class of interest; currently suppported: 'length', 'area', 'volume', 'volume per volume', 'permeability rock', 'time', 'thermodynamic temperature', 'mass per volume', 'pressure', 'volume per time'
- **english_volume_flavour** (*str, optional*) – only needed for ENGLISH unit system and volume, volume per volume, or volume per time quantity; one of 'PV', 'OVER PV', 'FVF', 'GOR', 'surface gas rate', or 'saturation'; see notes regarding FVF, also regarding flow rates

Returns

str – the RESQML uom string for the units required by Nexus

Notes

transmissibility not yet catered for here, as RESQML has transmissibility units without a viscosity component; Nexus volume unit expectations vary depending on the data being handled, and sometimes also where in the Nexus input dataset the data is being entered; resqpy.weights_and_measures.valid_quantities() and valid_uoms() may also be of interest; in the ENHLISH unit system, Nexus expacts gas formation volume factors in bbl / 1000 ft3 but that is not a valid RESQML uom – this function will return bbl/bbl for ENGLISH FVF; also be wary of pore volume units when using the medieval ENGLISH unit system: the OVER keyword expects different units than GRID or recurrent override input; ENGLISH fluid flow rates will be returned as bbl/d unless the flavour is specified as 'surface gas rate'

7.16.10 resqpy.weights_and_measures.rq_length_unit

resqpy.weights_and_measures.**rq_length_unit**(*units*)

Returns length units string as expected by resqml.

7.16.11 resqpy.weights_and_measures.rq_time_unit

resqpy.weights_and_measures.**rq_time_unit**(*units*)

Returns time units string as expected by resqml.

7.16.12 resqpy.weights_and_measures.rq_uom

resqpy.weights_and_measures.**rq_uom**(*units*, *quantity=None*)

Returns RESQML uom string equivalent to units.

Parameters

- **units** (*str*) – unit to coerce
- **quantity** (*str, optional*) – if given, raise an exception if the uom is not supported for this quantity

Returns

str – unit of measure

Raises

InvalidUnitError – if units cannot be coerced into RESQML units for the given quantity

7.16.13 resqpy.weights_and_measures.rq_uom_list

resqpy.weights_and_measures.**rq_uom_list**(*units_list*)

Returns a list of RESQML uom equivalents for units in list.

7.16.14 resqpy.weights_and_measures.valid_property_kinds

resqpy.weights_and_measures.**valid_property_kinds**()

Return set of valid property kinds.

7.16.15 resqpy.weights_and_measures.valid_quantities

resqpy.weights_and_measures.**valid_quantities**(*return_attributes=False*)

Return set of valid RESQML quantities.

Parameters

return_attributes (*bool*) – If True, return a dict of all quantities and their attributes, such as the supported units of measure. Else, simply return the set of valid properties.

returns

set or dict

7.16.16 resqpy.weights_and_measures.valid_uoms

resqpy.weights_and_measures.**valid_uoms**(*quantity=None*, *return_attributes=False*)

Return set of valid RESQML units of measure.

Parameters

- **quantity** (*str*) – If given, filter to uoms supported by this quanitity.
- **return_attributes** (*bool*) – If True, return a dict of all uoms and their attributes, such as the full name and dimension. Else, simply return the set of valid uoms.

returns

set or dict

resqpy.weights_and_measures.nexus_units	Functions specific to Nexus units of measure.
resqpy.weights_and_measures.weights_and_measures	Units of measure.

7.16.17 resqpy.weights_and_measures.nexus_units

Functions specific to Nexus units of measure.

7.16.18 resqpy.weights_and_measures.weights_and_measures

Units of measure.

7.17 resqpy.well

Classes relating to wells.

Classes

BlockedWell	Class for RESQML Blocked Wellbore Representation (Wells), ie cells visited by wellbore.
DeviationSurvey	Class for RESQML wellbore deviation survey.
MdDatum	Class for RESQML measured depth datum.
Trajectory	Class for RESQML Wellbore Trajectory Representation (Geometry).
WellboreFrame	Class for RESQML WellboreFrameRepresentation objects (supporting well log Properties)
WellboreMarker	Class to handle RESQML WellboreMarker objects.
WellboreMarkerFrame	Class to handle RESQML WellBoreMarkerFrameRepresentation objects.

7.17.1 resqpy.well.WellboreMarker

class resqpy.well.**WellboreMarker**(*parent_model*, *parent_frame*, *marker_index*, *marker_node=None*, *marker_type=None*, *interpretation_uuid=None*, *title=None*, *originator=None*, *extra_metadata=None*)

Bases: object

Class to handle RESQML WellboreMarker objects.

Note: wellbore markers are not high level RESQML objects

Public Data Attributes:

resqml_type
boundary_feature_dict

Methods:

__init__(parent_model, parent_frame, ...[, ...])	Creates a new wellbore marker object and loads it from xml or populates it from arguments.
create_xml(parent_node[, title])	Creates the xml tree for this wellbore marker.

__init__(*parent_model*, *parent_frame*, *marker_index*, *marker_node=None*, *marker_type=None*, *interpretation_uuid=None*, *title=None*, *originator=None*, *extra_metadata=None*)

Creates a new wellbore marker object and loads it from xml or populates it from arguments.

Parameters

- **parent_model** (*model.Model object*) – the model which the new wellbore marker belongs to
- **parent_frame** (*wellbore_marker_frame.WellboreMarkerFramer object*) – the wellbore marker frame to which the wellbore marker belongs
- **marker_index** (*int*) – index of the wellbore marker in the parent WellboreMarkerFrame object
- **marker_node** (*xml node, optional*) – if given, loads from xml. Else, creates new
- **marker_type** (*str, optional*) – the type of geologic, fluid or contact feature e.g. "fault", "geobody", "horizon ", "gas/oil/water down to", "gas/oil/water up to", "free water contact", "gas oil contact", "gas water contact", "water oil contact", "seal"; ignored if marker_node is present
- **interpretation_uuid** (*uuid.UUID or string, optional*) – uuid of the boundary feature interpretation organizational object that the marker refers to; ignored if marker_node is present
- **title** (*str, optional*) – the citation title to use for a new wellbore marker; ignored if marker_node is present
- **originator** (*str, optional*) – the name of the person creating the wellbore marker, defaults to login id; ignored if uuid is not None; ignored if marker_node is present
- **extra_metadata** (*dict, optional*) – string key, value pairs to add as extra metadata for the wellbore marker; ignored if uuid is not None; ignored if marker_node is present

Returns

the newly created wellbore marker object

Note: it is highly recommended that a related boundary feature interpretation uuid is provided

---

create_xml(*parent_node*, *title='wellbore marker'*)

Creates the xml tree for this wellbore marker.

Parameters

- **parent_node** (*xml node*) – the root node of the WellboreMarkerFrame object to which the newly created node will be appended
- **title** (*string, default "wellbore marker"*) – the citation title of the newly created node; only used if self.title is None

Returns

the newly created xml node

Functions

add_blocked_wells_from_wellspec	Add a blocked well for each well in a Nexus WELLSPEC file.
add_las_to_trajectory	Creates a WellLogCollection and WellboreFrame from a LAS file.
add_logs_from_cellio	Creates a WellIntervalPropertyCollection for a given BlockedWell, using a given cell I/O file.
add_wells_from_ascii_file	Creates new md datum, trajectory, interpretation and feature objects for each well in an ascii file.
lookup_from_cellio	Create a StringLookup Object from a cell I/O row containing a categorical column name and details.
well_name	Returns the 'best' citation title from the object or related well objects.

7.17.2 resqpy.well.add_blocked_wells_from_wellspec

resqpy.well.**add_blocked_wells_from_wellspec**(*model*, *grid*, *wellspec_file*, *usa_date_format=False*)

Add a blocked well for each well in a Nexus WELLSPEC file.

Parameters

- **model** (*model.Model object*) – model to which blocked wells are added
- **grid** (*grid.Grid object*) – grid against which wellspec data will be interpreted
- **wellspec_file** (*string*) – path of ascii file holding Nexus WELLSPEC keyword and data
- **usa_date_format** (*bool*) – mm/dd/yyyy (True) vs. dd/mm/yyyy (False)

Returns

int – count of number of blocked wells created

Notes

this function appends to the hdf5 file and creates xml for the blocked wells (but does not store epc); 'simulation' trajectory and measured depth datum objects will also be created

7.17.3 resqpy.well.add_las_to_trajectory

resqpy.well.**add_las_to_trajectory**(*las: LASFile*, *trajectory*, *realization=None*, *check_well_name=False*)

Creates a WellLogCollection and WellboreFrame from a LAS file.

Parameters

- **las** – an lasio.LASFile object
- **trajectory** – an instance of resqpy.well.Trajectory .
- **realization** (*integer*) – if present, the single realisation (within an ensemble) that this collection is for
- **check_well_name** (*bool*) – if True, raise warning if LAS well name does not match existing wellborefeature citation title

Returns

collection, well_frame –

instances of resqpy.property.WellLogCollection
and resqpy.well.WellboreFrame

Note: in this current implementation, the first curve in the las object must be Measured Depths, not e.g. TVDSS

7.17.4 resqpy.well.add_logs_from_cellio

resqpy.well.**add_logs_from_cellio**(*blockedwell*, *cellio*)

Creates a WellIntervalPropertyCollection for a given BlockedWell, using a given cell I/O file.

Parameters

- **blockedwell** – a resqml blockedwell object
- **cellio** – an ascii file exported from RMS containing blocked well geometry and logs; must contain columns i_index, j_index and k_index, plus additional columns for logs to be imported

7.17.5 resqpy.well.add_wells_from_ascii_file

resqpy.well.**add_wells_from_ascii_file**(*model*, *crs_uuid*, *trajectory_file*, *comment_character='#'*, *space_separated_instead_of_csv=False*, *well_col='WELL'*, *md_col='MD'*, *x_col='X'*, *y_col='Y'*, *z_col='Z'*, *length_uom='m'*, *md_domain=None*, *drilled=False*)

Creates new md datum, trajectory, interpretation and feature objects for each well in an ascii file.

Parameters

- **crs_uuid** (*uuid.UUID*) – the unique identifier of the coordinate reference system applicable to the x,y,z data; if None, a default crs will be created, making use of the length_uom and z_inc_down arguments
- **trajectory_file** (*string*) – the path of the ascii file holding the well trajectory data to be loaded
- **comment_character** (*string, default '#'*) – character deemed to introduce a comment in the trajectory file
- **space_separated_instead_of_csv** (*boolean, default False*) – if True, the columns in the trajectory file are space separated; if False, comma separated
- **well_col** (*string, default 'WELL'*) – the heading for the column containing well names
- **md_col** (*string, default 'MD'*) – the heading for the column containing measured depths
- **x_col** (*string, default 'X'*) – the heading for the column containing X (usually easting) data
- **y_col** (*string, default 'Y'*) – the heading for the column containing Y (usually northing) data
- **z_col** (*string, default 'Z'*) – the heading for the column containing Z (depth or elevation) data

- **length_uom** (*string, default 'm'*) – the units of measure for the measured depths; should be 'm' or 'ft'
- **md_domain** (*string, optional*) – the source of the original deviation data; may be 'logger' or 'driller'
- **drilled** (*boolean, default False*) – True should be used for wells that have been drilled; False otherwise (planned, proposed, or a location being studied)
- **z_inc_down** (*boolean, default True*) – indicates whether z values increase with depth; only used in the creation of a default coordinate reference system; ignored if crs_uuid is not None

Returns

tuple of lists of objects – (feature_list, interpretation_list, trajectory_list, md_datum_list)

Notes

ascii file must be table with first line being column headers, with columns for WELL, MD, X, Y & Z; actual column names can be set with optional arguments; all the objects are added to the model, with array data being written to the hdf5 file for the trajectories; the md_domain and drilled values are stored in the RESQML metadata but are only for human information and do not generally affect computations

7.17.6 resqpy.well.lookup_from_cellio

resqpy.well.**lookup_from_cellio**(*line*, *model*)

Create a StringLookup Object from a cell I/O row containing a categorical column name and details.

Parameters

- **line** – a string from a cell I/O file, containing the column (log) name, type and categorical information
- **model** – the model to add the StringTableLookup to

Returns

uuid – the uuid of a StringTableLookup, either for a newly created table, or for an existing table if an identical one exists

7.17.7 resqpy.well.well_name

resqpy.well.**well_name**(*well_object*, *model=None*)

Returns the 'best' citation title from the object or related well objects.

Parameters

- **well_object** (*object, uuid or root*) – Object for which a well name is required. Can be a Trajectory, WellboreInterpretation, WellboreFeature, BlockedWell, WellboreMarkerFrame, WellboreFrame, DeviationSurvey or MdDatum object
- **model** (model.Model, *optional*) – required if passing a uuid or root; not recommended otherwise

Returns

string being the 'best' citation title to serve as a well name, form the object or some related objects

Note: xml and relationships must be established for this function to work

resqpy.well.blocked_well_frame	Module holding functions relating to the mapping of welbore frame properties onto blocked wells.
resqpy.well.well_object_funcs	well_object_funcs.py: resqpy well module for functions that impact well objects
resqpy.well.well_utils	well_utils.py: functions used by the classes in resqpy.well

7.17.8 resqpy.well.blocked_well_frame

Module holding functions relating to the mapping of welbore frame properties onto blocked wells.

Functions

add_blocked_well_properties_from_wellbore_frame	Add properties to this blocked well by derivation from wellbore frame intervals properties.
blocked_well_frame_contributions_list	Returns wellbore frame contributions to each cell of a blocked well.

resqpy.well.blocked_well_frame.add_blocked_well_properties_from_wellbore_frame

resqpy.well.blocked_well_frame.**add_blocked_well_properties_from_wellbore_frame**(*bw*, *frame_uuid=None*, *property_kinds_list=None*, *realization=None*, *set_length=None*, *set_perforation_fraction=None*, *set_frame_interval=False*)

Add properties to this blocked well by derivation from wellbore frame intervals properties.

Parameters

- **bw** (*BlockedWell*) – the blocked well to add properties to
- **frame_uuid** (*UUID, optional*) – the uuid of the wellbore frame to source properties from; if None, a solitary wellbore frame relating to the same trajectory as the blocked well will be used
- **property_kinds_list** (*list of str, optional*) – if present, a list of handled property kinds which are to be set from the wellbore frame properties; if None, any handled property kinds that are present for the wellbore frame will be used
- **realization** (*int, optional*) – if present, wellbore frame properties will be filtered by this realization number and it will be assigned to the blocked well properties that are created
- **set_length** (*bool, optional*) – if True, a length property will be generated based on active measured depth intervals; if None, will be set True if length is in list of property kinds being processed

- **set_perforation_fraction** (*bool, optional*) – if True, a perforation fraction property will be created based on the fraction of the measured depth within a blocked well cell that is flagged as active, ie. perforated at some time; if None, it will be created only if length and permeability thickness are both absent
- **set_frame_interval** (*bool, default False*) – if True, a static discrete property holding the index of the dominant active wellbore frame interval (per blocked well cell) is created

Returns
list of uuids of created property parts (does not include any copied time series object)

Notes

this method is designed to set up some blocked well properties based on similar properties already established on a special wellbore frame, mainly for perforations and open hole completions; frame_uuid should be specified if there are well logs in the dataset, or other wellbore frames; if a permeability thickness property is being set based on a wellbore frame property, the value is divided between blocked well cells based solely on measured depth interval lengths, without reference to grid properties such as net to gross ratio or permeability; titles will be the same as those used in the frame properties, and 'PPERF' for partial perforation; if set_frame_interval is True, the resulting property will be given a soft relationship with the wellbore frame (in addition to its supporting representation reference relationship with the blocked well); a null value of -1 is used where no active frame interval is present in a cell; units of measure will also be the same as those in the wellbore frame; this method only supports single grid blocked wells at present; blocked well and wellbore frame must be in the same model

resqpy.well.blocked_well_frame.blocked_well_frame_contributions_list

resqpy.well.blocked_well_frame.**blocked_well_frame_contributions_list**(*bw*, *wbf*)

Returns wellbore frame contributions to each cell of a blocked well.

Parameters

- **bw** (*BlockedWell*) – the blocked well to map the wellbore frame onto
- **wbf** (*WellboreFrame*) – the wellbore frame to map to the blocked well

Returns

list of list of (int, float, float) with one entry per blocked well cell, and each

entry being a list of (wellbore frame interval index,
fraction of wellbore frame interval in cell, fraction of cell's wellbore interval in wellbore frame interval)

7.17.9 resqpy.well.well_object_funcs

well_object_funcs.py: resqpy well module for functions that impact well objects

7.17.10 resqpy.well.well_utils

well_utils.py: functions used by the classes in resqpy.well

Functions

extract_xyz	Extracts an x,y,z coordinate from a solitary point xml node.
find_entry_and_exit	Returns (entry_axis, entry_polarity, entry_xyz, exit_axis, exit_polarity, exit_xyz).
load_hdf5_array	Loads the property array data as an attribute of object, from the hdf5 referenced in xml node.
well_names_in_cellio_file	Returns a list of well names as found in the RMS blocked well export cell I/O file.

resqpy.well.well_utils.extract_xyz

resqpy.well.well_utils.**extract_xyz**(*xyz_node*)

Extracts an x,y,z coordinate from a solitary point xml node.

argument:

xyz_node: the xml node representing the solitary point (in 3D space)

Returns

triple float – (x, y, z) coordinates as a tuple

resqpy.well.well_utils.find_entry_and_exit

resqpy.well.well_utils.**find_entry_and_exit**(*cp*, *entry_vector*, *exit_vector*, *well_name*)

Returns (entry_axis, entry_polarity, entry_xyz, exit_axis, exit_polarity, exit_xyz).

resqpy.well.well_utils.load_hdf5_array

resqpy.well.well_utils.**load_hdf5_array**(*object*, *node*, *array_attribute*, *tag='Values'*, *dtype='float'*, *model=None*)

Loads the property array data as an attribute of object, from the hdf5 referenced in xml node.

resqpy.well.well_utils.well_names_in_cellio_file

resqpy.well.well_utils.**well_names_in_cellio_file**(*cellio_file*)

Returns a list of well names as found in the RMS blocked well export cell I/O file.

7.18 resqpy.olio

Low level supporting modules, mostly providing functions rather than classes.

resqpy.olio.ab_toolbox	Small utility functions related to use of pure binary files.
resqpy.olio.base	Base class for generic resqml objects.
resqpy.olio.box_utilities	Simple functions relating to cartesian grid boxes.
resqpy.olio.class_dict	A simple dictionary mapping resqml class names to more readable names.
resqpy.olio.consolidation	Support for consolidation of datasets based on equivalence between parts.
resqpy.olio.dataframe	Classes for storing and retrieving dataframes as RESQML objects.
resqpy.olio.exceptions	Custom exceptions used in resqpy.
resqpy.olio.factors	Factorization and functions supporting grid extent determination from corner points.
resqpy.olio.fine_coarse	fine_coarse.py: Module providing support for grid refinement and coarsening.
resqpy.olio.grid_functions	Miscellaneous functions relating to grids.
resqpy.olio.intersection	intersection.py: functions to test whether lines intersect with planes.
resqpy.olio.keyword_files	Basic functions for searching for keywords in an ascii control file such as a nexus deck.
resqpy.olio.load_data	Functions to load data from various ASCII simulator file formats.
resqpy.olio.point_inclusion	point_inclusion.py: functions to test whether a point is within a polygon; also line intersection with planes.
resqpy.olio.random_seed	Module providing wrapper for random number generator seeding functions.
resqpy.olio.read_nexus_fault	read_nexus_fault.py: functions for reading Nexus fault definition data from an ascii file.
resqpy.olio.relperm	relperm.py: class for dataframes of relative permeability data as RESQML objects.
resqpy.olio.simple_lines	simple_lines.py: functions for handling simple lines in relation to a resqml grid.
resqpy.olio.time	time.py: A very thin wrapper around python datetime functionality, to meet resqml standard.
resqpy.olio.trademark	trademark.py module for mentioning trademarks in diagnostic log.
resqpy.olio.transmission	Transmissibility functions for grids.
resqpy.olio.triangulation	triangulation.py: functions for finding Delaunay triangulation and Voronoi graph from a set of points.
resqpy.olio.uuid	uuid.py: Thin wrapper around python uuid (universally unique identifier) module.

continues on next page

Table 4 – continued from previous page

resqpy.olio.vdb	vdb.py: Module providing functions for reading from VDB datasets.
resqpy.olio.vector_utilities	Utilities for working with 3D vectors in cartesian space.
resqpy.olio.volume	volume.py: Functions to calculate volumes of hexahedral cells; assumes consistent length units.
resqpy.olio.wellspec_keywords	Module for loading WELLSPEC files.
resqpy.olio.write_data	Array writing functions.
resqpy.olio.write_hdf5	write_hdf5.py: Class to write a resqml hdf5 file and functions for copying hdf5 data.
resqpy.olio.xml_et	xml_et.py: Resqml xml element tree utilities module.
resqpy.olio.xml_namespaces	xml_namespaces.py: Module defining constant resqml xml namespaces.
resqpy.olio.zmap_reader	Functions for reading zmap and roxar format files.

7.18.1 resqpy.olio.ab_toolbox

Small utility functions related to use of pure binary files.

Functions

binary_file_extension_and_np_type_for_data_type	Returns a file extension suitable for a pure binary array (ab) file of given data type.
cp_binary_filename	Returns a version of the file name with extension adjusted to indicate reseq order and pure binary.
load_array_from_ab_file	Loads a pure binary file into a numpy array, optionally converting to 64 bit.

resqpy.olio.ab_toolbox.binary_file_extension_and_np_type_for_data_type

resqpy.olio.ab_toolbox.**binary_file_extension_and_np_type_for_data_type**(*data_type: str*) → Optional[Tuple[str, object]]

Returns a file extension suitable for a pure binary array (ab) file of given data type.

resqpy.olio.ab_toolbox.cp_binary_filename

resqpy.olio.ab_toolbox.**cp_binary_filename**(*file_name*, *nexus_ordering=True*)

Returns a version of the file name with extension adjusted to indicate reseq order and pure binary.

resqpy.olio.ab_toolbox.load_array_from_ab_file

resqpy.olio.ab_toolbox.**load_array_from_ab_file**(*file_name*, *shape*, *return_64_bit=False*)

Loads a pure binary file into a numpy array, optionally converting to 64 bit.

7.18.2 resqpy.olio.base

Base class for generic resqml objects.

Classes

BaseResqpy	Base class for generic resqpy classes.

resqpy.olio.base.BaseResqpy

class resqpy.olio.base.**BaseResqpy**(*model*, *uuid=None*, *title=None*, *originator=None*, *extra_metadata=None*)

Bases: object

Base class for generic resqpy classes.

Implements generic attributes such as uuid, root, part, title, originator.

Implements generic magic methods, such as pretty printing and testing for equality.

Example use:

```
class AnotherResqpyObject(BaseResqpy):

    resqml_type = 'obj_anotherresqmlobjectrepresentation'
```

Public Data Attributes:

resqml_type	Definition of which RESQML object the class represents.
part	Standard part name corresponding to self.uuid.
root	XML node corresponding to self.uuid.
citation_title	Citation block title equivalent to self.title.

Methods:

__init__(model[, uuid, title, originator, ...])	Load an existing resqml object, or create new.
try_reuse()	Look for an equivalent existing RESQML object and modify the uuid of this object if found.
create_xml([title, originator, ...])	Write citation block to XML.
append_extra_metadata(meta_dict)	Append a given dictionary of metadata to the existing metadata.
__eq__(other)	Implements equals operator; uses is_equivalent() otherwise compares class type and uuid.
__ne__(other)	Implements not equal operator.
__repr__()	String representation.

abstract property resqml_type

Definition of which RESQML object the class represents.

Subclasses must overwrite this abstract attribute.

__init__(*model*, *uuid=None*, *title=None*, *originator=None*, *extra_metadata=None*)

Load an existing resqml object, or create new.

Parameters

- **model** (*resqpy.model.Model*) – Parent model
- **uuid** (*str, optional*) – Load from existing uuid (if given), else create new.
- **title** (*str, optional*) – Citation title
- **originator** (*str, optional*) – Creator of object. By default, uses user id.

uuid

Unique identifier

title

Citation title

originator

Creator of object. By default, user id.

property part

Standard part name corresponding to self.uuid.

property root

XML node corresponding to self.uuid.

property citation_title

Citation block title equivalent to self.title.

try_reuse()

Look for an equivalent existing RESQML object and modify the uuid of this object if found.

Returns

boolean – True if an equivalent object was found, False if not

Note: by design this method may change this object's uuid as a side effect

create_xml(*title=None*, *originator=None*, *extra_metadata=None*, *add_as_part=False*)

Write citation block to XML.

Note: *add_as_part* is False by default in this base method. Derived classes should typically extend this method to complete the XML representation, and then finally ensure the node is added as a part to the model.

Parameters

- **title** (*string*) – used as the citation Title text
- **originator** (*string, optional*) – the name of the human being who created the deviation survey part; default is to use the login name
- **extra_metadata** (*dict, optional*) – extra metadata items to be added
- **add_as_part** (*boolean*) – if True, the newly created xml node is added as a part in the model

Returns

node – the newly created root node

append_extra_metadata(*meta_dict*)

Append a given dictionary of metadata to the existing metadata.

7.18.3 resqpy.olio.box_utilities

Simple functions relating to cartesian grid boxes.

A box is a logical cuboid subset of the cells of a cartesian grid. A box is defined by a small numpy array: [[min_k, min_j, min_i], [max_k, max_j, max_i]]. The cells identified by the max indices are included in the box (not following the python convention) The ordering of the i,j & k indices might be reversed - identifier names then have a suffix of _ijk instead of _kji. The indices can be in simulator convention, starting at 1, or python convention, starting at 0, indicated by suffix of 0 or 1

Functions

box_kji0_from_words_iijjkk1	Returns an integer array of extent [2, 3] converted from a list of words representing logical box.
boxes_overlap	Returns True if the two boxes have any overlap in 3D, otherwise False.
cell_in_box	Returns True if cell is within box, otherwise False.
central_cell	Returns the indices of the cell at the centre of the box.
extent_of_box	Returns a 3 integer numpy array holding the size of the box, with the same ordering as the box.
full_extent_box0	Returns a box containing all the cells in a grid of the given extent.
local_box_cell_from_parent_cell	Given a cell index triplet in the host grid, and a box, returns the equivalent local cell index triplet.
overlapping_boxes	Checks for 3D overlap of two boxes; returns True and sets trim_box if there is overlap, otherwise False.
parent_cell_from_local_box_cell	Given a box and a local cell index triplet, converts to the equivalent cell index triplet in the host grid.
single_cell_box	Returns a box containing the single given cell; protocol for box matches that of cell.
spaced_string_iijjkk1_for_box_kji0	Returns a string representing the box space in simulator input format, eg.
string_iijjkk1_for_box_kji0	Returns a string representing the box space in simulator protocol, eg.
trim_box_by_box_returning_new_mask	Reduces box_to_be_trimmed by trim_box; trim_box must be a neat subset box at one face of box_to_be_trimmed.
trim_box_to_mask_returning_new_mask	Reduce the coverage of bounding box to the minimum needed to contain True elements of mask.
union	Returns the box which contains both box_1 and box_2.
valid_box	Returns True if the entire box is within a grid of size host_extent.
volume_of_box	Returns the number of cells in the logical 3D cell space defined by box.

resqpy.olio.box_utilities.box_kji0_from_words_iijjkk1

resqpy.olio.box_utilities.**box_kji0_from_words_iijjkk1**(*words*)

Returns an integer array of extent [2, 3] converted from a list of words representing logical box.

input argument (unmodified):

words: a list of strings with at least 6 elements castable to int
[min_i, max_i, min_j, max_j, min_k, max_k] in Fortran/simulator protocol (indices start at 1)

returns: 2D numpy int array of shape (2, 3)
[min, max][k, j, i] with cell indices in python protocol (zero base)

Notes

designed to take string format numbers: minI maxI minJ maxJ minK maxK and convert to a pair of integer cell id triplets: min(k, j, i), max(k, j, i) NB: output indices have been decremented by 1 (for python indexing starting at zero)

resqpy.olio.box_utilities.boxes_overlap

resqpy.olio.box_utilities.**boxes_overlap**(*box_a*, *box_b*)

Returns True if the two boxes have any overlap in 3D, otherwise False.

Parameters

- **box_a** – numpy int or float array of shape (2, 3)
- **box_b** – numpy int or float array of shape (2, 3)

if int arrays, each is lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid protocol of indices for the two boxes must be the same if float arrays, each is min & max x,y,z triplets

returns: boolean
True if box_a and box_b overlap, False otherwise

resqpy.olio.box_utilities.cell_in_box

resqpy.olio.box_utilities.**cell_in_box**(*cell*, *box*)

Returns True if cell is within box, otherwise False.

input arguments (unmodified):

cell: numpy int array of shape (3)
index of a cell in a 3D cartesian grid, in the same protocol as box (usually python protocol kji, zero base)

box: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid in the same protocol as cell

returns: boolean
True if cell is within box, False otherwise

resqpy.olio.box_utilities.central_cell

resqpy.olio.box_utilities.**central_cell**(*box*)

Returns the indices of the cell at the centre of the box.

resqpy.olio.box_utilities.extent_of_box

resqpy.olio.box_utilities.**extent_of_box**(*box*)

Returns a 3 integer numpy array holding the size of the box, with the same ordering as the box.

input argument (unmodified):

box: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid

returns: numpy int array of shape (3)
the extent (shape) of the cuboid defined by box

resqpy.olio.box_utilities.full_extent_box0

resqpy.olio.box_utilities.**full_extent_box0**(*extent*)

Returns a box containing all the cells in a grid of the given extent.

input argument (unmodified):

extent: numpy int array of shape (3)
extent (shape) of a 3D cartesian grid, usually in kji python protocol

returns: numpy int array of shape (2, 3)
indices defining a maximal box containing the entire grid; kji ordering is same as that of extent; zero base

resqpy.olio.box_utilities.local_box_cell_from_parent_cell

resqpy.olio.box_utilities.**local_box_cell_from_parent_cell**(*box*, *parent_cell*, *based_0_or_1=0*)

Given a cell index triplet in the host grid, and a box, returns the equivalent local cell index triplet.

input arguments (unmodified):

box: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid indices in the same ordering as parent_cell; start value (python or Fortran/simulator) given by based_0_or_1

parent_cell: numpy int array of shape (3)
indices of a cell within host grid indices in the same ordering as box; start value (python or Fortran/simulator) given by based_0_or_1

based_0_or_1: int, value 0 or 1
start value (base) for indices of box and parent_cell arguments, and of return value

returns: numpy int array of shape (3); or None
indices defining the parent_cell in coords local to box, if the cell is within the box if parent_cell is not within box, None is returned

resqpy.olio.box_utilities.overlapping_boxes

resqpy.olio.box_utilities.**overlapping_boxes**(*established_box*, *new_box*, *trim_box*)

Checks for 3D overlap of two boxes; returns True and sets trim_box if there is overlap, otherwise False.

trim_box is modified in place.

Parameters

- **established_box** – numpy int array of shape (2, 3)
- **new_box** – numpy int array of shape (2, 3) each is lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid protocol of indices for the two boxes must be the same
- **trim_box** – numpy int array of shape (2, 3) set to lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid a subset of new_box such that if removed from new_box, a valid box would remain with no overlap with established_box indices protocol is the same as that used for established_box and new_box (if return value is True) if there is no overlap (return value False), all elements of trim_box are set to 0

Note: when there is overlap between the boxes, there can be more than one way to trim the new_box, with trim_box fully covering either ij, jk or ik planes of new_box the function selects the trim_box containing the minimum number of cells (minimum 'loss' to trimming) this function does not actually apply the trimming, ie. new_box is not modified here

---

returns: boolean
True if established_box and new_box overlap (implies trim_box valid), False otherwise (trim_box elements all 0)

resqpy.olio.box_utilities.parent_cell_from_local_box_cell

resqpy.olio.box_utilities.**parent_cell_from_local_box_cell**(*box*, *box_cell*, *based_0_or_1=0*)

Given a box and a local cell index triplet, converts to the equivalent cell index triplet in the host grid.

input arguments (unmodified):

box: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid indices in the same ordering as box_cell; start value (python or Fortran/simulator) given by based_0_or_1

box_cell: numpy int array of shape (3)
indices of a cell within box, in coords local to box indices in the same ordering as box; start value (python or Fortran/simulator) given by based_0_or_1

based_0_or_1: int, value 0 or 1
start value (base) for indices of box and box_cell arguments, and of return value

returns: numpy int array of shape (3)
indices defining the cell in the host grid space equivalent to box_cell ordering of indices is same as that of box and box_cell; base is given by based_0_or_1 argument

resqpy.olio.box_utilities.single_cell_box

resqpy.olio.box_utilities.**single_cell_box**(*cell*)

Returns a box containing the single given cell; protocol for box matches that of cell.

input argument (unmodified):

cell: numpy int array of shape (3)
indices of a cell within a 3D cartesian grid, usually in python protocol (kji ordering, zero base)

returns: numpy int array of shape (2, 3)
indices defining a minimal box containing a single cell; protocol is same as that of cell

resqpy.olio.box_utilities.spaced_string_iijjkk1_for_box_kji0

resqpy.olio.box_utilities.**spaced_string_iijjkk1_for_box_kji0**(*box_kji0*, *colon_separator=' '*)

Returns a string representing the box space in simulator input format, eg. '1 5 3 20 100 103'.

input arguments (unmodified):

box_kji0: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid with python kji ordering and zero start for indices

colon_separator: string (typically ':' or ' ')
the character(s) included in the return string between lower and upper bounds in each direction

returns: string
ascii representation of box in Fortran/simulator ijk protocol starting 1, suitable for use in include files

resqpy.olio.box_utilities.string_iijjkk1_for_box_kji0

resqpy.olio.box_utilities.**string_iijjkk1_for_box_kji0**(*box_kji0*)

Returns a string representing the box space in simulator protocol, eg. '[1:5, 3:20, 100:103]'.

input argument (unmodified):

box_kji0: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid with python kji ordering and zero start for indices

returns: string
human readable representation of box in Fortran/simulator ijk protocol starting 1

resqpy.olio.box_utilities.trim_box_by_box_returning_new_mask

resqpy.olio.box_utilities.**trim_box_by_box_returning_new_mask**(*box_to_be_trimmed*, *trim_box*, *mask_kji0*)

Reduces box_to_be_trimmed by trim_box; trim_box must be a neat subset box at one face of box_to_be_trimmed.

input/output argument (modified):

box_to_be_trimmed: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid indices protocol is python kji ordering with zero base modified to exclude space occupied by trim_box

input arguments (unmodified):

trim_box: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid indices protocol is python kji ordering with zero base the volume to be removed from box_to_be_trimmed, must be a subset of box_to_be_trimmed completely covering one face of box_to_be_trimmed (thereby ensuring that after trimming, a valid cuboid box results)

mask_kji0: numpy 3D boolean array of shape matching extent of input box_to_be_trimmed
indices protocol is python kji ordering with zero base

returns: numpy 3D boolean array of shape matching extent of output box_to_be_trimmed
the return array is a version of mask_kji0 that has been trimmed in accordance with the box trimming

resqpy.olio.box_utilities.trim_box_to_mask_returning_new_mask

resqpy.olio.box_utilities.**trim_box_to_mask_returning_new_mask**(*bounding_box_kji0*, *mask_kji0*)

Reduce the coverage of bounding box to the minimum needed to contain True elements of mask.

Returns trimmed mask.

resqpy.olio.box_utilities.union

resqpy.olio.box_utilities.**union**(*box_1*, *box_2*)

Returns the box which contains both box_1 and box_2.

resqpy.olio.box_utilities.valid_box

resqpy.olio.box_utilities.**valid_box**(*box*, *host_extent*)

Returns True if the entire box is within a grid of size host_extent.

input arguments (unmodified):

box: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid in python protocol of zero base, kji (normally) or ijk ordering same as for host_extent

host_extent: triple int
the extent (shape) of a 3D cartesian grid

returns: boolean
True if box is a valid box within a grid of shape host_extent, False otherwise

resqpy.olio.box_utilities.volume_of_box

resqpy.olio.box_utilities.**volume_of_box**(*box*)

Returns the number of cells in the logical 3D cell space defined by box.

input argument (unmodified):

box: numpy int array of shape (2, 3)
lower & upper indices in 3 dimensions defining a logical cuboid subset of a 3D cartesian grid

returns: int
the total number of cells in box

7.18.4 resqpy.olio.class_dict

A simple dictionary mapping resqml class names to more readable names.

Functions

readable_class	Given a resqml object class name as a string, returns a more human readable string.

resqpy.olio.class_dict.readable_class

resqpy.olio.class_dict.**readable_class**(*class_name*)

Given a resqml object class name as a string, returns a more human readable string.

argument:

class_name (string): the resqml class name, eg. 'obj_IjkGridRepresentation'

Returns

a human readable version of the class name, eg. 'Grid (IJK)'

7.18.5 resqpy.olio.consolidation

Support for consolidation of datasets based on equivalence between parts.

Classes

Consolidation	Class supporting equivalence mapping of high level RESQML parts between models.

resqpy.olio.consolidation.Consolidation

class resqpy.olio.consolidation.**Consolidation**(*resident_model*)

Bases: object

Class supporting equivalence mapping of high level RESQML parts between models.

Methods:

__init__(resident_model)	Initialise a new Consolidation object prior to merging parts from another model.
equivalent_uuid_for_part(part[, ...])	Returns uuid of an equivalent part in resident model, or None if no equivalent found.
equivalent_uuid_int_for_part(part[, ...])	Returns uuid.int of an equivalent part in resident model, or None if no equivalent found.
force_uuid_equivalence(immigrant_uuid, ...)	Forces immigrant object to be treated as equivalent to (same as) resident object, identified by uuids.
force_uuid_int_equivalence(...)	Forces immigrant object to be treated as equivalent to (same as) resident object, identified by uuid ints.
force_part_equivalence(immigrant_part, ...)	Forces immigrant part to be treated as equivalent to resident part.
check_map_integrity()	Raises assertion failure if map contains any potentially circular references.

__init__(*resident_model*)

Initialise a new Consolidation object prior to merging parts from another model.

Parameters
resident_model (model.Model) – the model into which potentially equivalent parts will be merged

Returns
the new Consolidation object

equivalent_uuid_for_part(*part*, *immigrant_model=None*, *ignore_identical_part=False*)

Returns uuid of an equivalent part in resident model, or None if no equivalent found.

equivalent_uuid_int_for_part(*part*, *immigrant_model=None*, *ignore_identical_part=False*)

Returns uuid.int of an equivalent part in resident model, or None if no equivalent found.

force_uuid_equivalence(*immigrant_uuid*, *resident_uuid*)

Forces immigrant object to be treated as equivalent to (same as) resident object, identified by uuids.

force_uuid_int_equivalence(*immigrant_uuid_int*, *resident_uuid_int*)

Forces immigrant object to be treated as equivalent to (same as) resident object, identified by uuid ints.

force_part_equivalence(*immigrant_part*, *resident_part*)

Forces immigrant part to be treated as equivalent to resident part.

check_map_integrity()

Raises assertion failure if map contains any potentially circular references.

Functions

sort_parts_list	Returns a copy of the parts list sorted into the preferred order for consolidating.
sort_uuids_list	Returns a copy of the uuids list (or uuid ints list) sorted into the preferred order for consolidating.

resqpy.olio.consolidation.sort_parts_list

resqpy.olio.consolidation.**sort_parts_list**(*model*, *parts_list*)

Returns a copy of the parts list sorted into the preferred order for consolidating.

resqpy.olio.consolidation.sort_uuids_list

resqpy.olio.consolidation.**sort_uuids_list**(*model*, *uuids_list*)

Returns a copy of the uuids list (or uuid ints list) sorted into the preferred order for consolidating.

7.18.6 resqpy.olio.dataframe

Classes for storing and retrieving dataframes as RESQML objects.

Note that this module uses the obj_Grid2dRepresentation class in a way that was not envisaged when the RESQML standard was defined; software that does not use resqpy is unlikely to be able to do much with data stored in this way

Classes

DataFrame	Class for storing and retrieving a pandas dataframe of numerical data as a RESQML property.
TimeTable	Class for storing and retrieving a pandas dataframe where rows relate to steps in a time series.

resqpy.olio.dataframe.DataFrame

class resqpy.olio.dataframe.**DataFrame**(*model*, *uuid=None*, *df=None*, *uom_list=None*, *realization=None*, *title='dataframe'*, *column_lookup_uuid=None*, *uom_lookup_uuid=None*, *extra_metadata=None*)

Bases: object

Class for storing and retrieving a pandas dataframe of numerical data as a RESQML property.

Notes

actual values are stored either as z values in a Mesh (Grid2d) object, or as a property on such a mesh when multiple raalizations are in use; a regular Mesh object is created to act as a supporting representation; columns are mapped onto I and rows onto J; if a property is used then the indexable elements are 'nodes'; column titles are stored in a related StringLookup object, indexed by column number; column units are optionally treated in the same way (uom for the property is generally set to Euc); all values are stored as floats; use the derived TimeTable class if rows relate to steps in a TimeSeries; use the derived RelPerm class if the rows relate to relative permeability data

Methods:

__init__(model[, uuid, df, uom_list, ...])	Create a new Dataframe object from either a previously stored property or a pandas dataframe.
dataframe()	Returns the Dataframe as a pandas DataFrame.
column_uom(col_index)	Returns units of measure for the specified column, or Euc if no units present.
write_hdf5_and_create_xml()	Write dataframe data to hdf5 file and create xml for RESQML objects to represent dataframe.

__init__(*model*, *uuid=None*, *df=None*, *uom_list=None*, *realization=None*, *title='dataframe'*, *column_lookup_uuid=None*, *uom_lookup_uuid=None*, *extra_metadata=None*)

Create a new Dataframe object from either a previously stored property or a pandas dataframe.

Parameters

- **model** (*model.Model*) – the model to which the new Dataframe will be attached
- **uuid** (*uuid.UUID, optional*) – the uuid of an existing Grid2dRepresentation object acting as support for a dataframe property (or holding the dataframe as z values)
- **df** (*pandas.DataFrame, optional*) – a dataframe from which the new Dataframe is to be created; if both uuid and df are supplied, realization must not be None and a new realization property will be created
- **uom_list** (*list of str, optional*) – a list holding the units of measure for each column; if present, length of list must match number of columns in df; ignored if uuid is not None
- **realization** (*int, optional*) – if present, the realization number of the RESQML property holding the dataframe
- **title** (*str, default 'dataframe'*) – used as the citation title for the Mesh (and property); ignored if uuid is not None
- **column_lookup_uuid** (*uuid, optional*) – if present, the uuid of a string lookup table holding the column names; if present, the contents and order of the table must match the columns in the dataframe; if absent, a new lookup table will be created
- **uom_lookup_uuid** (*uuid, optional*) – if present, the uuid of a string lookup table holding the units of measure for each column; if None and uom_list is present, a new table will be created; if both uom_list and uom_lookup_uuid are present, their contents must match
- **extra_metadata** (*dict, optional*) – if present, a dictionary of extra metadata items, str: str; ignored if uuid is not None

Returns
a newly created Dataframe object

Notes

when initialising from an existing RESQML object, the supporting mesh and its property should have been originally created using this class; when working with ensembles, each object of this class will only handle the data for one realization, though they may share a common support; if both a uuid and a df are provided, a realization number must also be given and the dataframe is used to create a new realization similar to that identified by the uuid

dataframe()

Returns the Dataframe as a pandas DataFrame.

column_uom(*col_index*)

Returns units of measure for the specified column, or Euc if no units present.

write_hdf5_and_create_xml()

Write dataframe data to hdf5 file and create xml for RESQML objects to represent dataframe.

Functions

dataframe_for_title	Returns a DataFrame object loaded from model, with given title (optionally for given realization).
dataframe_parts_in_model	Returns list of part names within model that are representing DataFrame support objects.
timetable_for_title	Returns a TimeTable object loaded from model, with given title (optionally for given realization).
timetable_parts_in_model	Returns list of part names within model that are representing TimeTable dataframe support objects.

resqpy.olio.dataframe.dataframe_for_title

resqpy.olio.dataframe.**dataframe_for_title**(*model*, *title*, *realization=None*)

Returns a DataFrame object loaded from model, with given title (optionally for given realization).

resqpy.olio.dataframe.dataframe_parts_in_model

resqpy.olio.dataframe.**dataframe_parts_in_model**(*model*, *timetables=None*, *title=None*, *related_uuid=None*)

Returns list of part names within model that are representing DataFrame support objects.

Parameters

- **model** (*model.Model*) – the model to be inspected for dataframes
- **timetables** (*boolean or None*) – if True, only TimeTable dataframe parts will be included; if False only DataFrame parts that are not representing TimeTable objects will be included; if None, both parts for both types of dataframe will be included
- **title** (*str, optional*) – if present, only parts with a citation title exactly matching will be included

- **related_uuid** (*uuid, optional*) – if present, only parts relating to this uuid are included

Returns

list of str, each element in the list is a part name, within model, which is representing the support for a DataFrame object

resqpy.olio.dataframe.timetable_for_title

resqpy.olio.dataframe.**timetable_for_title**(*model*, *title*, *realization=None*)

Returns a TimeTable object loaded from model, with given title (optionally for given realization).

resqpy.olio.dataframe.timetable_parts_in_model

resqpy.olio.dataframe.**timetable_parts_in_model**(*model*, *title=None*, *related_uuid=None*)

Returns list of part names within model that are representing TimeTable dataframe support objects.

Parameters

- **model** (model.Model) – the model to be inspected for dataframes
- **title** (*str, optional*) – if present, only parts with a citation title exactly matching will be included
- **related_uuid** (*uuid, optional*) – if present, only parts relating to this uuid are included

Returns

list of str, each element in the list is a part name, within model, which is representing the support for a TimeTable object

7.18.7 resqpy.olio.exceptions

Custom exceptions used in resqpy.

Exceptions

IncompatibleUnitsError	Raised when two units do not share compatible base units and dimensions.
InvalidUnitError	Raised when a unit cannot be converted into a valid RESQML unit of measure.

resqpy.olio.exceptions.IncompatibleUnitsError

exception resqpy.olio.exceptions.**IncompatibleUnitsError**

Raised when two units do not share compatible base units and dimensions.

resqpy.olio.exceptions.InvalidUnitError

exception resqpy.olio.exceptions.**InvalidUnitError**

Raised when a unit cannot be converted into a valid RESQML unit of measure.

7.18.8 resqpy.olio.factors

Factorization and functions supporting grid extent determination from corner points.

Functions

all_factors	Returns a sorted list of unique factors of n.
all_factors_from_primes	Returns a sorted list of unique factors from prime factorization.
combinatorial	Returns a list of all possible product combinations of numbers from list numbers, with some duplicates.
factorize	Returns list of prime factors of positive integer n.
remove_subset	Remove all elements of subset from primary list.

resqpy.olio.factors.all_factors

resqpy.olio.factors.**all_factors**(*n*)

Returns a sorted list of unique factors of n.

resqpy.olio.factors.all_factors_from_primes

resqpy.olio.factors.**all_factors_from_primes**(*primes*)

Returns a sorted list of unique factors from prime factorization.

resqpy.olio.factors.combinatorial

resqpy.olio.factors.**combinatorial**(*numbers*)

Returns a list of all possible product combinations of numbers from list numbers, with some duplicates.

resqpy.olio.factors.factorize

resqpy.olio.factors.**factorize**(*n*)

Returns list of prime factors of positive integer n.

resqpy.olio.factors.remove_subset

resqpy.olio.factors.**remove_subset**(*primary*, *subset*)

Remove all elements of subset from primary list.

7.18.9 resqpy.olio.fine_coarse

fine_coarse.py: Module providing support for grid refinement and coarsening.

Classes

FineCoarse	Class for holding a mapping between fine and coarse grids.

resqpy.olio.fine_coarse.FineCoarse

class resqpy.olio.fine_coarse.**FineCoarse**(*fine_extent_kji*, *coarse_extent_kji*, *within_fine_box=None*, *within_coarse_box=None*)

Bases: object

Class for holding a mapping between fine and coarse grids.

Methods:

__init__(fine_extent_kji, coarse_extent_kji)	Partial initialisation function, call other methods to assign ratios and proportions.
assert_valid()	Checks consistency of everything within the fine coarse mapping; raises assertion error if not valid.
ratio(axis, c0)	Return fine:coarse ratio in given axis and coarse slice.
ratios(c_kji0)	Return find:coarse ratios triplet for coarse cell.
coarse_for_fine()	Returns triplet of numpy int vectors being the axial coarse cell indices for the axial fine cell indices.
coarse_for_fine_kji0(fine_kji0)	Returns the index of the coarse cell which the given fine cell falls within.
coarse_for_fine_axial(axis, f0)	Returns the index, for a single axis, of the coarse cell which the given fine cell falls within.
coarse_for_fine_axial_vector(axis)	Returns a numpy int vector, for a single axis, of the coarse cell index which each fine cell falls within.
fine_base_for_coarse_axial(axis, c0)	Returns the index, for a single axis, of the 'first' fine cell within the coarse cell (lowest fine index).
fine_base_for_coarse(c_kji0)	Returns a 3-tuple being the 'first' (min) k, j, i0 in fine grid for given coarse cell.
fine_box_for_coarse(c_kji0)	Return the min, max for k, j, i0 in fine grid for given coarse cell.
proportion(axis, c0)	Return the axial relative proportions of fine within coarse.
proportions_for_axis(axis)	Return the axial relative proportions as array of floats summing to one for each coarse slice.
interpolation(axis, c0)	Return a float array ready for interpoltion.
proportions(c_kji0)	Return triplet of axial proportions for refinement of coarse cell.
set_constant_ratio(axis)	Set the refinement ratio for axis based on the ratio of the fine to coarse extents.
set_ij_ratios_constant()	Set the refinement ratio for I & J axes based on the ratio of the fine to coarse extents.
set_all_ratios_constant()	Set all refinement ratios constant based on the ratio of the fine to coarse extents.
set_ratio_vector(axis, vector)	Set fine:coarse ratios for axis from numpy int vector of length matching coarse extent.
set_equal_proportions(axis)	Set proportions equal for axis.
set_all_proportions_equal()	Sets proportions equal in all 3 axes.
set_proportions_list_of_vectors(axis, ...)	Sets the proportions for given axis, with one vector for each coarse slice in the axis.
fine_for_coarse_natural_column_index(coarse_co)	Returns the fine equivalent natural (first) column index for coarse natural column index.
fine_for_coarse_natural_pillar_index(coarse_p)	Returns the fine equivalent natural (first) pillar index for coarse natural pillar index.
write_cartref(filename, lgr_name[, mode, ...])	Write Nexus ascii input format CARTREF; within_coarse_box must have been set.

__init__(*fine_extent_kji*, *coarse_extent_kji*, *within_fine_box=None*, *within_coarse_box=None*)

Partial initialisation function, call other methods to assign ratios and proportions.

Parameters

- **fine_extent_kji** (*triple int*) – the local extent of the fine grid in k, j, i axes, ie (nk, nj, ni).
- **coarse_extent_kji** (*triple int*) – the local extent of the coarse grid in k, j, i axes.
- **within_fine_box** (*numpy int array of shape (2, 3), optional*) – if present, the subset of a larger fine grid that the mapping refers to; axes are min,max and k,j,i; values are zero based indices; max values are included in box (un-pythonesque); use this in the case of a local grid coarsening
- **within_coarse_box** (*numpy int array of shape (2, 3), optional*) – if present, the subset of a larger coarse grid that the mapping refers to; axes are min,max and k,j,i; values are zero based indices; max values are included in box (un-pythonesque); use this in the case of a local grid refinement; required for write_cartref() method to work

Returns

newly formed FineCoarse object awaiting determination of ratios and proportions by axes.

Notes

at most one of within_fine_box and within_coarse_box may be passed; this information is not really used internally by the FineCoarse class but is noted in order to support local grid refinement and local grid coarsening applications; after intialisation, set_* methods should be called to establish the mapping

fine_extent_kji

fine extent

coarse_extent_kji

coarse extent

within_fine_box

if not None, a box within an unidentified larger fine grid

within_coarse_box

if not None, a box within an unidentified larger coarse grid

constant_ratios

list for 3 axes kji, each None or int

vector_ratios

list for 3 axes kji, each numpy vector of int or None

equal_proportions

list for 3 axes kji, each boolean defaulting to equal proportions

vector_proportions

list for 3 axes kji, each None or list of numpy vectors of float summing to 1.0

assert_valid()

Checks consistency of everything within the fine coarse mapping; raises assertion error if not valid.

ratio(*axis*, *c0*)

Return fine:coarse ratio in given axis and coarse slice.

ratios(*c_kji0*)

Return find:coarse ratios triplet for coarse cell.

coarse_for_fine()

Returns triplet of numpy int vectors being the axial coarse cell indices for the axial fine cell indices.

coarse_for_fine_kji0(*fine_kji0*)

Returns the index of the coarse cell which the given fine cell falls within.

coarse_for_fine_axial(*axis*, *f0*)

Returns the index, for a single axis, of the coarse cell which the given fine cell falls within.

coarse_for_fine_axial_vector(*axis*)

Returns a numpy int vector, for a single axis, of the coarse cell index which each fine cell falls within.

fine_base_for_coarse_axial(*axis*, *c0*)

Returns the index, for a single axis, of the 'first' fine cell within the coarse cell (lowest fine index).

fine_base_for_coarse(*c_kji0*)

Returns a 3-tuple being the 'first' (min) k, j, i0 in fine grid for given coarse cell.

fine_box_for_coarse(*c_kji0*)

Return the min, max for k, j, i0 in fine grid for given coarse cell.

Returns

Numpy int array of shape (2, 3) being the min, max for k, j, i0

proportion(*axis*, *c0*)

Return the axial relative proportions of fine within coarse.

Returns

numpy vector of floats, summing to one

proportions_for_axis(*axis*)

Return the axial relative proportions as array of floats summing to one for each coarse slice.

interpolation(*axis*, *c0*)

Return a float array ready for interpoltion.

Returns floats starting at zero and increasing monotonically to less than one.

proportions(*c_kji0*)

Return triplet of axial proportions for refinement of coarse cell.

set_constant_ratio(*axis*)

Set the refinement ratio for axis based on the ratio of the fine to coarse extents.

set_ij_ratios_constant()

Set the refinement ratio for I & J axes based on the ratio of the fine to coarse extents.

set_all_ratios_constant()

Set all refinement ratios constant based on the ratio of the fine to coarse extents.

set_ratio_vector(*axis*, *vector*)

Set fine:coarse ratios for axis from numpy int vector of length matching coarse extent.

set_equal_proportions(*axis*)

Set proportions equal for axis.

set_all_proportions_equal()

Sets proportions equal in all 3 axes.

set_proportions_list_of_vectors(*axis*, *list_of_vectors*)

Sets the proportions for given axis, with one vector for each coarse slice in the axis.

fine_for_coarse_natural_column_index(*coarse_col*)

Returns the fine equivalent natural (first) column index for coarse natural column index.

fine_for_coarse_natural_pillar_index(*coarse_p*)

Returns the fine equivalent natural (first) pillar index for coarse natural pillar index.

write_cartref(*filename*, *lgr_name*, *mode='a'*, *root_name=None*, *preceeding_blank_lines=0*, *trailing_blank_lines=0*)

Write Nexus ascii input format CARTREF; within_coarse_box must have been set.

Functions

axis_for_letter	Returns 0, 1 or 2 for 'K', 'J', or 'I'; as required for axis arguments in FineCoarse methods.
letter_for_axis	Returns K, J, or I for axis; axis as required for axis arguments in FineCoarse methods.
tartan_refinement	Returns a new FineCoarse object set to a tartan grid refinement; fine extent is determined from arguments.

resqpy.olio.fine_coarse.axis_for_letter

resqpy.olio.fine_coarse.**axis_for_letter**(*letter*)

Returns 0, 1 or 2 for 'K', 'J', or 'I'; as required for axis arguments in FineCoarse methods.

resqpy.olio.fine_coarse.letter_for_axis

resqpy.olio.fine_coarse.**letter_for_axis**(*axis*)

Returns K, J, or I for axis; axis as required for axis arguments in FineCoarse methods.

resqpy.olio.fine_coarse.tartan_refinement

resqpy.olio.fine_coarse.**tartan_refinement**(*coarse_extent_kji*, *coarse_fovea_box*, *fovea_ratios_kji*, *decay_rates_kji=None*, *decay_mode='exponential'*, *within_coarse_box=None*)

Returns a new FineCoarse object set to a tartan grid refinement; fine extent is determined from arguments.

Parameters

- **coarse_extent_kji** (*triple int*) – the extent of the coarse grid being refined
- **coarse_fovea_box** (*numpy int array of shape (2, 3)*) – the central box within the coarse grid to receive maximum refinement
- **fovea_ratios_kji** (*triple int*) – the maximum refinement ratios, to be applied in the coarse_fovea_box
- **decay_rates_kji** (*triple float or triple int, optional*) – controls how quickly refinement ratio reduces in slices away from fovea; if None then default values will be generated; see notes for more details

- **decay_mode** (*str, default 'exponential'*) – 'exponential' or 'linear'; see notes
- **within_coarse_box** (*numpy int array of shape (2, 3), optional*) – if present, is preserved in FineCoarse for possible use in setting resqml ParentWindow or generating Nexus CARTREF

Returns

FineCoarse object holding the tartan refinement mapping

Notes

each axis is treated independently; the fovea (box of maximum refinement) may be a column of cells (for a vertical well) or any other logical cuboid; the refinement factor is reduced monotonically in slices moving away from the fovea; two refinement factor decay functions are available: 'exponential' and 'linear', with different meaning to decay_rates_kji; for exponential decay, each decay rate should be a float in the range 0.0 to 1.0, with 0.0 causing immediate change to no refinement (factor 1), and 1.0 causing no decay (constant refinement at fovea factor); for linear decay, each decay rate should typically be a non-negative integer (though float is also okay), with 0 causing no decay, 1 causing a reduction in refinement factor of 1 per coarse slice, 2 meaning refinement factor reduces by 2 with each coarse slice etc.; in all cases, the refinement factor is given a lower limit of 1; the factor is rounded to an int for each slice, when working with floats; if decay rates are not passed as arguments, suitable values are generated to give a gradual reduction in refinement to a ratio of one at the boundary of the grid

7.18.10 resqpy.olio.grid_functions

Miscellaneous functions relating to grids.

Functions

actual_pillar_shape	Returns 'curved', 'straight' or 'vertical' for shape of pillar points.
columns_to_nearest_split_face	Return an int array of shape (NJ, NI) being number of cells to nearest split edge.
determine_corp_extent	Returns extent of grid derived from 7D corner points with all cells temporarily in I.
determine_corp_ijk_handedness	Determine true ijk handedness from corner point data in pagoda style 7D array; returns 'right' or 'left'.
infill_block_geometry	Scans logically vertical columns of cells setting depth (& thickness) of inactive cells.
left_right_foursome	Returns (2, 2) bool numpy array indicating which columns around a primary pillar are to the right of a line.
random_cell	Returns a random cell's (k,j,i) tuple for a cell with non-zero lengths on all 3 primary edges.
resequence_nexus_corp	Reorders corner point data in situ, to handle bizarre nexus orderings.
translate_corp	Adjusts x and y values of corner points by a constant offset.
triangles_for_cell_faces	Returns numpy array of shape (3, 2, 4, 3, 3) with axes being kji, -+, triangle within face, triangle corner, xyz.

resqpy.olio.grid_functions.actual_pillar_shape

resqpy.olio.grid_functions.**actual_pillar_shape**(*pillar_points*, *tolerance=0.001*)

Returns 'curved', 'straight' or 'vertical' for shape of pillar points.

Parameters

pillar_points (*numpy float array*) – fully defined points array of shape (nk + k_gaps + 1,. . . , 3).

resqpy.olio.grid_functions.columns_to_nearest_split_face

resqpy.olio.grid_functions.**columns_to_nearest_split_face**(*grid*)

Return an int array of shape (NJ, NI) being number of cells to nearest split edge.

Note: uses Manhattan distance

resqpy.olio.grid_functions.determine_corp_extent

resqpy.olio.grid_functions.**determine_corp_extent**(*corner_points*, *tolerance=0.003*)

Returns extent of grid derived from 7D corner points with all cells temporarily in I.

resqpy.olio.grid_functions.determine_corp_ijk_handedness

resqpy.olio.grid_functions.**determine_corp_ijk_handedness**(*corner_points*, *xyz_is_left_handed=True*)

Determine true ijk handedness from corner point data in pagoda style 7D array; returns 'right' or 'left'.

resqpy.olio.grid_functions.infill_block_geometry

resqpy.olio.grid_functions.**infill_block_geometry**(*extent*, *depth*, *thickness*, *x*, *y*, *k_increase_direction='down'*, *depth_zero_tolerance=0.01*, *x_y_zero_tolerance=0.01*, *vertical_cell_overlap_tolerance=0.01*, *snap_to_top_and_base=True*, *nudge=True*)

Scans logically vertical columns of cells setting depth (& thickness) of inactive cells.

Parameters

- **extent** (*numpy integer vector of shape (3,)*) – corresponds to nk, nj and ni
- **depth** (*3D numpy float array*) – shape matches extent
- **thickness** (*3D numpy float array*) – shape matches extent
- **x** (*3D numpy float array*) – shape matches extent
- **y** (*3D numpy float array*) – shape matches extent
- **k_increase_direction** (*string, default 'down'*) – direction of increasing K indices; either 'up' or 'down'
- **depth_zero_tolerance** (*float, optional, default 0.01*) – maximum value for which the depth is considered zero

- **vertical_cell_overlap_tolerance** (*float, optional, default 0.01*) – maximum acceptable overlap of cells on input
- **snap_to_top_and_base** (*boolean, optional, default True*) – when True, causes cells above topmost active and below deepest active to be populated with pinched out cells at the top and bottom faces respectively
- **nudge** (*boolean, optional, default True*) – when True causes the depth of cells with greater k to be moved to clean up overlap over pinchouts

Note: depth values are assumed more positive with increasing depth; zero values indicate inactive cells

resqpy.olio.grid_functions.left_right_foursome

resqpy.olio.grid_functions.**left_right_foursome**(*full_pillar_list*, *p_index*)

Returns (2, 2) bool numpy array indicating which columns around a primary pillar are to the right of a line.

resqpy.olio.grid_functions.random_cell

resqpy.olio.grid_functions.**random_cell**(*corner_points*, *border=0.25*, *max_tries=20*, *tolerance=0.003*)

Returns a random cell's (k,j,i) tuple for a cell with non-zero lengths on all 3 primary edges.

resqpy.olio.grid_functions.resequence_nexus_corp

resqpy.olio.grid_functions.**resequence_nexus_corp**(*corner_points*, *eight_mode=False*, *undo=False*)

Reorders corner point data in situ, to handle bizarre nexus orderings.

resqpy.olio.grid_functions.translate_corp

resqpy.olio.grid_functions.**translate_corp**(*corner_points*, *x_shift=None*, *y_shift=None*, *min_xy=None*, *preserve_digits=None*)

Adjusts x and y values of corner points by a constant offset.

resqpy.olio.grid_functions.triangles_for_cell_faces

resqpy.olio.grid_functions.**triangles_for_cell_faces**(*cp*)

Returns numpy array of shape (3, 2, 4, 3, 3) with axes being kji, -+, triangle within face, triangle corner, xyz.

Parameters

cp (*numpy float array of shape (2, 2, 2, 3)*) – single cell corner point array in pagoda protocol

Returns

numpy float array of shape (3, 2, 4, 3, 3) holding triangle corner coordinates for cell faces represented with quad triangles

Note: resqpy.surface also contains methods for working with cell faces as triangulated sets

7.18.11 resqpy.olio.intersection

intersection.py: functions to test whether lines intersect with planes.

Functions

distilled_intersects	Returns lists of line and triangle indices, and corresponding intersection points.
intersects_indices	Returns a list of the (triangle) indices where a valid intersection has been found for a single line.
last_intersects	From the result of line_set_triangles_intersects(), returns a vector of intersection points, one per line.
line_line_intersect	Returns the intersection x',y' of two lines x,y 1 to 2 and x,y 3 to 4.
line_plane_intersect	Find the intersection of a line with a plane defined by a triangle.
line_set_triangles_intersects	Find the intersections of each of set of lines within each of a set of triangles in 3D space.
line_triangle_intersect	Find the intersection of a line within a triangle in 3D space.
line_triangle_intersect_numba	Find the intersection of a line within a triangle in 3D space.
line_triangles_intersects	Find the intersections of a line within each of a set of triangles in 3D space.
lines_for_triangle	From the result of line_set_triangles_intersects(), returns a list of lines intersecing given triangle.
point_projected_to_line_2d	Return the point on the unbounded line passing through l1 & l2 which is closest to point p, in xy plane.
point_snapped_to_line_segment_2d	Returns the point on the bounded line segment l1, l2 which is closest to point p, in xy plane.
poly_line_triangles_first_intersect	Finds the first intersection of a segment of an open polyline with any of a set of triangles in 3D space.
poly_line_triangles_intersects	Find the intersections of each segment of an open polyline with each of a set of triangles in 3D space.
triangles_for_line	From the result of line_set_triangles_intersects(), returns a list of intersected triangles for a line.

resqpy.olio.intersection.distilled_intersects

resqpy.olio.intersection.**distilled_intersects**(*line_set_intersections*)

Returns lists of line and triangle indices, and corresponding intersection points.

argument:

line_set_intersections (numpy float array of shape (nl, nt, 3)): where nl is the number of lines, nt is the number of triangles and the final axis is x,y,z; nan values indicate no intersection; this array is as returned by the line_set_triangles_intersects() function or the poly_line_triangles_intersects() function

Returns

(numpy int array of shape (N,), numpy int array of shape (N,), numpy float array of shape (N, 3))

– for N intersections, first array is list of line indices, second is list of triangle indices, the third array contains the (x, y, z) coordinates of the intersection points

resqpy.olio.intersection.intersects_indices

resqpy.olio.intersection.**intersects_indices**(*single_line_intersects*)

Returns a list of the (triangle) indices where a valid intersection has been found for a single line.

resqpy.olio.intersection.last_intersects

resqpy.olio.intersection.**last_intersects**(*line_set_intersections*)

From the result of line_set_triangles_intersects(), returns a vector of intersection points, one per line.

argument:

line_set_intersections (numpy float array of shape (nl, nt, 3)): where nl is the number of lines, nt is the number of triangles and the final axis is x,y,z; nan values indicate no intersection; this array is as returned by the line_set_triangles_intersects() function or the poly_line_triangles_intersects() function

Returns

numpy float array (nl, 3) – intersection points, where nl is number of lines

Notes

Use this function to force at most one intersection point per line (with a triangulated surface). Applicable where lines are expected to be very roughly orthogonal to a gently verying untorn surface, eg. pillar lines intersecting an unfaulted horizon. If more than one triangle is intersected by a line, the returned point is for the 'last' triangle intersected by the line (when checking triangles in the order they appear in the list of triangles). If no triangles are intersected by a line, the resulting point will be (nan, nan, nan).

resqpy.olio.intersection.line_line_intersect

resqpy.olio.intersection.**line_line_intersect**(*x1*, *y1*, *x2*, *y2*, *x3*, *y3*, *x4*, *y4*, *line_segment=False*, *half_segment=False*)

Returns the intersection x',y' of two lines x,y 1 to 2 and x,y 3 to 4.

Parameters

- **x1** – coordinates of two points defining first line
- **y1** – coordinates of two points defining first line
- **x2** – coordinates of two points defining first line
- **y2** – coordinates of two points defining first line
- **x3** – coordinates of two points defining second line
- **y3** – coordinates of two points defining second line
- **x4** – coordinates of two points defining second line
- **y4** – coordinates of two points defining second line

- **line_segment** (*bool, default False*) – if False, both lines are treated as unbounded; if True second line is treated as segment bounded by both end points and first line bounding depends on half_segment arg
- **half_segment** (*bool, default False*) – if True and line_segment is True, first line is bounded only at end point x1, y1, whilst second line is fully bounded; if False and line_segment is True, first line is also treated as fully bounded; ignored if line_segment is False (ie. both lines unbounded)

Returns
pair of floats being x, y of intersection point; or None, None if no qualifying intersection

Note: in the case of bounded line segments, both end points are 'included' in the segment

resqpy.olio.intersection.line_plane_intersect

resqpy.olio.intersection.**line_plane_intersect**(*line_p*, *line_v*, *triangle*)

Find the intersection of a line with a plane defined by a triangle.

Parameters

- **line_p** (*3 element numpy vector*) – a point on the line
- **line_v** (*3 element numpy vector*) – vector being the direction of the line
- **triangle** (*(3, 3) numpy array*) – three points on the plane (second index is xyz)

Returns
point (3 element numpy vector) of intersection of the line with the plane, or None if line is parallel to plane

resqpy.olio.intersection.line_set_triangles_intersects

resqpy.olio.intersection.**line_set_triangles_intersects**(*line_ps*, *line_vs*, *triangles*, *line_segment=False*)

Find the intersections of each of set of lines within each of a set of triangles in 3D space.

Parameters

- **line_ps** (*(c, 3) numpy array*) – a point on each of c lines
- **line_vs** (*(c, 3) numpy array*) – vectors being the direction of each of the c lines (or 1 common vector)
- **triangles** (*(n, 3, 3) numpy array*) – three corners of each of the n triangles (final index is xyz)
- **line_segment** (*boolean, default False*) – if True, each line is treated as a finite segment between p and p + v, and only intersections within the segment are included

Returns
points ((c, n, 3) numpy array) of intersections of the lines within the triangles, (nan, nan, nan) where a line is parallel to plane of triangle or intersection with the plane is outside the triangle

Note: this function is computationally and memory intensive; it could benefit from parallelisation

resqpy.olio.intersection.line_triangle_intersect

resqpy.olio.intersection.**line_triangle_intersect**(*line_p*, *line_v*, *triangle*, *line_segment=False*, *l_tol=0.0*, *t_tol=0.0*)

Find the intersection of a line within a triangle in 3D space.

Parameters

- **line_p** (*3 element numpy vector*) – a point on the line
- **line_v** (*3 element numpy vector*) – vector being the direction of the line
- **triangle** (*(3, 3) numpy array*) – three corners of the triangle (second index is xyz)
- **line_segment** (*boolean*) – if True, returns None if intersection is outwith (line_p .. line_p + line_v)
- **l_tol** (*float, default 0.0*) – a fraction of the line length to allow for an intersection to be found just outside the segment
- **t_tol** (*float, default 0.0*) – a fraction of the triangle size to allow for an intersection to be found just outside the triangle

Returns

point (3 element numpy vector) of intersection of the line within the triangle, or None if line is parallel to plane of triangle or intersection with the plane is outside the triangle

resqpy.olio.intersection.line_triangle_intersect_numba

resqpy.olio.intersection.**line_triangle_intersect_numba**(*line_p: ndarray*, *line_v: ndarray*, *triangle: ndarray*, *line_segment: bool = False*, *l_tol: float = 0.0*, *t_tol: float = 0.0*) → Union[None, ndarray]

Find the intersection of a line within a triangle in 3D space.

Parameters

- **line_p** (*np.ndarray*) – a point on the line.
- **line_v** (*np.ndarray*) – vector being the direction of the line.
- **triangle** (*np.ndarray*) – shape (3, 3); three corners of the triangle (second index is xyz).
- **line_segment** (*bool*) – if True, returns None if intersection is outwith (line_p .. line_p + line_v).
- **l_tol** (*float, default 0.0*) – a fraction of the line length to allow for an intersection to be found just outside the segment.
- **t_tol** (*float, default 0.0*) – a fraction of the triangle size to allow for an intersection to be found just outside the triangle.

Returns

point (np.ndarray) of intersection of the line within the triangle, or None if line is parallel to plane of triangle or intersection with the plane is outside the triangle.

resqpy.olio.intersection.line_triangles_intersects

resqpy.olio.intersection.**line_triangles_intersects**(*line_p*, *line_v*, *triangles*, *line_segment=False*)

Find the intersections of a line within each of a set of triangles in 3D space.

Parameters

- **line_p** (*3 element numpy vector*) – a point on the line
- **line_v** (*3 element numpy vector*) – vector being the direction of the line
- **triangles** (*(n, 3, 3) numpy array*) – three corners of each of the n triangles (final index is xyz)
- **line_segment** (*boolean, default False*) – if True, the line is treated as a finite segment between p and p + v, and only intersections within the segment are included

Returns

points ((n, 3) numpy array) of intersection points of the line within the triangles, (nan, nan, nan) where line is parallel to plane of triangle or intersection with the plane is outside the triangle (or beyond the ends of the segment if applicable)

resqpy.olio.intersection.lines_for_triangle

resqpy.olio.intersection.**lines_for_triangle**(*line_set_intersections*, *triangle_index*)

From the result of line_set_triangles_intersects(), returns a list of lines intersecing given triangle.

Parameters

- **line_set_intersections** (*numpy float array of shape (nl, nt, 3)*) – where nl is the number of lines, nt is the number of triangles and the final axis is x,y,z; nan values indicate no intersection; this array is as returned by the line_set_triangles_intersects() function or the poly_line_triangles_intersects() function
- **triangle_index** (*integar*) – the index of the triangle for which the intersecting line list is required

Returns

(numpy 1D int array of size N, numpy 2D array of shape (N, 3)) –

the first of the pair of arrays
returned is a list of the indices of lines which intersect with the given triangle; the second array is the list of corresponding intersection points (each x,y,z)

Notes

if no lines intersect the triangle, both the resulting arrays will have size zero

resqpy.olio.intersection.point_projected_to_line_2d

resqpy.olio.intersection.**point_projected_to_line_2d**(*p*, *l1*, *l2*)

Return the point on the unbounded line passing through l1 & l2 which is closest to point p, in xy plane.

resqpy.olio.intersection.point_snapped_to_line_segment_2d

resqpy.olio.intersection.**point_snapped_to_line_segment_2d**(*p*, *l1*, *l2*)

Returns the point on the bounded line segment l1, l2 which is closest to point p, in xy plane.

resqpy.olio.intersection.poly_line_triangles_first_intersect

resqpy.olio.intersection.**poly_line_triangles_first_intersect**(*line_ps*, *triangles*, *start=0*)

Finds the first intersection of a segment of an open poly-line with any of a set of triangles in 3D space.

Parameters

- **line_ps** (*(c, 3) numpy array*) – ordered points on each of c-1 segments of a poly-line
- **triangles** (*(n, 3, 3) numpy array*) – three corners of each of the n triangles (final index is xyz)
- **start** (*int, default 0*) – the index of the point in line_ps to start searching from

Returns

(int, numpy float array of shape (3,)) where the int is the line segment index where one or more intersections were found and the floats are the xyz of one intersection of that line segment within the triangles; if no line segments intersect any of the trianges, (None, None) is returned

Note: this function is computationally and memory intensive; it could benefit from parallelisation

resqpy.olio.intersection.poly_line_triangles_intersects

resqpy.olio.intersection.**poly_line_triangles_intersects**(*line_ps*, *triangles*)

Find the intersections of each segment of an open poly-line with each of a set of triangles in 3D space.

Parameters

- **line_ps** (*(c, 3) numpy array*) – ordered points on each of c-1 segments of a poly-line
- **triangles** (*(n, 3, 3) numpy array*) – three corners of each of the n triangles (final index is xyz)

Returns

points ((c-1, n, 3) numpy array) of intersections of the line segments within the triangles, (nan, nan, nan) where a line segment is parallel to plane of triangle or intersection of the segment with the plane is outside the triangle or beyond the ends of the segment

Note: this function is computationally and memory intensive; it could benefit from parallelisation

resqpy.olio.intersection.triangles_for_line

resqpy.olio.intersection.**triangles_for_line**(*line_set_intersections*, *line_index*)

From the result of line_set_triangles_intersects(), returns a list of intersected triangles for a line.

Parameters

- **line_set_intersections** (*numpy float array of shape (nl, nt, 3)*) – where nl is the number of lines, nt is the number of triangles and the final axis is x,y,z; nan values indicate no intersection; this array is as returned by the line_set_triangles_intersects() function or the poly_line_triangles_intersects() function
- **line_index** (*integar*) – the index of the line for which the intersected triangle list is required

Returns

(numpy 1D int array of size N, numpy 2D array of shape (N, 3)) –

the first of the pair of arrays
returned is a list of the triangle indices which the given line intersects; the second array is the list of corresponding intersection points (each x,y,z)

Notes

if the line does not intersect any triangles, both the resulting arrays will have size zero

7.18.12 resqpy.olio.keyword_files

Basic functions for searching for keywords in an ascii control file such as a nexus deck.

Ascii file must already have been opened for reading before calling any of these functions.

Functions

blank_line	Returns True if the next line contains only white space; False otherwise (including comments).
end_of_file	Returns True if the end of the file has been reached.
find_keyword	Looks for line starting with given keyword; file pointer is left at start of that line.
find_keyword_pair	Looks for line starting with a given pair of keywords.
find_keyword_with_copy	Looks for line starting with given keyword, copying lines in the meantime.
find_keyword_without_passing	Looks for line starting with keyword, but without passing line starting with no_pass_keyword.
find_number	Looks for line starting with any number.
guess_comment_char	Returns a string (usually one character) being the guess as to the comment character, or None.
number_next	Returns True if next token in file is a number.
skip_blank_lines_and_comments	Skips any lines containing only white space or comment.
skip_comments	Skips any lines containing only a comment.
specific_keyword_next	Returns True if next token in file is the specified keyword.
split_trailing_comment	Returns a pair of strings: (line stripped of trailing comment, trailing comment).
strip_trailing_comment	Returns a copy of line with any trailing comment removed.
substring	Returns True if the first argument is a substring of the second.

resqpy.olio.keyword_files.blank_line

resqpy.olio.keyword_files.**blank_line**(*ascii_file*)

Returns True if the next line contains only white space; False otherwise (including comments).

resqpy.olio.keyword_files.end_of_file

resqpy.olio.keyword_files.**end_of_file**(*ascii_file*)

Returns True if the end of the file has been reached.

resqpy.olio.keyword_files.find_keyword

resqpy.olio.keyword_files.**find_keyword**(*ascii_file*, *keyword*, *max_lines=None*)

Looks for line starting with given keyword; file pointer is left at start of that line.

resqpy.olio.keyword_files.find_keyword_pair

resqpy.olio.keyword_files.**find_keyword_pair**(*ascii_file*, *primary_keyword*, *secondary_keyword*)

Looks for line starting with a given pair of keywords.

resqpy.olio.keyword_files.find_keyword_with_copy

resqpy.olio.keyword_files.**find_keyword_with_copy**(*ascii_file_in*, *keyword*, *ascii_file_out*)

Looks for line starting with given keyword, copying lines in the meantime.

resqpy.olio.keyword_files.find_keyword_without_passing

resqpy.olio.keyword_files.**find_keyword_without_passing**(*ascii_file*, *keyword*, *no_pass_keyword*)

Looks for line starting with keyword, but without passing line starting with no_pass_keyword.

resqpy.olio.keyword_files.find_number

resqpy.olio.keyword_files.**find_number**(*ascii_file*)

Looks for line starting with any number.

resqpy.olio.keyword_files.guess_comment_char

resqpy.olio.keyword_files.**guess_comment_char**(*ascii_file*)

Returns a string (usually one character) being the guess as to the comment character, or None.

resqpy.olio.keyword_files.number_next

resqpy.olio.keyword_files.**number_next**(*ascii_file*, *skip_blank_lines=True*, *comment_char='!'*)

Returns True if next token in file is a number.

resqpy.olio.keyword_files.skip_blank_lines_and_comments

resqpy.olio.keyword_files.**skip_blank_lines_and_comments**(*ascii_file*, *comment_char='!'*, *skip_c_space=True*)

Skips any lines containing only white space or comment.

resqpy.olio.keyword_files.skip_comments

resqpy.olio.keyword_files.**skip_comments**(*ascii_file*, *comment_char='!'*, *skip_c_space=True*)

Skips any lines containing only a comment.

resqpy.olio.keyword_files.specific_keyword_next

resqpy.olio.keyword_files.**specific_keyword_next**(*ascii_file*, *keyword*, *skip_blank_lines=True*, *comment_char='!'*)

Returns True if next token in file is the specified keyword.

resqpy.olio.keyword_files.split_trailing_comment

resqpy.olio.keyword_files.**split_trailing_comment**(*line*, *comment_char='!'*)

Returns a pair of strings: (line stripped of trailing comment, trailing comment).

resqpy.olio.keyword_files.strip_trailing_comment

resqpy.olio.keyword_files.**strip_trailing_comment**(*line*, *comment_char='!'*)

Returns a copy of line with any trailing comment removed.

resqpy.olio.keyword_files.substring

resqpy.olio.keyword_files.**substring**(*shorter*, *longer*)

Returns True if the first argument is a substring of the second.

7.18.13 resqpy.olio.load_data

Functions to load data from various ASCII simulator file formats.

Functions

file_exists	Returns True if the file exists (and is more recent than other file, if given).
load_array_from_ascii_file	Returns a numpy array with data loaded from an ascii file.
load_array_from_file	Load an array from an ascii (or pure binary) file.
load_corp_array_from_file	Loads a nexus corner point (CORP) array from a file, returns a 7D numpy array in pagoda ordering.

resqpy.olio.load_data.file_exists

resqpy.olio.load_data.**file_exists**(*file_name*, *must_be_more_recent_than_file=None*)

Returns True if the file exists (and is more recent than other file, if given).

resqpy.olio.load_data.load_array_from_ascii_file

resqpy.olio.load_data.**load_array_from_ascii_file**(*file_name*, *extent=None*, *data_type='real'*, *keyword=None*, *max_lines_for_keyword=None*, *comment_char=None*, *data_free_of_comments=False*, *skip_c_space=True*, *use_numbers_only=None*)

Returns a numpy array with data loaded from an ascii file.

Parameters

- **file_name** – string holding name of the existing ascii data file, eg. 'Cell_depth.dat'
- **extent** – a python list of integers specifying the extent (shape) of the array, eg. [148, 270, 103]; if None, all data is read and returned as a 'flat' 1D array (data must be free from comments)
- **data_type** – the type of individual data elements, one of 'real','float','int', 'integer', 'bool' or 'boolean'
- **keyword** – if present, an attempt is made to find the keyword before reading data, if keyword is None or is not found, data is read from the start of the file
- **max_lines_for_keyword** – can be used to limit the search for keyword (for speed efficiency)
- **comment_char** – a single character string being the character used to introduce a comment in the file
- **data_free_of_comments** – if set to True, a faster load is used once any header line comments have been skipped
- **skip_c_space** – if True then a line starting 'C' followed by white space is skipped as a comment
- **use_numbers_only** – this argument is no longer in use and is ignored

Returns

a numpy array of shape specified in extent argument with dtype matching data_type

example call:

depth_array = load_data.load_array_from_ascii_file('Cell_depth.dat', [148, 270, 103])

Notes

In all use cases, this function is designed to load a single array of data from an ascii file that DOES NOT CONTAIN OTHER ARRAYS as well, ie. data for a single simulation keyword in the file.

If skip_c_space is True, lines starting 'C ' are also treated as comments. If data_free_of_comments is True, there must be at least one blank line before the data begins, and no further comments are permitted. (This format is designed to handle data files generated by a commonly used geomodelling package.)

Repeat counts must not be present in the ascii data.

The extent, if present, can contain any number of dimensions, typically 3 for reservoir modelling work. The total number of numbers in the file must match the number of elements in the given extent (ie. the product of the list of numbers in the extent argument). The order of indices in extent should be 'slowest changing' first, eg.: k,j,i

The data_type defaults to 'real' 'real' and 'float' are synonymous; 'int' and 'integer' are synonymous; 'bool' and 'boolean' are synonymous; default is 'real' The numpy data type will be the default 64 bit float or 64 bit int

resqpy.olio.load_data.load_array_from_file

resqpy.olio.load_data.**load_array_from_file**(*file_name*, *extent=None*, *data_type='real'*, *keyword=None*, *max_lines_for_keyword=None*, *comment_char=None*, *data_free_of_comments=False*, *use_binary=False*, *use_numbers_only=None*)

Load an array from an ascii (or pure binary) file.

Arguments are similar to those for load_corp_array_from_file().

resqpy.olio.load_data.load_corp_array_from_file

resqpy.olio.load_data.**load_corp_array_from_file**(*file_name*, *extent_kji=None*, *corp_bin=False*, *swap_bytes=True*, *max_lines_for_keyword=100*, *comment_char=None*, *data_free_of_comments=False*, *use_binary=False*, *eight_mode=False*, *use_numbers_only=None*)

Loads a nexus corner point (CORP) array from a file, returns a 7D numpy array in pagoda ordering.

Parameters

- **file_name** – The name of an ascii file holding the CORP data (no other keywords with numeric data should be in the file); write access to the directory is likely to be needed if use_binary is True
- **extent_kji** – The extent of the grid as a list or a 3 element numpy array, in the order [NK, NJ, NI]. If extent_kji is None, the extent is figured out from the data. It must be given for 1D or 2D models
- **corp_bin** (*boolean, default False*) – if True, input file is in bespoke corp binary format, otherwise ascii
- **swap_bytes** (*boolean, default True*) – if True, byte ordering of corp bin data is reversed; only relevant if corp_bin is True
- **max_lines_for_keyword** – the maximum number of lines to search for CORP keyword; set to zero if file is known to be data only
- **comment_char** – A single character string which is interpreted as introducing a comment
- **data_free_of_comments** – If True, once the numeric data is encountered, it is assumed that there are no further comments (allowing a faster load)
- **use_binary** – If True, a more recent file containing a pure binary copy of the data is looked for first, in the same directory; if found, the data is loaded directly from that file; if not found, the binary file is created after the ascii has been loaded (ready for next time)
- **eight_mode** – If True, the data is assumed to be in CORP EIGHT ordering; otherwise the normal ordering (The code does not look for keywords.); this is not automatically determined from any keyword in the file
- **use_numbers_only** – no longer in use, ignored

Returns

A numpy array containing the CORP data in 7D pagoda protocol ordering. The extent of the grid, and hence shape of the array is determined from the corner point data unless extent_kji has been specified.

7.18.14 resqpy.olio.point_inclusion

point_inclusion.py: functions to test whether a point is within a polygon; also line intersection with planes.

Functions

pip_array_cn	Return bool array of 2D points inclusion, True where point is inside polygon.
pip_cn	2D point inclusion: returns True if point is inside polygon (uses crossing number algorithm).
pip_wn	2D point inclusion: returns True if point is inside polygon (uses winding number algorithm).
points_in_polygon	Takes a pair of numpy arrays x, y defining points to be tested against polygon specified in polygon_file.
scan	Scans lines of pixels returning 2D boolean array of points within convex polygon.

resqpy.olio.point_inclusion.pip_array_cn

resqpy.olio.point_inclusion.**pip_array_cn**(*p_a*, *poly*)

Return bool array of 2D points inclusion, True where point is inside polygon.

Uses crossing number algorithm.

Parameters

- **p_a** (*2D numpy float array*) – set of xy points to test for inclusion in polygon
- **poly** (*2D numpy array, tuple or list of tuples or lists of at least 2 floats*) – the xy points defining a polygon

Returns

numpy boolean vector – True where corresponding point in p_a is within the polygon

Note: if the polygon is represented by a closed resqpy Polyline, pass Polyline.coordinates as poly

resqpy.olio.point_inclusion.pip_cn

resqpy.olio.point_inclusion.**pip_cn**(*p*, *poly*)

2D point inclusion: returns True if point is inside polygon (uses crossing number algorithm).

Parameters

- **p** (*numpy array, tuple or list of at least 2 floats*) – xy point to test for inclusion in polygon
- **poly** (*2D numpy array, tuple or list of tuples or lists of at least 2 floats*) – the xy points defining a polygon

Returns

boolean – True if point is within the polygon

Note: if the polygon is represented by a closed resqpy Polyline, pass Polyline.coordinates as poly

resqpy.olio.point_inclusion.pip_wn

resqpy.olio.point_inclusion.**pip_wn**(*p*, *poly*)

2D point inclusion: returns True if point is inside polygon (uses winding number algorithm).

Parameters

- **p** (*numpy array, tuple or list of at least 2 floats*) – xy point to test for inclusion in polygon
- **poly** (*2D numpy array, tuple or list of tuples or lists of at least 2 floats*) – the xy points defining a polygon

Returns

boolean – True if point is within the polygon

Note: if the polygon is represented by a closed resqpy Polyline, pass Polyline.coordinates as poly

resqpy.olio.point_inclusion.points_in_polygon

resqpy.olio.point_inclusion.**points_in_polygon**(*x*, *y*, *polygon_file*, *poly_unit_multiplier=None*)

Takes a pair of numpy arrays x, y defining points to be tested against polygon specified in polygon_file.

resqpy.olio.point_inclusion.scan

resqpy.olio.point_inclusion.**scan**(*origin*, *ncol*, *nrow*, *dx*, *dy*, *poly*)

Scans lines of pixels returning 2D boolean array of points within convex polygon.

7.18.15 resqpy.olio.random_seed

Module providing wrapper for random number generator seeding functions.

Functions

seed	Set seed for random number generator of one or more packages, to allow for repeatable behaviour.

resqpy.olio.random_seed.seed

resqpy.olio.random_seed.**seed**(*seed*, *package='all'*)

Set seed for random number generator of one or more packages, to allow for repeatable behaviour.

Parameters

- **seed** (*int*) – the value to use to seed the random number generator(s); a value of None will generally result in an unrepeatable sequence
- **package** (*string or list of strings*) – one or more of known packages: 'random' and 'numpy' at present; passing 'all' will cause all packages known to have a random number generator to be re-seeded

7.18.16 resqpy.olio.read_nexus_fault

read_nexus_fault.py: functions for reading Nexus fault definition data from an ascii file.

Functions

load_nexus_fault_mult_table	Reads a Nexus (!) format file containing one or more MULT keywords and returns a dataframe with the MULT rows.
load_nexus_fault_mult_table_from_list	Reads a Nexus (!) format list of file contents containing one or more MULT keywords and returns a dataframe with the MULT rows.

resqpy.olio.read_nexus_fault.load_nexus_fault_mult_table

resqpy.olio.read_nexus_fault.**load_nexus_fault_mult_table**(*file_name*)

Reads a Nexus (!) format file containing one or more MULT keywords and returns a dataframe with the MULT rows.

resqpy.olio.read_nexus_fault.load_nexus_fault_mult_table_from_list

resqpy.olio.read_nexus_fault.**load_nexus_fault_mult_table_from_list**(*file_as_list*)

Reads a Nexus (!) format list of file contents containing one or more MULT keywords and returns a dataframe with the MULT rows.

7.18.17 resqpy.olio.relperm

relperm.py: class for dataframes of relative permeability data as RESQML objects.

Note that this module uses the obj_Grid2dRepresentation class in a way that was not envisaged when the RESQML standard was defined; software that does not use resqpy is unlikely to be able to do much with data stored in this way.

Classes

RelPerm	Class for storing and retrieving a pandas dataframe of relative permeability data.

Functions

relperm_parts_in_model	Returns list of part names within model that are representing RelPerm dataframe support objects.
text_to_relperm_dict	Return dict of dataframes with relative permeability and capillary pressure data and phase combinations.

resqpy.olio.relperm.relperm_parts_in_model

resqpy.olio.relperm.**relperm_parts_in_model**(*model*, *phase_combo=None*, *low_sal=None*, *table_index=None*, *title=None*, *related_uuid=None*)

Returns list of part names within model that are representing RelPerm dataframe support objects.

Parameters

- **model** (model.Model) – the model to be inspected for dataframes
- **phase_combo** (*str, optional*) – the combination of phases whose relative permeability behaviour is described. Options include 'water-oil', 'gas-oil', 'gas-water', 'oil-water', 'oil-gas' and 'water-gas'
- **low_sal** (*boolean, optional*) – if True, indicates that the water-oil table contains the low-salinity data for relative permeability and capillary pressure
- **table_index** (*int, optional*) – the index of the relative permeability table when multiple relative permeability tables are present. Note, indices should start at 1.
- **title** (*str, optional*) – if present, only parts with a citation title exactly matching will be included
- **related_uuid** (*uuid, optional*) – if present, only parts relating to this uuid are included

Returns

list of str, each element in the list is a part name, within model, which is representing the support for a RelPerm object

resqpy.olio.relperm.text_to_relperm_dict

resqpy.olio.relperm.**text_to_relperm_dict**(*relperm_data*, *is_file=True*)

Return dict of dataframes with relative permeability and capillary pressure data and phase combinations.

Parameters

- **relperm_data** (*str*) – relative or full path of the text file to be processed or string of relative permeability data
- **is_file** (*boolean*) – if True, indicates that a text file of relative permeability data has been provided. Default value is True

Returns

dict, each element in the dictionary contains a dataframe, with saturation and rel. permeability/capillary pressure data, and the phase combination being described

---

Note:

Only Nexus compatible text files are currently supported. Text files from other reservoir simulators may be
supported in the future.

7.18.18 resqpy.olio.simple_lines

simple_lines.py: functions for handling simple lines in relation to a resqml grid.

Functions

drape_lines	Roughly drapes lines over grid horizon; draped lines are suitable for 3D visualisation.
drape_lines_to_rods	Roughly drapes lines near grid cross section; draped lines are suitable for 3D visualisation.
duplicate_vertices_removed	Returns a copy of the line with neighbouring duplicate vertices removed.
nearest_pillars	Finds pillars nearest to each point on each line; returns list of lists of (j0, i0).
nearest_rods	Finds rods nearest to each point on each line; returns list of lists of (k0, j0) or (k0, i0).
polygon_line	Returns a copy of the line with the last vertex stripped off if it is close to the first vertex.
read_lines	Returns a list of line arrays, read from ascii file.

resqpy.olio.simple_lines.drape_lines

resqpy.olio.simple_lines.**drape_lines**(*line_list*, *pillar_list_list*, *grid*, *ref_k=0*, *ref_kp=0*, *offset=-1.0*, *snap=False*)

Roughly drapes lines over grid horizon; draped lines are suitable for 3D visualisation.

Parameters

- **line_list** (*list of numpy arrays of floats*) – the undraped poly-lines
- **pillar_list_list** – (list of lists of pairs of integers): as returned by nearest_pillars()
- **grid** – (grid.Grid object): the grid to which the poly-lines are to be draped
- **ref_k** (*integer, default 0*) – the reference layer in the grid to which the lines will be draped; zero based
- **ref_kp** (*integer, default 0*) – 0 to indicate the top corners of the reference layer, 1 for the base
- **offset** (*float, default -1.0*) – the vertical offset to add to the z value of pillar points; positive drapes lines deeper, negative shallower (assumes grid's crs has z increasing downwards)

- **snap** (*boolean, default False*) – if True, the x & y values of each vertex in the lines are moved to match the pillar point; if False, only the z values are adjusted

Returns
list of numpy arrays of floats, being the draped equivalents of the line_list

Notes

the units of the line list must implicitly be those of the crs for the grid; for grids with split coordinate lines (faults), only the primary pillars are currently used; the results of this function are intended for 3D visualisation of an indicative nature; resulting draped lines may penetrate the grid layer faces depending on the 3D geometry and the spacing of the vertices compared to cell sizes

resqpy.olio.simple_lines.drape_lines_to_rods

resqpy.olio.simple_lines.**drape_lines_to_rods**(*line_list*, *rod_list_list*, *projection*, *grid*, *axis*, *ref_slice0=0*, *plus_face=False*, *offset=-1.0*, *snap=False*)

Roughly drapes lines near grid cross section; draped lines are suitable for 3D visualisation.

Parameters

- **line_list** (*list of numpy arrays of floats*) – the undraped poly-lines
- **rod_list_list** – (list of arrays of pairs of integers): as returned by nearest_rods()
- **grid** – (grid.Grid object): the grid to which the poly-lines are to be draped
- **axis** (*string*) – 'I' or 'J' being the axis removed during slicing
- **ref_slice0** (*integer, default 0*) – the reference slice in the grid to drape to; zero based
- **plus_face** (*boolean, default False*) – which face of the reference slice to use
- **offset** (*float, default -1.0*) – the horzontal offset to add to the y or x value of rod points
- **snap** (*boolean, default False*) – if True, the x & z or y & z values of each vertex in the lines are moved to match the rod point; if False, only the y or x values are adjusted

Returns
list of numpy arrays of floats, being the draped equivalents of the line_list

Notes

the units of the line list must implicitly be those of the crs for the grid; currently limited to unsplit grids with no k gaps; the results of this function are intended for 3D visualisation of an indicative nature; resulting draped lines may penetrate the grid layer faces depending on the 3D geometry and the spacing of the vertices compared to cell sizes

resqpy.olio.simple_lines.duplicate_vertices_removed

resqpy.olio.simple_lines.**duplicate_vertices_removed**(*line*, *tolerance=0.001*)

Returns a copy of the line with neighbouring duplicate vertices removed.

Parameters

- **line** (*2D numpy array of floats*) – representation of a poly-line
- **tolerance** (*float, default = 0.001*) – the maximum Manhatten distance between two points for them to be treated as coincident

Returns

numpy array of floats which is either a copy of line, or a copy of line with some points removed

Notes

does not treat the line as a closed polyline, use polygon_line() function as well to remove duplicated first/last point; always preserves first and last point, even if they are identical and there are no other vertices

resqpy.olio.simple_lines.nearest_pillars

resqpy.olio.simple_lines.**nearest_pillars**(*line_list*, *grid*, *ref_k=0*, *ref_kp=0*)

Finds pillars nearest to each point on each line; returns list of lists of (j0, i0).

Parameters

- **line_list** (*list of numpy arrays of floats*) – set of poly-lines, the points of which are used to find the nearest pillars in grid
- **grid** (*grid.Grid object*) – the grid whose pillars are compared with the poly-line vertices
- **ref_k** (*integer, default 0*) – the reference layer in the grid to compare against the vertices; zero based
- **ref_kp** (*integer, default 0*) – 0 to indicate the top corners of the reference layer, 1 for the base

Returns

a list of lists of pairs of integers, each being the (j, i) pillar indices of the nearest pillar to the corresponding vertex of the poly-line; zero based indexing

Notes

this is a 2D search in the x, y plane; z values are ignored; poly-line x, y values must implicitly be in the same crs as the grid's points data

resqpy.olio.simple_lines.nearest_rods

resqpy.olio.simple_lines.**nearest_rods**(*line_list*, *projection*, *grid*, *axis*, *ref_slice0=0*, *plus_face=False*)

Finds rods nearest to each point on each line; returns list of lists of (k0, j0) or (k0, i0).

Parameters

- **line_list** (*list of numpy arrays of floats*) – set of poly-lines, the points of which are used to find the nearest rods in grid
- **projection** (*string*) – 'xz' or 'yz'
- **grid** (*grid.Grid object*) – the grid whose cross section points are compared with the poly-line vertices
- **axis** (*string*) – 'I' or 'J' being the axis removed during slicing
- **ref_slice0** (*integer, default 0*) – the reference slice in the grid to compare against the vertices; zero based
- **plus_face** (*boolean, default False*) – which face of the reference slice to use

Returns

a list of numpy arrays of pairs of integers, each being the (k, j) or (k, i) indices of the nearest rod to the corresponding vertex of the poly-line under projection; zero based indexing

Notes

this is a 2D search in the x, z or y, z plane; currently limited to unsplit grids without k gaps; poly-line x, y, z values must implicitly be in the same crs as the grid's points data

resqpy.olio.simple_lines.polygon_line

resqpy.olio.simple_lines.**polygon_line**(*line*, *tolerance=0.001*)

Returns a copy of the line with the last vertex stripped off if it is close to the first vertex.

Parameters

- **line** (*numpy array of floats*) – representation of a poly-line which might be closed (last vertex matches first vertex)
- **tolerance** (*float, default = 0.001*) – the maximum Manhatten distance between two points for them to be treated as coincident

Returns

numpy array of floats which is either a copy of line, or a copy of line with the last point removed

resqpy.olio.simple_lines.read_lines

resqpy.olio.simple_lines.**read_lines**(*filename*)

Returns a list of line arrays, read from ascii file.

argument:

filename (string): the path of the ascii file holding a set of poly-lines

Returns

list of numpy arrays, each array representing one poly-line

Notes

each line in the file must contain 3 floating point numbers: x, y, z; each poly-line must be terminated with a null marker line: 999.0 999.0 999.0 there is no handling of units; elsewhere they will implicitly be assumed to be those of a crs for a grid object

7.18.19 resqpy.olio.time

time.py: A very thin wrapper around python datetime functionality, to meet resqml standard.

Functions

now	Returns an iso format string representation of the current time, to the nearest second.

resqpy.olio.time.now

resqpy.olio.time.**now**(*use_utc=False*)

Returns an iso format string representation of the current time, to the nearest second.

argument:
use_utc (boolean, default False): if True, the current UTC time is used, otherwise local time

Returns
string of form YYYY-MM-DDThh – mm:ssZ representing the current time in iso format

Note: this is the format used by the resqml standard for representing date-times

7.18.20 resqpy.olio.trademark

trademark.py module for mentioning trademarks in diagnostic log.

Functions

log_nexus_tm	Produces a Nexus trademark log message once at the given severity.

resqpy.olio.trademark.log_nexus_tm

resqpy.olio.trademark.**log_nexus_tm**(*level=20*)

Produces a Nexus trademark log message once at the given severity.

Note: this function should be called after referring to Nexus in another log message, passing the severity of that other message

7.18.21 resqpy.olio.transmission

Transmissibility functions for grids.

Functions

fault_connection_set	Builds a GridConnectionSet for juxtaposed faces where there is a split pillar, with fractional area data.
half_cell_t	Creates a half cell transmissibilty property array for a regular or irregular IJK grid.
half_cell_t_2d_triangular_precursor	Creates a precursor to horizontal transmissibility for prism grids (see notes).
half_cell_t_irregular	Creates a half cell transmissibilty property array for an IJK grid.
half_cell_t_regular	Creates a half cell transmissibilty property array for a RegularGrid.
half_cell_t_vertical_prism	Creates a half cell transmissibilty property array for a vertical prism grid.
projected_tri_area	Return array holding areas of triangles projected onto each of yz, xz, xy.

resqpy.olio.transmission.fault_connection_set

resqpy.olio.transmission.**fault_connection_set**(*grid*, *skip_inactive=False*)

Builds a GridConnectionSet for juxtaposed faces where there is a split pillar, with fractional area data.

Parameters

- **grid** (*grid.Grid object*) – the grid for which a fault connection set is required
- **skip_inactive** (*boolean, default False*) – if True, connections where either cell is inactive will be excluded

Returns

(GridConnectionSet, numpy float array of shape (count, 2)) where the connection set identifies all cell face pairs where there is juxtaposition and the array contains the fraction of the face areas that are juxtaposed; count is the number of cell face pairs in the connection set

Notes

the current algorithm is designed for faults where slip has occurred along pillars – sideways slip (strike-slip) will currently cause erroneous results; inaccuracies may also arise as pillars become less straight, less co-planar or less parallel; the combination of non-parallel pillars and layers of non-uniform thickness can produce inaccuracies in some situations; if the grid does not have split pillars (ie. is unfaulted), or if there are no qualifying connections across faults, then (None, None) will be returned; as fractional areas are returned, the results are applicable whether xy & z units are the same or differ

resqpy.olio.transmission.half_cell_t

resqpy.olio.transmission.**half_cell_t**(*grid*, *perm_k=None*, *perm_j=None*, *perm_i=None*, *ntg=None*, *realization=None*, *darcy_constant=None*, *tolerance=1e-06*)

Creates a half cell transmissibilty property array for a regular or irregular IJK grid.

Parameters

- **grid** (*grid.Grid or grid.RegularGrid*) – the grid for which half cell transmissibilities are required
- **perm_k** (*float arrays of shape (nk, nj, ni), optional*) – cell permeability values (for each direction), in mD; if None, the permeabilities are found in the grid's property collection
- **j** (*float arrays of shape (nk, nj, ni), optional*) – cell permeability values (for each direction), in mD; if None, the permeabilities are found in the grid's property collection
- **i** (*float arrays of shape (nk, nj, ni), optional*) – cell permeability values (for each direction), in mD; if None, the permeabilities are found in the grid's property collection
- **ntg** (*float array of shape (nk, nj, ni), or float, optional*) – net to gross values to apply to I & J calculations; if a single float, is treated as a constant; if None, net to gross ratio data in the property collection is used
- **realization** (*int, optional*) – if present and the property collection is scanned for perm or ntg arrays, only those properties for this realization will be used; ignored if arrays passed in
- **darcy_constant** (*float, optional*) – if present, the value to use for the Darcy constant; if None, the grid's length units will determine the value as expected by Nexus
- **tolerance** (*float, default 1.0e-6*) – minimum half axis length below which the transmissibility will be deemed uncomputable (for the axis in question); NaN values will be returned (not Inf); units are implicitly those of the grid's crs length units; ignored if grid is a RegularGrid

Returns

numpy float array of shape (nk, nj, ni, 3) if grid is a RegularGrid otherwise (nk, nj, ni, 3, 2) where the 3 covers K,J,I and (for irregular grids) the 2 covers the face polarity: - (0) and + (1); units will depend on the length units of the coordinate reference system for the grid (and implicitly on the units of the darcy_constant); if darcy_constant is None, the units will be m3.cP/(kPa.d) or bbl.cP/(psi.d) for grid length units of m and ft respectively

Notes

calls either for half_cell_t_irregular() or half_cell_t_regular() depending on class of grid; see also notes for half_cell_t_irregular() and half_cell_t_regular(); prior to resqpy v4.8.0 the built in Darcy constant for metric units was two orders of magnitude too great (yielding transmissibilities in m3.cP/(bar.d) instead of m3.cP/(kPa.d)), sorry

resqpy.olio.transmission.half_cell_t_2d_triangular_precursor

resqpy.olio.transmission.**half_cell_t_2d_triangular_precursor**(*p*, *t*)

Creates a precursor to horizontal transmissibility for prism grids (see notes).

Parameters

- **p** (*numpy float array of shape (N, 2 or 3)*) – the xy(&z) locations of cell vertices
- **t** (*numpy int array of shape (M, 3)*) – the triangulation of p for which the transmissibility precursor is required

Returns

a pair of numpy float arrays, each of shape (M, 3) being the normal length and flow length relevant for flow across the face opposite each vertex as defined by t

Notes

this function acts as a precursor to the equivalent of the half cell transmissibility functions but for prism grids; for a resqpy VerticalPrismGrid, the triangulation can be shared by many layers with this function only needing to be called once; the first of the returned values (normal length) is the length of the triangle edge, in xy, when projected onto the normal of the flow direction; multiplying the normal length by a cell height will yield the area needed for transmissibility calculations; the second of the returned values (flow length) is the distance from the trangle centre to the midpoint of the edge and can be used as the distance term for a half cell transmissibilty; this function does not account for dip, it only handles the geometric aspects of half cell transmissibility in the xy plane

resqpy.olio.transmission.half_cell_t_irregular

resqpy.olio.transmission.**half_cell_t_irregular**(*grid*, *perm_k=None*, *perm_j=None*, *perm_i=None*, *ntg=None*, *darcy_constant=None*, *tolerance=1e-06*)

Creates a half cell transmissibilty property array for an IJK grid.

Parameters

- **grid** (*grid.Grid*) – the grid for which half cell transmissibilities are required
- **perm_k** (*float arrays of shape (nk, nj, ni), reqwuired*) – cell permeability values (for each direction), in mD;
- **j** (*float arrays of shape (nk, nj, ni), reqwuired*) – cell permeability values (for each direction), in mD;
- **i** (*float arrays of shape (nk, nj, ni), reqwuired*) – cell permeability values (for each direction), in mD;
- **ntg** (*float array of shape (nk, nj, ni), or float, required*) – net to gross values to apply to I & J calculations; if a single float, is treated as a constant;
- **darcy_constant** (*float, required*) – the value to use for the Darcy constant

- **tolerance** (*float, default 1.0e-6*) – minimum half axis length below which the transmissibility will be deemed uncomputable (for the axis in question); NaN values will be returned (not Inf); units are implicitly those of the grid's crs length units

Returns

numpy float array of shape (nk, nj, ni, 3, 2) where the 3 covers K,J,I and the 2 covers the face polarity: - (0) and + (1); units will depend on the length units of the coordinate reference system for the grid (and implicitly on the units of the darcy_constant); if darcy_constant is None, the units will typically be m3.cP/(kPa.d) or bbl.cP/(psi.d) for grid length units of m and ft respectively, depending on the correct darcy_constant being passed

Notes

the algorithm is equivalent to the half cell transmissibility element of the Nexus NEWTRAN calculation; each resulting value is effectively for the entire face, so area proportional fractions will be needed at faults with throw, or at grid boundaries; the net to gross factor is only applied to I & J transmissibilities, not K; the results include the Darcy Constant factor but not any transmissibility multiplier applied at the face; to compute the transmissibilty between neighbouring cells, take the harmonic mean of the two half cell transmissibilities and multiply by any transmissibility multiplier; if the two cells do not have simple sharing of a common face, first reduce each half cell transmissibility by the proportion of the face that is shared (which may be a different proportion for each of the two juxtaposed cells); returned array will need to be re-ordered and re-shaped before storing as a RESQML property with indexable elements of 'faces per cell'; the coordinate referemce system for the grid must have the same length units for xy and z

resqpy.olio.transmission.half_cell_t_regular

resqpy.olio.transmission.**half_cell_t_regular**(*grid*, *perm_k=None*, *perm_j=None*, *perm_i=None*, *ntg=None*, *darcy_constant=None*)

Creates a half cell transmissibilty property array for a RegularGrid.

Parameters

- **grid** (*grid.RegularGrid*) – the grid for which half cell transmissibilities are required
- **perm_k** (*float arrays of shape (nk, nj, ni), required*) – cell permeability values (for each direction), in mD;
- **j** (*float arrays of shape (nk, nj, ni), required*) – cell permeability values (for each direction), in mD;
- **i** (*float arrays of shape (nk, nj, ni), required*) – cell permeability values (for each direction), in mD;
- **ntg** (*float array of shape (nk, nj, ni), or float, optional*) – net to gross values to apply to I & J calculations; if a single float, is treated as a constant; if None, a value of 1.0 is used
- **darcy_constant** (*float, required*) – the value to use for the Darcy constant

Returns

numpy float array of shape (nk, nj, ni, 3) where the 3 covers K,J,I;
units will depend on the length units of the coordinate reference system for the grid (and implicitly on the units of the darcy_constant); the units will typically be m3.cP/(kPa.d) or bbl.cP/(psi.d) for grid length units of m and ft respectively, depending on the correct darcy_constant being passed

Notes

the same half cell transmissibility value is applicable to both - and + polarity faces; the axes of the regular grid are assumed to be orthogonal; the net to gross factor is only applied to I & J transmissibilities, not K; the results include the Darcy Constant factor but not any transmissibility multiplier applied at the face; to compute the transmissibilty between neighbouring cells, take the harmonic mean of the two half cell transmissibilities and multiply by any transmissibility multiplier; returned array will need to be re-shaped before storing as a RESQML property with indexable elements of 'faces'; the coordinate referemce system for the grid must have the same length units for xy and z; this function is vastly more computationally efficient than the general (irregular grid) function

resqpy.olio.transmission.half_cell_t_vertical_prism

resqpy.olio.transmission.**half_cell_t_vertical_prism**(*vpg*, *triple_perm_horizontal=None*, *perm_k=None*, *ntg=None*, *darcy_constant=None*, *tolerance=1e-06*)

Creates a half cell transmissibilty property array for a vertical prism grid.

Parameters

- **vpg** (*VerticalPrismGrid*) – the grid for which the half cell transmissibilities are required
- **triple_perm_horizontal** (*numpy float array of shape (N, 3)*) – the directional permeabilities to apply to each of the three vertical faces per cell
- **perm_k** (*numpy float array of shape (N,)*) – the permeability to use for the vertical transmissibilities
- **ntg** (*numpy float array of shape (N,), optional*) – if present, acts as a multiplier in the computation of non-vertical transmissibilities
- **darcy_constant** (*float, optional*) – the value to use for the Darcy constant; if None, a suitable value will be used depending on the length units of the vpg grid's crs
- **tolerance** (*float, default 1.0e-6*) – minimum half axis length below which the transmissibility will be deemed uncomputable (for the axis in question); NaN values will be returned (not Inf); units are implicitly those of the grid's crs length units

Returns

numpy float array of shape (N, 5) being the per-face half cell transmissibilities for each cell

Notes

order of 5 faces matches those of faces per cell, ie. top, base, then the 3 vertical faces; if no Darcy constant is provided, the returned values will have unite of m3.cP/(kPa.d) if the grid has length units of metres, or bbl.cP/(psi.d) if in feet

resqpy.olio.transmission.projected_tri_area

resqpy.olio.transmission.**projected_tri_area**(*pa*, *pb*, *pc*)

Return array holding areas of triangles projected onto each of yz, xz, xy.

Parameters

- **pa** (*numpy float array of shape (..., 3)*) – the corner points of a set of triangles (one corner in each of pa, pb & pc, for every triangle); last axis in arrays covers x,y,z
- **pb** (*numpy float array of shape (..., 3)*) – the corner points of a set of triangles (one corner in each of pa, pb & pc, for every triangle); last axis in arrays covers x,y,z
- **pc** (*numpy float array of shape (..., 3)*) – the corner points of a set of triangles (one corner in each of pa, pb & pc, for every triangle); last axis in arrays covers x,y,z

Returns

numpy float array of same shape as pa, pb & pc, with the last axis covering yz, xz, xy projections; the return values are the areas of the triangles projected in the three principal x,y,z axes

Note: assumes that units for x, y & z are the same; returned area units are those implicit units squared

7.18.22 resqpy.olio.triangulation

triangulation.py: functions for finding Delaunay triangulation and Voronoi graph from a set of points.

Functions

ccc	Returns the centre of the circumcircle of the three points in the xy plane.
dt	Returns the Delauney Triangulation of 2D point set p.
edges	Returns unique edges as pairs of point indices, and a count of uses of each edge.
internal_edges	Returns a subset of all edges where the edge count is 2.
make_all_clockwise_xy	Modifies t in situ such that each triangle is clockwise in xy plane (viewed from -ve z axis).
reorient	Returns a reoriented copy of a set of points, such that z axis is approximate normal to average plane of points.
rim_edges	Returns a subset of all edges where the edge count is 1.
rims	Returns edge index and points index lists of distinct rims.
surrounding_xy_ring	Creates a set of points surrounding the point set p, in the xy plane.
triangles_using_edge	Returns list-like 1D int array of indices of triangles using edge identified by pair of point indices.
triangles_using_edges	Returns int array of shape (len(edges), 2) with indices of upto 2 triangles using each edge (-1 for unused).
triangles_using_point	Returns list-like 1D int array of indices of triangles using vertex identified by point_index.
triangulated_polygons	Returns triangulation of polygons using centres as extra points.
voronoi	Returns dual Voronoi diagram for a Delauney triangulation.

resqpy.olio.triangulation.ccc

resqpy.olio.triangulation.**ccc**(*p1*, *p2*, *p3*)

Returns the centre of the circumcircle of the three points in the xy plane.

resqpy.olio.triangulation.dt

resqpy.olio.triangulation.**dt**(*p*, *algorithm='scipy'*, *plot_fn=None*, *progress_fn=None*, *container_size_factor=100.0*, *return_hull=False*)

Returns the Delauney Triangulation of 2D point set p.

Parameters

- **p** (*numpy float array of shape (N, 2)*) – the x,y coordinates of the points
- **algorithm** (*string, optional*) – selects which algorithm to use; current options: ['simple', 'scipy']; if None, the current best algorithm is selected
- **plot_fn** (*function of form f(p, t), optional*) – if present, this function is called each time the algorithm feels it is worth refreshing a plot of the progress; p is a copy of the point set, depending on the algorithm with 3 extra points added to form an enveloping triangle
- **progress_fn** (*function of form f(x), optional*) – if present, this function is called at regulat intervals by the algorithm, passing increasing values in the range 0.0 to 1.0 as x
- **container_size_factor** (*float, default 100.0*) – the larger this number, the more likely the resulting triangulation is to be convex; reduce to 1.0 to allow slight concavities
- **return_hull** (*boolean, default False*) – if True, a pair is returned with the second item being a clockwise ordered list of indices into p identifying the points on the boundary of the returned triangulation

Returns

numpy int array of shape (M, 3) - being the indices into the first axis of p of the 3 points per triangle in the Delauney Triangulation - and if return_hull is True, another int array of shape (B,) - being indices into p of the clockwise ordered points on the boundary of the triangulation

Notes

the plot_fn, progress_fn and container_size_factor arguments are only used by the 'simple' algorithm; if points p are 3D, the projection onto the xy plane is used for the triangulation

resqpy.olio.triangulation.edges

resqpy.olio.triangulation.**edges**(*t*)

Returns unique edges as pairs of point indices, and a count of uses of each edge.

Parameters

t (*numpy int array of shape (N, 3)*) – the points indices defining a set of triangles

Returns

(numpy int array of shape (N, 2), numpy int array of shape (N,)) where 2D array is list-like sorted points index pairs for unique edges and 1D array contains corresponding edge usage count (usually 1 or 2)

Notes

first entry in each pair is always the lower of the two point indices; for well formed surfaces, the count should everywhere be zero or one; the function does not attempt to detect coincident points

resqpy.olio.triangulation.internal_edges

resqpy.olio.triangulation.**internal_edges**(*all_edges*, *edge_counts*)

Returns a subset of all edges where the edge count is 2.

resqpy.olio.triangulation.make_all_clockwise_xy

resqpy.olio.triangulation.**make_all_clockwise_xy**(*t*, *p*)

Modifies t in situ such that each triangle is clockwise in xy plane (viewed from -ve z axis).

Note: assumes xyz axes are left handed; all will be made anti-clockwise in the case of right handed xyz axes

resqpy.olio.triangulation.reorient

resqpy.olio.triangulation.**reorient**(*points*, *rough=True*, *max_dip=None*, *use_linalg=True*, *sample=500*)

Returns a reoriented copy of a set of points, such that z axis is approximate normal to average plane of points.

Parameters

- **points** (*numpy float array of shape (..., 3)*) – the points to be reoriented
- **rough** (*bool, default True*) – if True, the resulting orientation will be within around 10 degrees of the optimum; if False, that reduces to around 2.5 degrees of the optimum; iugnored if use_linalg is True
- **max_dip** (*float, optional*) – if present, the reorientation of perspective off vertical is limited to this angle in degrees
- **use_linalg** (*bool, default True*) – if True, the numpy linear algebra svd function is used and rough is ignored
- **sample** (*int, default 500*) – downsample points to this number for the purposes of determining normal vector

Returns

numpy float array of the same shape as points, numpy xyz vector, numpy 3x3 matrix; the array being a copy of points rotated in 3D space to minimise the z range; the vector is a normal vector to the original points; the matrix is rotation matrix used to transform the original points to the reoriented points

Notes

the original points array is not modified by this function; implicit xy & z units for points are assumed to be the same; the function may typically be called prior to the Delauney triangulation, which uses an xy projection to determine the triangulation; the numpy linear algebra option seems to be memory intensive, not recommended; downsampling will occur (for normal vector determination) when the number of points exceeds double that given in the sample argument; set sample to None to use all points for normal vector determination

resqpy.olio.triangulation.rim_edges

resqpy.olio.triangulation.**rim_edges**(*all_edges*, *edge_counts*)

Returns a subset of all edges where the edge count is 1.

resqpy.olio.triangulation.rims

resqpy.olio.triangulation.**rims**(*all_rim_edges*)

Returns edge index and points index lists of distinct rims.

Parameters

all_rim_edges (*numpy int array of shape (N, 2)*) – edge point indices; as returned by rim_edges()

Returns

(list of arrays of rim edge indices, list of arrays of corresponding points indices) where arrays are 1D numpy int arrays; those of first list hold indices into rows of all_rim_edges; those of second list hold the corresponding points indices, both ordered in sequence of rim

resqpy.olio.triangulation.surrounding_xy_ring

resqpy.olio.triangulation.**surrounding_xy_ring**(*p*, *count=12*, *radial_factor=10.0*, *radial_distance=None*, *inner_ring=False*)

Creates a set of points surrounding the point set p, in the xy plane.

Parameters

- **p** (*numpy float array of shape (..., 3)*) – xyz set of points to be surrounded
- **count** (*int*) – the number of points to generate in the surrounding ring
- **radial_factor** (*float*) – a distance factor roughly determining the radius of the ring relative to the 'radius' of the outermost points in p
- **radial_distance** (*float*) – if present, the radius of the ring of points, unless radial_factor results in a greater distance in which case that is used
- **inner_ring** (*bool, default False*) – if True, an inner ring of points, with double count, is created at a radius just outside that of the furthest flung original point; this improves triangulation of the extended point set when the original has a non-convex hull

Returns

numpy float array of shape (N, 3) being xyz points in surrounding ring(s); z is set constant to mean value of z in p; N is count if inner_ring is False, 3 * count if True

resqpy.olio.triangulation.triangles_using_edge

resqpy.olio.triangulation.**triangles_using_edge**(*t*, *p1*, *p2*)

Returns list-like 1D int array of indices of triangles using edge identified by pair of point indices.

resqpy.olio.triangulation.triangles_using_edges

resqpy.olio.triangulation.**triangles_using_edges**(*t*, *edges*)

Returns int array of shape (len(edges), 2) with indices of upto 2 triangles using each edge (-1 for unused).

resqpy.olio.triangulation.triangles_using_point

resqpy.olio.triangulation.**triangles_using_point**(*t*, *point_index*)

Returns list-like 1D int array of indices of triangles using vertex identified by point_index.

resqpy.olio.triangulation.triangulated_polygons

resqpy.olio.triangulation.**triangulated_polygons**(*p*, *v*, *centres=None*)

Returns triangulation of polygons using centres as extra points.

Parameters

- **p** (*2D numpy float array*) – points used as vertices of polygons
- **v** (*list of list of ints*) – ordered indices into p for each polygon
- **centres** (*2D numpy float array, optional*) – the points to use as the centre for each polygon

Returns

points, triangles where – points is a copy of p extended with the centre points of polygons; and triangles is a numpy int array of shape (N, 3) being the triangulation of points, where N is equal to the overall length of v

Notes

if no centres are provided, balanced centre points are computed for the polygons; the polygons must be convex (at least from the perspective of the centre points); the clockwise/anti-clockwise order of the triangle edges will match that of the polygon; the centre point is the first point in each triangle; the order of triangles will match the order of vertices in a flattened view of list v; p and centres may have a shape of 2 or 3 in the second dimension (xy or xyz data); p & v could be the values (c, v) returned by the voronoi() function, in which case the original seed points p passed into voronoi() can be passed as centres here

resqpy.olio.triangulation.voronoi

resqpy.olio.triangulation.**voronoi**(*p*, *t*, *b*, *aoi: Polyline*)

Returns dual Voronoi diagram for a Delauney triangulation.

Parameters

- **p** (*numpy float array of shape (N, 2)*) – seed points used in the Delauney triangulation
- **t** (*numpy int array of shape (M, 3)*) – the Delauney triangulation of p as returned by dt()
- **b** (*numpy int array of shape (B,)*) – clockwise sorted list of indices into p of the boundary points of the triangulation t
- **aoi** (*lines.Polyline*) – area of interest; a closed clockwise polyline that must strictly contain all p (no points exactly on or outside the polyline)

Returns

c, v where – c is a numpy float array of shape (M+E, 2) being the circumcircle centres of the M triangles and E boundary points from the aoi polygon line; and v is a list of N Voronoi cell lists of clockwise ints, each int being an index into c

Notes

the aoi polyline forms the outer boundary for the Voronoi polygons for points on the outer edge of the triangulation; all points p must lie strictly within the aoi, which must be convex; the triangulation t, of points p, must also have a convex hull; note that the dt() function can produce a triangulation with slight concavities on the hull, especially for smaller values of its container_size_factor argument

7.18.23 resqpy.olio.uuid

uuid.py: Thin wrapper around python uuid (universally unique identifier) module.

Functions

is_uuid	Returns boolean indicating whether uuid_obj seems to be a uuid, in any allowed form.
matching_uuids	Returns True if the 2 uuid objects are for the same id; False otherwise.
new_uuid	Returns a new uuid based on the time (to 100ns) & MAC address option of the iso standard.
string_from_uuid	Returns standard hexadecimal string for uuid; same as str(uuid_obj).
switch_off_test_mode	Subsequent calls to new_uid() will produce standard uuid values (default behaviour).
switch_on_test_mode	Causes subsequent calls to new_uid() to produce integer sequence starting from successor to seed.
uuid_as_bytes	Returns the uuid as a 16 byte bytes sequence; same as uuid_obj.bytes.
uuid_as_int	Returns the uuid as a 128 bit int; same as uuid_obj.int.
uuid_from_int	Returns a uuid object for the given uuid int.
uuid_from_string	Returns a uuid object for the given uuid string; hyphens are ignored.
uuid_in_list	Returns True if the uuid is in the list of uuids.
version_string	Returns an integer string rendering of the time element of the uuid.

resqpy.olio.uuid.is_uuid

resqpy.olio.uuid.**is_uuid**(*uuid_obj*)

Returns boolean indicating whether uuid_obj seems to be a uuid, in any allowed form.

resqpy.olio.uuid.matching_uuids

resqpy.olio.uuid.**matching_uuids**(*uuid_a*, *uuid_b*)

Returns True if the 2 uuid objects are for the same id; False otherwise.

Parameters

- **uuid_a** (*uuid.UUID objects*) – the two uuids to be compared
- **uuid_b** (*uuid.UUID objects*) – the two uuids to be compared

Returns

boolean – True if the two uuids are the same; False otherwise

Note: this function is resilient to uuids being passed in hexadecimal string format, or int

resqpy.olio.uuid.new_uuid

resqpy.olio.uuid.**new_uuid**()

Returns a new uuid based on the time (to 100ns) & MAC address option of the iso standard.

Returns
uuid.UUID object

Notes

at present, the multi-processor safe functionality is not deployed, so multiple processors sharing the same MAC address could generate the same uuid simultaneously; an integer sequence is generated when in test mode

resqpy.olio.uuid.string_from_uuid

resqpy.olio.uuid.**string_from_uuid**(*uuid_obj*)

Returns standard hexadecimal string for uuid; same as str(uuid_obj).

Parameters
uuid_obj (*uuid.UUID object*) – the uuid which is required in hexadecimal string format

Returns
string (36 characters – 32 lowercase hexadecimals and 4 hyphens)

resqpy.olio.uuid.switch_off_test_mode

resqpy.olio.uuid.**switch_off_test_mode**()

Subsequent calls to new_uid() will produce standard uuid values (default behaviour).

Note: this function will have no effect unless switch_on_test_mode() has previously been called

resqpy.olio.uuid.switch_on_test_mode

resqpy.olio.uuid.**switch_on_test_mode**(*seed=0*)

Causes subsequent calls to new_uid() to produce integer sequence starting from successor to seed.

Parameters
seed (*integer, default 0*) – The predecessor to the first uuid returned by subsequent calls to new_uuid()

Returns
None

Notes

call switch_off_test_mode() to reactivate normal behaviour; uuids generated whilst in test mode do not adhere to the iso standard; test mode is intended to allow replicatable behaviour for testing purposes

resqpy.olio.uuid.uuid_as_bytes

resqpy.olio.uuid.**uuid_as_bytes**(*uuid_obj*)

Returns the uuid as a 16 byte bytes sequence; same as uuid_obj.bytes.

Parameters
uuid_obj (*uuid.UUID object*) – the uuid for which a bytes representation is required

Returns
bytes (16 bytes long)

resqpy.olio.uuid.uuid_as_int

resqpy.olio.uuid.**uuid_as_int**(*uuid_obj*)

Returns the uuid as a 128 bit int; same as uuid_obj.int.

Parameters
uuid_obj (*uuid.UUID object*) – the uuid for which a bytes representation is required

Returns
int (128 bit, though python no longer differentiates int precision)

resqpy.olio.uuid.uuid_from_int

resqpy.olio.uuid.**uuid_from_int**(*uuid_int*)

Returns a uuid object for the given uuid int.

resqpy.olio.uuid.uuid_from_string

resqpy.olio.uuid.**uuid_from_string**(*uuid_str*)

Returns a uuid object for the given uuid string; hyphens are ignored.

Parameters
uuid_str (*string*) – the hexadecimal representation of the 128 bit uuid integer; hyphens are ignored

Returns
uuid.UUID object

Notes

if a uuid.UUID object is passed by accident, it is returned; if the string starts with an underscore, the underscore is skipped (to cater for a fesapi quirk); any tail beyond the uuid string is ignored

resqpy.olio.uuid.uuid_in_list

resqpy.olio.uuid.**uuid_in_list**(*uuid*, *uuid_list*)

Returns True if the uuid is in the list of uuids.

resqpy.olio.uuid.version_string

resqpy.olio.uuid.**version_string**(*uuid_obj*)

Returns an integer string rendering of the time element of the uuid.

Parameters

uuid_obj (*uuid.UUID*) – the uuid for which a string representation of the time component is required

Returns

string (of digits)

Notes

this function has nothing to do with the uuid.version attribute, it is used to populate the version field of a resqml citation block; the time component of the uuid is the number of 100ns periods that have elapsed since October 1582 (when the Gregorian calendar was adopted), as a 60 bit integer

7.18.24 resqpy.olio.vdb

vdb.py: Module providing functions for reading from VDB datasets.

Classes

Data	Internal class for handling Data records in a vdb file.
Fragment	Internal class for handling an individual Fragment record in a vdb file.
FragmentChain	Internal class for handling a chain of Fragment records in a vdb file.
FragmentHeader	Internal class for handling a Fragment Header record in a vdb file.
Header	Internal class for handling a Header record in a vdb file.
KP	Internal class for a (Key, Pointer) record in a vdb file.
RawData	Internal class for handling Raw Data records in a vdb file.
VDB	Class for handling a vdb, particularly to support import of grid and properties.

resqpy.olio.vdb.Data

class resqpy.olio.vdb.**Data**(*fp*, *header*)

Bases: object

Internal class for handling Data records in a vdb file.

Methods:

__init__(fp, header)	Creates a new Data object.

__init__(*fp*, *header*)

Creates a new Data object.

resqpy.olio.vdb.Fragment

class resqpy.olio.vdb.**Fragment**(*fp*, *place*, *header*)

Bases: object

Internal class for handling an individual Fragment record in a vdb file.

Methods:

__init__(fp, place, header)	Creates a new Fragment record object.

__init__(*fp*, *place*, *header*)

Creates a new Fragment record object.

resqpy.olio.vdb.FragmentChain

class resqpy.olio.vdb.**FragmentChain**(*fp*, *place*, *header*)

Bases: object

Internal class for handling a chain of Fragment records in a vdb file.

Methods:

__init__(fp, place, header)	Creates a new Fragment Chain object.

__init__(*fp*, *place*, *header*)

Creates a new Fragment Chain object.

resqpy.olio.vdb.FragmentHeader

class resqpy.olio.vdb.**FragmentHeader**(*fp*, *place*)

Bases: object

Internal class for handling a Fragment Header record in a vdb file.

Methods:

__init__(fp, place)	Creates a new Fragment Header record object.

__init__(*fp*, *place*)

Creates a new Fragment Header record object.

resqpy.olio.vdb.Header

class resqpy.olio.vdb.**Header**(*fp*, *place*)

Bases: object

Internal class for handling a Header record in a vdb file.

Methods:

__init__(fp, place)	Creates a new Header record object.

__init__(*fp*, *place*)

Creates a new Header record object.

resqpy.olio.vdb.KP

class resqpy.olio.vdb.**KP**(*fp*, *place=4*)

Bases: object

Internal class for a (Key, Pointer) record in a vdb file.

Methods:

__init__(fp[, place])	Creates a new (Key, Pointer) record object.
header_place_for_key(key[, search])	Returns file position pointer for a given key.
head_for_key(key[, search])	Returns a Header record object for the given key.
data_for_key(key[, search])	Returns a Data object for the given key.
key_list([filter])	Returns a list of keys.
sub_key_list(keyword[, filter])	Returns a list of keys subordinate to keyword.

__init__(*fp*, *place=4*)

Creates a new (Key, Pointer) record object.

header_place_for_key(*key*, *search=False*)

Returns file position pointer for a given key.

head_for_key(*key*, *search=False*)

Returns a Header record object for the given key.

data_for_key(*key*, *search=False*)

Returns a Data object for the given key.

key_list(*filter=False*)

Returns a list of keys.

sub_key_list(*keyword*, *filter=False*)

Returns a list of keys subordinate to keyword.

resqpy.olio.vdb.RawData

class resqpy.olio.vdb.**RawData**(*fp*, *place*, *item_type*, *count*, *max_count*)

Bases: object

Internal class for handling Raw Data records in a vdb file.

Methods:

__init__(fp, place, item_type, count, max_count)	Creates a new Raw Data record object.

__init__(*fp*, *place*, *item_type*, *count*, *max_count*)

Creates a new Raw Data record object.

resqpy.olio.vdb.VDB

class resqpy.olio.vdb.**VDB**(*path*)

Bases: object

Class for handling a vdb, particularly to support import of grid and properties.

Methods:

__init__(path)	Initialises a VDB object and associates it with the given vdb directory path.
cases()	Returns a list of simulation case strings as found in the main xml file.
set_use_case(case)	Sets the simulation case to use in other functions.
print_header_tree(relative_path)	Low level: prints out the raw header tree found in the vdb file (for debugging).

continues on next page

Table 5 – continued from previous page

print_key_tree(relative_path)	Low level: prints out the keyword tree found in the vdb file.
data_for_keyword(relative_path, keyword[, ...])	Reads data associated with a keyword from a vdb binary file; returns a numpy array (or string).
data_for_keyword_chain(relative_path, ...)	Follows a list of keywords down through hierarchy and returns the data as a numpy array (or string).
set_extent_kji(extent_kji[, use_case, grid_name])	Sets extent for one use case (defaults to current use case) as alternative to processing corp data.
fetch_corp_patch(relative_path)	Loads one patch of grid corp data from one file in the vdb; returns (number of cells, 1D array).
list_of_grids()	Returns a list of grid names for which corp data exists in the vdb for the current use case.
root_corp()	Loads root grid corp data from vdb; returns pagoda style resequenced 7D numpy array of doubles.
grid_corp(grid_name)	Loads corp data for named grid from vdb; returns pagoda style resequenced 7D numpy array of doubles.
load_init_mapdata_array(file, keyword[, ...])	Loads an INIT MAPDATA array from vdb; returns 3D numpy array coerced to dtype (if not None).
load_recurrent_mapdata_array(file, keyword)	Loads a RECUR MAPDATA array from vdb; returns 3D numpy array coerced to dtype (if not None).
root_dad()	Loads and returns the IROOTDAD array from vdb; returns 3D numpy int32 array.
grid_dad(grid_name)	Loads and returns the DAD array from vdb for the named grid; returns 3D numpy int32 array.
root_kid()	Loads and returns the IROOTKID array from vdb; returns 3D numpy int32 array (can be inactive cell mask).
grid_kid(grid_name)	Loads and returns the IROOTKID array from vdb; returns 3D numpy int32 array (can be inactive cell mask).
root_kid_inactive_mask()	Loads the IROOTKID array and returns boolean mask of cells inactive in ROOT grid.
grid_kid_inactive_mask(grid_name)	Loads the KID array for the named grid and returns boolean mask of cells inactive in grid.
root_uid()	Loads and returns the IROOTUID array from vdb; returns 3D numpy int32 array.
grid_uid(grid_name)	Loads and returns the UID array from vdb for the named grid; returns 3D numpy int32 array.
root_unpack()	Loads and returns the IROOTUNPACK array from vdb; returns 3D numpy int32 array.
grid_unpack(grid_name)	Loads and returns the IROOTUNPACK array from vdb; returns 3D numpy int32 array.
list_of_static_properties()	Returns list of static property keywords present in the vdb for ROOT.
grid_list_of_static_properties(grid_name)	Returns list of static property keywords present in the vdb for named grid.
root_static_property(keyword[, dtype, unpack])	Loads and returns a ROOT static property array.
grid_static_property(grid_name, keyword[, ...])	Loads and returns a static property array for named grid.

continues on next page

Table 5 – continued from previous page

list_of_timesteps()	Returns a list of integer timesteps for which a ROOT recurrent mapdata file exists.
grid_list_of_timesteps(grid_name)	Returns a list of integer timesteps for which a recurrent mapdata file for the named grid exists.
list_of_recurrent_properties(timestep)	Returns list of recurrent property keywords present in the vdb for given timestep.
grid_list_of_recurrent_properties(grid_name,...)	Returns list of recurrent property keywords present in the vdb for named grid for given timestep.
root_recurrent_property_for_timestep(...[,...])	Loads and returns a ROOT recurrent property array for one timestep.
grid_recurrent_property_for_timestep(...[,...])	Loads and returns a recurrent property array for named grid for one timestep.
header_place_for_keyword(relative_path, keyword)	Low level function to return file position for header relating to given keyword.
root_shaped(a)	Returns array reshaped to root grid extent for current use case, if known; otherwise unchanged.
grid_shaped(grid_name, a)	Returns array reshaped to named grid extent for current use case, if known; otherwise unchanged.
zip_glob(path_with_asterisk)	Performs glob.glob like function for zipped file, path must contain a single asterisk.

__init__(*path*)

Initialises a VDB object and associates it with the given vdb directory path.

cases()

Returns a list of simulation case strings as found in the main xml file.

set_use_case(*case*)

Sets the simulation case to use in other functions.

print_header_tree(*relative_path*)

Low level: prints out the raw header tree found in the vdb file (for debugging).

print_key_tree(*relative_path*)

Low level: prints out the keyword tree found in the vdb file.

data_for_keyword(*relative_path*, *keyword*, *search=True*)

Reads data associated with a keyword from a vdb binary file; returns a numpy array (or string).

data_for_keyword_chain(*relative_path*, *keyword_chain*)

Follows a list of keywords down through hierarchy and returns the data as a numpy array (or string).

set_extent_kji(*extent_kji*, *use_case=None*, *grid_name='ROOT'*)

Sets extent for one use case (defaults to current use case) as alternative to processing corp data.

fetch_corp_patch(*relative_path*)

Loads one patch of grid corp data from one file in the vdb; returns (number of cells, 1D array).

list_of_grids()

Returns a list of grid names for which corp data exists in the vdb for the current use case.

root_corp()

Loads root grid corp data from vdb; returns pagoda style resequenced 7D numpy array of doubles.

grid_corp(*grid_name*)

Loads corp data for named grid from vdb; returns pagoda style resequenced 7D numpy array of doubles.

load_init_mapdata_array(*file*, *keyword*, *dtype=None*, *unpack=False*, *grid_name='ROOT'*)

Loads an INIT MAPDATA array from vdb; returns 3D numpy array coerced to dtype (if not None).

load_recurrent_mapdata_array(*file*, *keyword*, *dtype=None*, *unpack=False*, *grid_name='ROOT'*)

Loads a RECUR MAPDATA array from vdb; returns 3D numpy array coerced to dtype (if not None).

root_dad()

Loads and returns the IROOTDAD array from vdb; returns 3D numpy int32 array.

grid_dad(*grid_name*)

Loads and returns the DAD array from vdb for the named grid; returns 3D numpy int32 array.

root_kid()

Loads and returns the IROOTKID array from vdb; returns 3D numpy int32 array (can be inactive cell mask).

grid_kid(*grid_name*)

Loads and returns the IROOTKID array from vdb; returns 3D numpy int32 array (can be inactive cell mask).

root_kid_inactive_mask()

Loads the IROOTKID array and returns boolean mask of cells inactive in ROOT grid.

grid_kid_inactive_mask(*grid_name*)

Loads the KID array for the named grid and returns boolean mask of cells inactive in grid.

root_uid()

Loads and returns the IROOTUID array from vdb; returns 3D numpy int32 array.

grid_uid(*grid_name*)

Loads and returns the UID array from vdb for the named grid; returns 3D numpy int32 array.

root_unpack()

Loads and returns the IROOTUNPACK array from vdb; returns 3D numpy int32 array.

grid_unpack(*grid_name*)

Loads and returns the IROOTUNPACK array from vdb; returns 3D numpy int32 array.

list_of_static_properties()

Returns list of static property keywords present in the vdb for ROOT.

grid_list_of_static_properties(*grid_name*)

Returns list of static property keywords present in the vdb for named grid.

root_static_property(*keyword*, *dtype=None*, *unpack=None*)

Loads and returns a ROOT static property array.

grid_static_property(*grid_name*, *keyword*, *dtype=None*, *unpack=None*)

Loads and returns a static property array for named grid.

list_of_timesteps()

Returns a list of integer timesteps for which a ROOT recurrent mapdata file exists.

grid_list_of_timesteps(*grid_name*)

Returns a list of integer timesteps for which a recurrent mapdata file for the named grid exists.

list_of_recurrent_properties(*timestep*)

Returns list of recurrent property keywords present in the vdb for given timestep.

grid_list_of_recurrent_properties(*grid_name*, *timestep*)

Returns list of recurrent property keywords present in the vdb for named grid for given timestep.

root_recurrent_property_for_timestep(*keyword*, *timestep*, *dtype=None*, *unpack=True*)

Loads and returns a ROOT recurrent property array for one timestep.

grid_recurrent_property_for_timestep(*grid_name*, *keyword*, *timestep*, *dtype=None*, *unpack=True*)

Loads and returns a recurrent property array for named grid for one timestep.

header_place_for_keyword(*relative_path*, *keyword*, *search=True*)

Low level function to return file position for header relating to given keyword.

root_shaped(*a*)

Returns array reshaped to root grid extent for current use case, if known; otherwise unchanged.

grid_shaped(*grid_name*, *a*)

Returns array reshaped to named grid extent for current use case, if known; otherwise unchanged.

zip_glob(*path_with_asterisk*)

Performs glob.glob like function for zipped file, path must contain a single asterisk.

Functions

bad_keyword	Return False if key is a valid keyword, otherwise True.
coerce	Returns a version of numpy array a with elements coerced to dtype.
ensemble_vdb_list	Returns a sorted list of vdb paths found in the directory tree under run_dir.

resqpy.olio.vdb.bad_keyword

resqpy.olio.vdb.**bad_keyword**(*key*)

Return False if key is a valid keyword, otherwise True.

resqpy.olio.vdb.coerce

resqpy.olio.vdb.**coerce**(*a*, *dtype*)

Returns a version of numpy array a with elements coerced to dtype.

resqpy.olio.vdb.ensemble_vdb_list

resqpy.olio.vdb.**ensemble_vdb_list**(*run_dir*, *sort_list=True*)

Returns a sorted list of vdb paths found in the directory tree under run_dir.

7.18.25 resqpy.olio.vector_utilities

Utilities for working with 3D vectors in cartesian space.

note: some of these functions are redundant as they are provided by built-in numpy operations.

a vector is a one dimensional numpy array with 3 elements: x, y, z. some functions accept a tuple or list of 3 elements as an alternative to a numpy array.

Functions

add	Returns vector sum a+b.
amplify	Returns vector with direction of v, amplified by scaling.
area_of_triangle	Returns the area of the triangle defined by three vertices.
area_of_triangles	Returns numpy array of areas of triangles, optionally when projected onto xy plane.
azimuth	Returns the compass bearing in degrees of the direction of v (x = East, y = North), ignoring z.
azimuths	Returns the compass bearings in degrees of the direction of each vector in va (x = East, y = North), ignoring z.
clockwise	Returns a +ve value if 2D points a,b,c are in clockwise order, 0.0 if in line, -ve for ccw.
clockwise_sorted_indices	Returns a clockwise sorted numpy list of indices b into the points p.
clockwise_triangles	Returns a numpy array of +ve values where triangle points are in clockwise order, 0.0 if in line, -ve for ccw.
cross_product	Returns the cross product (vector product) of the two vectors.
degrees_difference	Returns the angle between two vectors, in degrees.
degrees_from_radians	Converts angle from radians to degrees.
determinant	Returns the determinant of the 3 x 3 matrix comprised of the 3 vectors.
determinant_3x3	Returns the determinant of the 3 x 3 matrix.
dot_product	Returns the dot product (scalar product) of the two vectors.
dot_products	Returns the dot products of pairs of vectors; last axis covers element of a vector.
elemental_multiply	Returns vector with products of corresponding elements of a and b.
in_circumcircle	Returns True if point d lies within the circumcircle pf ccw points a, b, c, projected onto xy plane.
in_triangle	Returns True if point d lies wholly within the triangle pf ccw points a, b, c, projected onto xy plane.
in_triangle_edged	Returns True if d lies within or on the boudnary of triangle of ccw points a,b,c projected onto xy plane.

continues on next page

Table 6 – continued from previous page

inclination	Returns the inclination in degrees of v (angle relative to +ve z axis).
inclinations	Returns the inclination in degrees of each vector in a (angle relative to +ve z axis).
is_close	Returns True if the two points are extremely close to one another (ie.
is_obtuse_2d	Returns True if the angle at point p subtended by points p1 and p2, in xy plane, is greater than 90 degrees; else False.
isclose	Returns True if the two points are extremely close to one another (ie.
manhattan_distance	Returns the Manhattan distance between two points.
manhatten_distance	Returns the Manhattan distance between two points.
mesh_points_in_triangle	Calculates which implicit mesh points are within a triangle in 2D for normalised triangle.
meshgrid	Returns coordinate matrices from coordinate vectors x and y.
naive_2d_length	Returns the length of the vector projected onto xy plane, assuming consistent units.
naive_2d_lengths	Returns the lengths of the vectors projected onto xy plane, assuming consistent units.
naive_length	Returns the length of the vector assuming consistent units.
naive_lengths	Returns the lengths of the vectors assuming consistent units.
nan_inclinations	Returns the inclination in degrees of each vector in a (angle relative to +ve z axis), allowing NaNs.
nan_unit_vectors	Returns vectors with same direction as those in v but with unit length, allowing NaNs.
nearest_point_projected	Returns the index into points array closest to point p; projection is 'xy', 'xz' or 'yz'.
no_rotation_matrix	Returns a rotation matrix which will not move points (identity matrix).
perspective_vector	Returns a version of vector with a perspective applied.
point_distance_sqr_to_points_projected	Returns an array of projected distances squared between p and points; projection is 'xy', 'xz' or 'yz'.
point_distance_to_line_2d	Ignoring any z values, returns the xy distance of point p from line passing through l1 and l2.
point_distance_to_line_segment_2d	Ignoring any z values, returns the xy distance of point p from line segment between l1 and l2.
point_in_polygon	Calculates if a point in within a polygon in 2D.
point_in_triangle	Calculates if a point in within a triangle in 2D.
points_direction_vector	Returns an average direction vector based on first and last non-NaN points or slices in given axis.
points_in_polygon	Calculates which points are within a polygon in 2D.
points_in_polygons	Calculates which points are within which polygons in 2D.
points_in_triangle	Calculates which points are within a triangle in 2D.
points_in_triangles	Returns 2D numpy bool array indicating which of points da are within which triangles.

continues on next page

Table 6 – continued from previous page

points_in_triangles_aligned	Calculates which points are within which triangles in 2D for a regular mesh of aligned points.
points_in_triangles_aligned_optimised	Calculates which points are within which triangles in 2D for a regular mesh of aligned points.
points_in_triangles_njit	Calculates which points are within which triangles in 2D.
project_points_onto_plane	Modifies array of xyz points in situ to project onto a plane defined by a point and normal vector.
radians_difference	Returns the angle between two vectors, in radians.
radians_from_degrees	Converts angle from degrees to radians.
reverse_rotation_3d_matrix	Returns a rotation matrix which will rotate points about the y, z, then x axis by angles in degrees.
rotate_array	Returns a copy of array a with each vector rotated by the rotation matrix.
rotate_array_njit	Returns a copy of array a with each vector rotated by the rotation matrix.
rotate_vector	Returns the rotated vector.
rotate_xyz_array_around_z_axis	Returns a copy of array a suitable for presenting a cross-section using the resulting x,z values.
rotation_3d_matrix	Returns a rotation matrix which will rotate points about the x, z, then y axis by angles in degrees.
rotation_3d_matrix_njit	Returns a rotation matrix which will rotate points about the x, z, then y axis by angles in degrees.
rotation_matrix_3d_axial	Returns a rotation matrix which will rotate points about axis (0: x, 1: y, or 2: z) by angle in degrees.
rotation_matrix_3d_vector	Returns a rotation matrix which will rotate vector v to the vertical (z) axis.
rotation_matrix_3d_vector_njit	Returns a rotation matrix which will rotate points by inclination and azimuth of vector.
subtract	Returns vector difference a-b.
tilt_3d_matrix	Returns a 3D rotation matrix for applying a dip in a certain azimuth.
tilt_points	Modifies array of xyz points in situ to apply dip in direction of azimuth, about pivot point.
triangle_box	Finds the minimum and maximum x and y values of a single traingle.
triangle_normal_vector	For a triangle in 3D space, defined by 3 vertex points, returns a unit vector normal to the plane of the triangle.
triangle_normal_vector_numba	For a triangle in 3D space, defined by 3 vertex points, returns a unit vector normal to the plane of the triangle.
unit_corrected_length	Returns the length of the vector v after applying the unit_conversion factors.
unit_vector	Returns vector with same direction as v but with unit length.
unit_vector_from_azimuth	Returns horizontal unit vector in compass bearing given by azimuth (x = East, y = North).
unit_vector_from_azimuth_and_inclination	Returns unit vector with compass bearing of azimuth and inclination off +z axis.
unit_vector_njit	Returns vector with same direction as v but with unit length.

continues on next page

Table 6 – continued from previous page

unit_vectors	Returns vectors with same direction as those in v but with unit length.
v_3d	Returns a 3D vector for a 2D or 3D vector.
vertical_intercept	Finds the y value of a straight line between two points at a given x.
xy_sorted	Returns copy of points p sorted according to x or y (whichever has greater range).
xy_sorted_njit	Returns copy of points p sorted according to x or y (whichever has greater range).
zero_vector	Returns a zero vector [0.0, 0.0, 0.0].

resqpy.olio.vector_utilities.add

resqpy.olio.vector_utilities.**add**(*a*, *b*)

Returns vector sum a+b.

resqpy.olio.vector_utilities.amplify

resqpy.olio.vector_utilities.**amplify**(*v*, *scaling*)

Returns vector with direction of v, amplified by scaling.

resqpy.olio.vector_utilities.area_of_triangle

resqpy.olio.vector_utilities.**area_of_triangle**(*a*, *b*, *c*)

Returns the area of the triangle defined by three vertices.

resqpy.olio.vector_utilities.area_of_triangles

resqpy.olio.vector_utilities.**area_of_triangles**(*p*, *t*, *xy_projection=False*)

Returns numpy array of areas of triangles, optionally when projected onto xy plane.

resqpy.olio.vector_utilities.azimuth

resqpy.olio.vector_utilities.**azimuth**(*v*)

Returns the compass bearing in degrees of the direction of v (x = East, y = North), ignoring z.

resqpy.olio.vector_utilities.azimuths

resqpy.olio.vector_utilities.**azimuths**(*va*)

Returns the compass bearings in degrees of the direction of each vector in va (x = East, y = North), ignoring z.

resqpy.olio.vector_utilities.clockwise

resqpy.olio.vector_utilities.**clockwise**(*a*, *b*, *c*)

Returns a +ve value if 2D points a,b,c are in clockwise order, 0.0 if in line, -ve for ccw.

Note: assumes positive y-axis is anticlockwise from positive x-axis

resqpy.olio.vector_utilities.clockwise_sorted_indices

resqpy.olio.vector_utilities.**clockwise_sorted_indices**(*p*, *b*)

Returns a clockwise sorted numpy list of indices b into the points p.

Note: this function is designed for preparing a list of points defining a convex polygon when projected in the xy plane, starting from a subset of the unsorted points; more specifically, it assumes that the mean of p (over axis 0) lies within the polygon and the clockwise ordering is relative to that mean point

resqpy.olio.vector_utilities.clockwise_triangles

resqpy.olio.vector_utilities.**clockwise_triangles**(*p*, *t*, *projection='xy'*)

Returns a numpy array of +ve values where triangle points are in clockwise order, 0.0 if in line, -ve for ccw.

Parameters

- **p** (*numpy float array of shape (N, 2 or 3)*) – points in use as vertices of triangles
- **t** (*numpy int array of shape (M, 3)*) – indices into first axis of p defining the triangles
- **projection** (*string, default 'xy'*) – one of 'xy', 'xz' or 'yz' being the direction of projection, ie. which elements of the second axis of p to use; must be 'xy' if p has shape (N, 2)

Returns

numpy float array of shape (M,) being the +ve or -ve values indicating clockwise or anti-clockwise ordering of each triangle's vertices when projected onto the specified plane and viewed in the direction negative to positive of the omitted axis

Note: assumes xyz axes are left-handed (reverse the result for a right handed system)

resqpy.olio.vector_utilities.cross_product

resqpy.olio.vector_utilities.**cross_product**(*a*, *b*)

Returns the cross product (vector product) of the two vectors.

resqpy.olio.vector_utilities.degrees_difference

resqpy.olio.vector_utilities.**degrees_difference**(*a*, *b*)

Returns the angle between two vectors, in degrees.

resqpy.olio.vector_utilities.degrees_from_radians

resqpy.olio.vector_utilities.**degrees_from_radians**(*rad*)

Converts angle from radians to degrees.

resqpy.olio.vector_utilities.determinant

resqpy.olio.vector_utilities.**determinant**(*a*, *b*, *c*)

Returns the determinant of the 3 x 3 matrix comprised of the 3 vectors.

resqpy.olio.vector_utilities.determinant_3x3

resqpy.olio.vector_utilities.**determinant_3x3**(*a*)

Returns the determinant of the 3 x 3 matrix.

resqpy.olio.vector_utilities.dot_product

resqpy.olio.vector_utilities.**dot_product**(*a*, *b*)

Returns the dot product (scalar product) of the two vectors.

resqpy.olio.vector_utilities.dot_products

resqpy.olio.vector_utilities.**dot_products**(*a*, *b*)

Returns the dot products of pairs of vectors; last axis covers element of a vector.

resqpy.olio.vector_utilities.elemental_multiply

resqpy.olio.vector_utilities.**elemental_multiply**(*a*, *b*)

Returns vector with products of corresponding elements of a and b.

resqpy.olio.vector_utilities.in_circumcircle

resqpy.olio.vector_utilities.**in_circumcircle**(*a*, *b*, *c*, *d*)

Returns True if point d lies within the circumcircle pf ccw points a, b, c, projected onto xy plane.

Note: a, b & c must be sorted into anti-clockwise order before calling this function

resqpy.olio.vector_utilities.in_triangle

resqpy.olio.vector_utilities.**in_triangle**(*a*, *b*, *c*, *d*)

Returns True if point d lies wholly within the triangle pf ccw points a, b, c, projected onto xy plane.

Note: a, b & c must be sorted into anti-clockwise order before calling this function

resqpy.olio.vector_utilities.in_triangle_edged

resqpy.olio.vector_utilities.**in_triangle_edged**(*a*, *b*, *c*, *d*)

Returns True if d lies within or on the boudnary of triangle of ccw points a,b,c projected onto xy plane.

Note: a, b & c must be sorted into anti-clockwise order before calling this function

resqpy.olio.vector_utilities.inclination

resqpy.olio.vector_utilities.**inclination**(*v*)

Returns the inclination in degrees of v (angle relative to +ve z axis).

resqpy.olio.vector_utilities.inclinations

resqpy.olio.vector_utilities.**inclinations**(*a*)

Returns the inclination in degrees of each vector in a (angle relative to +ve z axis).

resqpy.olio.vector_utilities.is_close

resqpy.olio.vector_utilities.**is_close**(*a*, *b*, *tolerance=1e-06*)

Returns True if the two points are extremely close to one another (ie.

the same point).

resqpy.olio.vector_utilities.is_obtuse_2d

resqpy.olio.vector_utilities.**is_obtuse_2d**(*p*, *p1*, *p2*)

Returns True if the angle at point p subtended by points p1 and p2, in xy plane, is greater than 90 degrees; else False.

resqpy.olio.vector_utilities.isclose

resqpy.olio.vector_utilities.**isclose**(*a*, *b*, *tolerance=1e-06*)

Returns True if the two points are extremely close to one another (ie.

the same point).

resqpy.olio.vector_utilities.manhattan_distance

resqpy.olio.vector_utilities.**manhattan_distance**(*p1*, *p2*)

Returns the Manhattan distance between two points.

resqpy.olio.vector_utilities.manhatten_distance

resqpy.olio.vector_utilities.**manhatten_distance**(*p1*, *p2*)

Returns the Manhattan distance between two points.

resqpy.olio.vector_utilities.mesh_points_in_triangle

resqpy.olio.vector_utilities.**mesh_points_in_triangle**(*triangle: ndarray*, *points_xlen: int*, *points_ylen: int*, *triangle_num: int = 0*) → ndarray

Calculates which implicit mesh points are within a triangle in 2D for normalised triangle.

Parameters

- **triangle** (*np.ndarray*) – array of the triangle's vertices in 2D, shape (3, 2).
- **points_xlen** (*int*) – the number of unique x coordinates, starting at 0.0, spacing 1.0.
- **points_ylen** (*int*) – the number of unique y coordinates, starting at 0.0, spacing 1.0.
- **triangle_num** (*int*) – the triangle number, default is 0.

Returns

triangle_points (np.ndarray) –

2D array containing only the points within the triangle,
with each row being the triangle number, points y index, and points x index.

resqpy.olio.vector_utilities.meshgrid

resqpy.olio.vector_utilities.**meshgrid**(*x: ndarray*, *y: ndarray*) → Tuple[ndarray, ndarray]

Returns coordinate matrices from coordinate vectors x and y.

Parameters

- **x** (*np.ndarray*) – 1d array of x coordinates.
- **y** (*np.ndarray*) – 1d array of y coordinates.

Returns

Tuple containing – - xx (np.ndarray): the elements of x repeated to fill the matrix along the first dimension. - yy (np.ndarray): the elements of y repeated to fill the matrix along the second dimension.

resqpy.olio.vector_utilities.naive_2d_length

resqpy.olio.vector_utilities.**naive_2d_length**(*v*)

Returns the length of the vector projected onto xy plane, assuming consistent units.

resqpy.olio.vector_utilities.naive_2d_lengths

resqpy.olio.vector_utilities.**naive_2d_lengths**(*v*)

Returns the lengths of the vectors projected onto xy plane, assuming consistent units.

resqpy.olio.vector_utilities.naive_length

resqpy.olio.vector_utilities.**naive_length**(*v*)

Returns the length of the vector assuming consistent units.

resqpy.olio.vector_utilities.naive_lengths

resqpy.olio.vector_utilities.**naive_lengths**(*v*)

Returns the lengths of the vectors assuming consistent units.

resqpy.olio.vector_utilities.nan_inclinations

resqpy.olio.vector_utilities.**nan_inclinations**(*a*, *already_unit_vectors=False*)

Returns the inclination in degrees of each vector in a (angle relative to +ve z axis), allowing NaNs.

resqpy.olio.vector_utilities.nan_unit_vectors

resqpy.olio.vector_utilities.**nan_unit_vectors**(*v*)

Returns vectors with same direction as those in v but with unit length, allowing NaNs.

resqpy.olio.vector_utilities.nearest_point_projected

resqpy.olio.vector_utilities.**nearest_point_projected**(*p*, *points*, *projection*)

Returns the index into points array closest to point p; projection is 'xy', 'xz' or 'yz'.

resqpy.olio.vector_utilities.no_rotation_matrix

resqpy.olio.vector_utilities.**no_rotation_matrix**()

Returns a rotation matrix which will not move points (identity matrix).

resqpy.olio.vector_utilities.perspective_vector

resqpy.olio.vector_utilities.**perspective_vector**(*xyz_box*, *view_axis*, *vanishing_distance*, *vector*)

Returns a version of vector with a perspective applied.

resqpy.olio.vector_utilities.point_distance_sqr_to_points_projected

resqpy.olio.vector_utilities.**point_distance_sqr_to_points_projected**(*p*, *points*, *projection*)

Returns an array of projected distances squared between p and points; projection is 'xy', 'xz' or 'yz'.

resqpy.olio.vector_utilities.point_distance_to_line_2d

resqpy.olio.vector_utilities.**point_distance_to_line_2d**(*p*, *l1*, *l2*)

Ignoring any z values, returns the xy distance of point p from line passing through l1 and l2.

resqpy.olio.vector_utilities.point_distance_to_line_segment_2d

resqpy.olio.vector_utilities.**point_distance_to_line_segment_2d**(*p*, *l1*, *l2*)

Ignoring any z values, returns the xy distance of point p from line segment between l1 and l2.

resqpy.olio.vector_utilities.point_in_polygon

resqpy.olio.vector_utilities.**point_in_polygon**(*x*, *y*, *polygon*)

Calculates if a point in within a polygon in 2D.

Parameters

- **x** (*float*) – the point's x-coordinate.
- **y** (*float*) – the point's y-coordinate.
- **polygon** (*np.ndarray*) – array of the polygon's vertices in 2D.

Returns

inside (bool) – True if point is within the polygon, False otherwise.

Note: the polygon is assumed closed, the closing point should not be repeated

---

resqpy.olio.vector_utilities.point_in_triangle

resqpy.olio.vector_utilities.**point_in_triangle**(*x*, *y*, *triangle*)

Calculates if a point in within a triangle in 2D.

Parameters

- **x** (*float*) – the point's x-coordinate.
- **y** (*float*) – the point's y-coordinate.
- **triangle** (*np.ndarray*) – array of the triangles's vertices in 2D, of shape (3, 2)

Returns

inside (bool) – True if point is within the polygon, False otherwise.

resqpy.olio.vector_utilities.points_direction_vector

resqpy.olio.vector_utilities.**points_direction_vector**(*a*, *axis*)

Returns an average direction vector based on first and last non-NaN points or slices in given axis.

resqpy.olio.vector_utilities.points_in_polygon

resqpy.olio.vector_utilities.**points_in_polygon**(*points: ndarray*, *polygon: ndarray*, *points_xlen: int*, *polygon_num: int = 0*) → ndarray

Calculates which points are within a polygon in 2D.

Parameters

- **points** (*np.ndarray*) – array of shape (N, 2 or 3), of the points in 2D (xy, any z values are ignored)
- **polygon** (*np.ndarray*) – list-like array of the polygon's vertices in 2D
- **points_xlen** (*int*) – the original I extent of the now flattened points, use 1 if not applicable
- **polygon_num** (*int*) – the polygon number, default is 0, for copying to output

Returns

polygon_points (np.ndarray) –

list-like 2D array containing only the points within the polygon,
with each row being the polygon number (as input), points J index, and points I index

Note: the polygon is assumed closed, the closing point should not be repeated

resqpy.olio.vector_utilities.points_in_polygons

resqpy.olio.vector_utilities.**points_in_polygons**(*points: ndarray*, *polygons: ndarray*, *points_xlen: int*) → ndarray

Calculates which points are within which polygons in 2D.

Parameters

- **points** (*np.ndarray*) – array of the points in 2D.
- **polygons** (*np.ndarray*) – array of each polygons' vertices in 2D.
- **points_xlen** (*int*) – the number of unique x coordinates.

Returns

polygons_points (np.ndarray) –

2D array (list-like) containing only the points within each polygon,
with each row being the polygon number, points y index, and points x index.

resqpy.olio.vector_utilities.points_in_triangle

resqpy.olio.vector_utilities.**points_in_triangle**(*points: ndarray*, *triangle: ndarray*, *points_xlen: int*, *triangle_num: int = 0*) → ndarray

Calculates which points are within a triangle in 2D.

Parameters

- **points** (*np.ndarray*) – array of the points in 2D.
- **triangle** (*np.ndarray*) – array of the triangle's vertices in 2D, shape (3, 2).
- **points_xlen** (*int*) – the number of unique x coordinates.
- **triangle_num** (*int*) – the triangle number, default is 0.

Returns

triangle_points (np.ndarray) –

2D array containing only the points within the triangle,
with each row being the triangle number, points y index, and points x index.

resqpy.olio.vector_utilities.points_in_triangles

resqpy.olio.vector_utilities.**points_in_triangles**(*p*, *t*, *da*, *projection='xy'*, *edged=False*)

Returns 2D numpy bool array indicating which of points da are within which triangles.

Parameters

- **p** (*numpy float array of shape (N, 2 or 3)*) – points in use as vertices of triangles
- **t** (*numpy int array of shape (M, 3)*) – indices into first axis of p defining the triangles
- **da** (*numpy float array of shape (D, 2 or 3)*) – points to test for
- **projection** (*string, default 'xy'*) – one of 'xy', 'xz' or 'yz' being the direction of projection, ie. which elements of the second axis of p and da to use; must be 'xy' if p and da have shape (N, 2)
- **edged** (*bool, default False*) – if True, points lying exactly on the edge of a triangle are included as being in the triangle, otherwise they are excluded

Returns

numpy bool array of shape (M, D) indicating which points are within which triangles

Note: the triangles do not need to be in a consistent clockwise or anti-clockwise order

resqpy.olio.vector_utilities.points_in_triangles_aligned

resqpy.olio.vector_utilities.**points_in_triangles_aligned**(*nx: int*, *ny: int*, *dx: float*, *dy: float*, *triangles: ndarray*) → ndarray

Calculates which points are within which triangles in 2D for a regular mesh of aligned points.

Parameters

- **nx** (*int*) – number of points in x axis
- **ny** (*int*) – number of points in y axis
- **dx** (*float*) – spacing of points in x axis (first point is at half dx)
- **dy** (*float*) – spacing of points in y axis (first point is at half dy)
- **triangles** (*np.ndarray*) – float array of each triangles' vertices in 2D, shape (N, 3, 2).
- **points_xlen** (*int*) – the number of unique x coordinates.

Returns

triangles_points (np.ndarray) –

2D array (list-like) containing only the points within each triangle,
with each row being the triangle number, points y index, and points x index.

resqpy.olio.vector_utilities.points_in_triangles_aligned_optimised

resqpy.olio.vector_utilities.**points_in_triangles_aligned_optimised**(*nx: int*, *ny: int*, *dx: float*, *dy: float*, *triangles: ndarray*) → ndarray

Calculates which points are within which triangles in 2D for a regular mesh of aligned points.

Parameters

- **nx** (*int*) – number of points in x axis
- **ny** (*int*) – number of points in y axis
- **dx** (*float*) – spacing of points in x axis (first point is at half dx)
- **dy** (*float*) – spacing of points in y axis (first point is at half dy)
- **triangles** (*np.ndarray*) – float array of each triangles' vertices in 2D, shape (N, 3, 2)

Returns

triangles_points (np.ndarray) –

2D array (list-like) containing only the points within each triangle,
with each row being the triangle number, points y index, and points x index

resqpy.olio.vector_utilities.points_in_triangles_njit

resqpy.olio.vector_utilities.**points_in_triangles_njit**(*points: ndarray*, *triangles: ndarray*, *points_xlen: int*) → ndarray

Calculates which points are within which triangles in 2D.

Parameters

- **points** (*np.ndarray*) – array of the points in 2D.
- **triangles** (*np.ndarray*) – array of each triangles' vertices in 2D, shape (N, 3, 2).
- **points_xlen** (*int*) – the number of unique x coordinates.

Returns

triangles_points (np.ndarray) –

2D array (list-like) containing only the points within each triangle,
with each row being the triangle number, points y index, and points x index.

resqpy.olio.vector_utilities.project_points_onto_plane

resqpy.olio.vector_utilities.**project_points_onto_plane**(*plane_xyz*, *normal_vector*, *points*)

Modifies array of xyz points in situ to project onto a plane defined by a point and normal vector.

Note: implicit xy & z units must be the same

resqpy.olio.vector_utilities.radians_difference

resqpy.olio.vector_utilities.**radians_difference**(*a*, *b*)

Returns the angle between two vectors, in radians.

resqpy.olio.vector_utilities.radians_from_degrees

resqpy.olio.vector_utilities.**radians_from_degrees**(*deg*)

Converts angle from degrees to radians.

resqpy.olio.vector_utilities.reverse_rotation_3d_matrix

resqpy.olio.vector_utilities.**reverse_rotation_3d_matrix**(*xzy_axis_angles*)

Returns a rotation matrix which will rotate points about the y, z, then x axis by angles in degrees.

resqpy.olio.vector_utilities.rotate_array

resqpy.olio.vector_utilities.**rotate_array**(*rotation_matrix*, *a*)

Returns a copy of array a with each vector rotated by the rotation matrix.

resqpy.olio.vector_utilities.rotate_array_njit

resqpy.olio.vector_utilities.**rotate_array_njit**(*rotation_matrix*, *a*)

Returns a copy of array a with each vector rotated by the rotation matrix.

resqpy.olio.vector_utilities.rotate_vector

resqpy.olio.vector_utilities.**rotate_vector**(*rotation_matrix*, *vector*)

Returns the rotated vector.

resqpy.olio.vector_utilities.rotate_xyz_array_around_z_axis

resqpy.olio.vector_utilities.**rotate_xyz_array_around_z_axis**(*a*, *target_xy_vector*)

Returns a copy of array a suitable for presenting a cross-section using the resulting x,z values.

Parameters

- **a** (*numpy float array of shape (..., 3)*) – the xyz points to be rotated
- **target_xy_vector** (*2 (or 3) floats*) – a vector indicating which direction in source xy space will end up being mapped to the positive x axis in the returned data

Returns

numpy float array of same shape as a

Notes

if the input points of a lie in a vertical plane parallel to the target xy vector, then the resulting points will have constant y values; in general, a full rotation of the points is applied, so resulting y values will indicate distance 'into the page' for non-planar or unaligned data

resqpy.olio.vector_utilities.rotation_3d_matrix

resqpy.olio.vector_utilities.**rotation_3d_matrix**(*xzy_axis_angles*)

Returns a rotation matrix which will rotate points about the x, z, then y axis by angles in degrees.

resqpy.olio.vector_utilities.rotation_3d_matrix_njit

resqpy.olio.vector_utilities.**rotation_3d_matrix_njit**(*xzy_axis_angles*)

Returns a rotation matrix which will rotate points about the x, z, then y axis by angles in degrees.

resqpy.olio.vector_utilities.rotation_matrix_3d_axial

resqpy.olio.vector_utilities.**rotation_matrix_3d_axial**(*axis*, *angle*)

Returns a rotation matrix which will rotate points about axis (0: x, 1: y, or 2: z) by angle in degrees.

Note: this function follows the mathematical convention: a positive angle results in anti-clockwise rotation when viewed in direction of positive axis

resqpy.olio.vector_utilities.rotation_matrix_3d_vector

resqpy.olio.vector_utilities.**rotation_matrix_3d_vector**(*v*)

Returns a rotation matrix which will rotate vector v to the vertical (z) axis.

Note: the returned matrix will map a positive z axis vector onto v

resqpy.olio.vector_utilities.rotation_matrix_3d_vector_njit

resqpy.olio.vector_utilities.**rotation_matrix_3d_vector_njit**(*v*)

Returns a rotation matrix which will rotate points by inclination and azimuth of vector.

Note: the returned matrix will map a positive z axis vector onto v

resqpy.olio.vector_utilities.subtract

resqpy.olio.vector_utilities.**subtract**(*a*, *b*)

Returns vector difference a-b.

resqpy.olio.vector_utilities.tilt_3d_matrix

resqpy.olio.vector_utilities.**tilt_3d_matrix**(*azimuth*, *dip*)

Returns a 3D rotation matrix for applying a dip in a certain azimuth.

Note: if azimuth is compass bearing in degrees, and dip is in degrees, the resulting matrix can be used to rotate xyz points where x values are eastings, y values are northings and z increases downwards

resqpy.olio.vector_utilities.tilt_points

resqpy.olio.vector_utilities.**tilt_points**(*pivot_xyz*, *azimuth*, *dip*, *points*)

Modifies array of xyz points in situ to apply dip in direction of azimuth, about pivot point.

resqpy.olio.vector_utilities.triangle_box

resqpy.olio.vector_utilities.**triangle_box**(*triangle: ndarray*) → Tuple[float, float, float, float]

Finds the minimum and maximum x and y values of a single traingle.

Parameters

triangle (*np.ndarray*) – array of the traingle's vertices' x and y coordinates.

Returns

Tuple containing – - (float): minimum x value. - (float): maximum x value. - (float): minimum y value. - (float): maximum y value.

resqpy.olio.vector_utilities.triangle_normal_vector

resqpy.olio.vector_utilities.**triangle_normal_vector**(*p3*)

For a triangle in 3D space, defined by 3 vertex points, returns a unit vector normal to the plane of the triangle.

Note: resulting vector implicitly assumes that xy & z units are the same; if this is not the case, adjust vector afterwards as required

---

resqpy.olio.vector_utilities.triangle_normal_vector_numba

resqpy.olio.vector_utilities.**triangle_normal_vector_numba**(*points*)

For a triangle in 3D space, defined by 3 vertex points, returns a unit vector normal to the plane of the triangle.

Note: resulting vector implicitly assumes that xy & z units are the same; if this is not the case, adjust vector afterwards as required

---

resqpy.olio.vector_utilities.unit_corrected_length

resqpy.olio.vector_utilities.**unit_corrected_length**(*v*, *unit_conversion*)

Returns the length of the vector v after applying the unit_conversion factors.

Parameters

- **v** (*1D numpy float array*) – vector with mixed units of measure
- **unit_conversion** (*1D numpy float array*) – vector to multiply elements of v by, prior to finding length

Returns

float, being the length of v after adjustment by unit_conversion

Notes

example unit_conversion might be: [1.0, 1.0, 0.3048] to convert z from feet to metres, or [3.28084, 3.28084, 1.0] to convert x and y from metres to feet

resqpy.olio.vector_utilities.unit_vector

resqpy.olio.vector_utilities.**unit_vector**(*v*)

Returns vector with same direction as v but with unit length.

resqpy.olio.vector_utilities.unit_vector_from_azimuth

resqpy.olio.vector_utilities.**unit_vector_from_azimuth**(*azimuth*)

Returns horizontal unit vector in compass bearing given by azimuth (x = East, y = North).

resqpy.olio.vector_utilities.unit_vector_from_azimuth_and_inclination

resqpy.olio.vector_utilities.**unit_vector_from_azimuth_and_inclination**(*azimuth*, *inclination*)

Returns unit vector with compass bearing of azimuth and inclination off +z axis.

Note: assumes a left handed coordinate system with y axis north and x axis east

resqpy.olio.vector_utilities.unit_vector_njit

resqpy.olio.vector_utilities.**unit_vector_njit**(*v*)

Returns vector with same direction as v but with unit length.

resqpy.olio.vector_utilities.unit_vectors

resqpy.olio.vector_utilities.**unit_vectors**(*v*)

Returns vectors with same direction as those in v but with unit length.

resqpy.olio.vector_utilities.v_3d

resqpy.olio.vector_utilities.**v_3d**(*v*)

Returns a 3D vector for a 2D or 3D vector.

resqpy.olio.vector_utilities.vertical_intercept

resqpy.olio.vector_utilities.**vertical_intercept**(*x: float*, *x_values: ndarray*, *y_values: ndarray*) → Optional[float]

Finds the y value of a straight line between two points at a given x.

If the x value given is not within the x values of the points, returns None.

Parameters

- **x** (*float*) – x value at which to determine the y value.
- **x_values** (*np.ndarray*) – the x coordinates of point 1 and point 2.
- **y_values** (*np.ndarray*) – the y coordinates of point 1 and point 2.

Returns

y (Optional[float]) –

y value of the straight line between point 1 and point 2,
evaluated at x. If x is outside the x_values range, y is None.

resqpy.olio.vector_utilities.xy_sorted

resqpy.olio.vector_utilities.**xy_sorted**(*p*, *axis=None*)

Returns copy of points p sorted according to x or y (whichever has greater range).

Parameters

- **p** (*numpy float array of shape (..., 2) or (..., 3)*) – points to be sorted
- **axis** (*int, optional*) – 0 for x sort; 1 for y sort; None for whichever has greater range

Returns

p', axis where p' is a list-like (2D) version of p, sorted by either x or y and axis is 0 if the sort was by x, 1 if it were by y

Note: returned array is always 2D, ie. list of points

resqpy.olio.vector_utilities.xy_sorted_njit

resqpy.olio.vector_utilities.**xy_sorted_njit**(*p*, *axis=-1*)

Returns copy of points p sorted according to x or y (whichever has greater range).

resqpy.olio.vector_utilities.zero_vector

resqpy.olio.vector_utilities.**zero_vector**()

Returns a zero vector [0.0, 0.0, 0.0].

7.18.26 resqpy.olio.volume

volume.py: Functions to calculate volumes of hexahedral cells; assumes consistent length units.

Functions

pyramid_volume	Returns volume of a quadrilateral pyramid.
tetra_cell_volume	Returns volume of single hexahedral cell with corner points cp of shape (2, 2, 2, 3); assumes bilinear faces.
tetra_volumes	Returns volume array for all hexahedral cells assuming bilinear faces, using numpy operations.
tetra_volumes_slow	Returns volume array for all hexahedral cells assuming bilinear faces, using loop over cells.
tetrahedron_volume	Returns volume of a tetrahedron.

resqpy.olio.volume.pyramid_volume

resqpy.olio.volume.**pyramid_volume**(*apex*, *a*, *b*, *c*, *d*, *crs_is_right_handed=False*)

Returns volume of a quadrilateral pyramid.

Parameters

- **apex** (*triple float*) – location of the apex of the pyramid
- **a** (*each triple float*) – locations of corners of base of pyramid; clockwise viewed from apex for a left handed crs
- **b** (*each triple float*) – locations of corners of base of pyramid; clockwise viewed from apex for a left handed crs
- **c** (*each triple float*) – locations of corners of base of pyramid; clockwise viewed from apex for a left handed crs
- **d** (*each triple float*) – locations of corners of base of pyramid; clockwise viewed from apex for a left handed crs
- **crs_is_right_handed** (*boolean, default False*) – set True if xyz axes of crs are right handed

Returns

float, being the volume of the pyramid; units are implied by crs units in use by the vertices

resqpy.olio.volume.tetra_cell_volume

resqpy.olio.volume.**tetra_cell_volume**(*cp*, *centre=None*, *off_hand=False*)

Returns volume of single hexahedral cell with corner points cp of shape (2, 2, 2, 3); assumes bilinear faces.

resqpy.olio.volume.tetra_volumes

resqpy.olio.volume.**tetra_volumes**(*cp*, *centres=None*, *off_hand=False*)

Returns volume array for all hexahedral cells assuming bilinear faces, using numpy operations.

Parameters

- **cp** (*7D numpy array of floats*) – cell corner point data in Pagoda 7D format [nk, nj, ni, kp, jp, ip, xyz]
- **centres** (*optional, 4D numpy array of floats*) – cell centre points [nk, nj, ni, xyz]; calculated if None

- **off_hand** (*boolean, default False*) – if True, the handedness of IJK space is the opposite of that for xyz space; if this argument is not set correctly, negative volumes will be returned

Returns
numpy 3D array of floats being the cell volumes [nk, nj, ni]

Note: length units are assumed to be consistent in x, y & z; and untis of returned volumes are implicitly those length units cubed

---

resqpy.olio.volume.tetra_volumes_slow

resqpy.olio.volume.**tetra_volumes_slow**(*cp*, *centres=None*, *off_hand=False*)

Returns volume array for all hexahedral cells assuming bilinear faces, using loop over cells.

resqpy.olio.volume.tetrahedron_volume

resqpy.olio.volume.**tetrahedron_volume**(*a*, *b*, *c*, *d*)

Returns volume of a tetrahedron.

Parameters

- **a** (*each triple float*) – locations of corners of tetrahedron
- **b** (*each triple float*) – locations of corners of tetrahedron
- **c** (*each triple float*) – locations of corners of tetrahedron
- **d** (*each triple float*) – locations of corners of tetrahedron
- **crs_is_right_handed** (*boolean, default False*) – set True if xyz axes of crs are right handed

Returns
float, being the volume of the tetrahedron; units are implied by crs units in use by the vertices

7.18.27 resqpy.olio.wellspec_keywords

Module for loading WELLSPEC files.

The module includes a dictionary of nexus WELLSPEC column keywords, functionality to read WELLSPEC files and transform the well data into Pandas DataFrames.

Functions

add_unknown_keyword	Adds the keyword to the dictionary with attributes flagged as unknown.
check_value	Returns True if the value is acceptable for the keyword.
complaints	Returns the number of complaints (warnings) logged for the keyword.
default_value	Returns the default value for the keyword.
get_all_well_data	Creates a dataframe of all the well data for a given well name in the wellspec file.
get_well_data	Creates a dataframe of the well data for a given well name and at a specific time in the wellspec file.
get_well_pointers	Gets the file locations of each well in the wellspec file for optimised processing of the data.
increment_complaints	Increments the count of complaints (warnings) associated with the keyword.
known_keyword	Returns True if the keyword exists in the wellspec dictionary.
length_unit_conversion_applicable	Returns True if the keyword has a quantity class of length.
load_wellspecs	Reads the Nexus wellspec file returning a dictionary of well name to pandas dataframe.
required_out_list	Returns a list of keywords that are required.

resqpy.olio.wellspec_keywords.add_unknown_keyword

resqpy.olio.wellspec_keywords.**add_unknown_keyword**(*keyword*)

Adds the keyword to the dictionary with attributes flagged as unknown.

resqpy.olio.wellspec_keywords.check_value

resqpy.olio.wellspec_keywords.**check_value**(*keyword*, *value*)

Returns True if the value is acceptable for the keyword.

resqpy.olio.wellspec_keywords.complaints

resqpy.olio.wellspec_keywords.**complaints**(*keyword*)

Returns the number of complaints (warnings) logged for the keyword.

resqpy.olio.wellspec_keywords.default_value

resqpy.olio.wellspec_keywords.**default_value**(*keyword*)

Returns the default value for the keyword.

resqpy.olio.wellspec_keywords.get_all_well_data

resqpy.olio.wellspec_keywords.**get_all_well_data**(*file: TextIO*, *well_name: str*, *pointers: List[Tuple[int, Union[None, str]]]*, *column_list: List[str] = []*, *selecting: bool = False*, *keep_duplicate_cells: bool = False*, *keep_null_columns: bool = True*, *last_data_only: bool = True*) → Optional[DataFrame]

Creates a dataframe of all the well data for a given well name in the wellspec file.

This differs from the get_well_data function in that here multiple datasets for a well are combined into a single dataframe if they exist.

Parameters

- **file** (*TextIO*) – the opened wellspec file object.
- **well_name** (*str*) – name of the well.
- **pointers** (*List[Tuple[int, None/str]]*) – a list of the file object's start position of the well data represented as number of bytes from the beginning of the file and the well's date. If no date existed before the well in the file, the date will be None.
- **column_list** (*List[str]*) – if present, each dataframe returned contains these columns, in this order. If None, the resulting dictionary contains only well names as keys (each mapping to None rather than a dataframe). If an empty list (default), each dataframe contains the columns listed in the corresponding wellspec header, in the order found in the file.
- **selecting** (*bool*) – True if the column_list contains at least one column name, False otherwise (default).
- **keep_duplicate_cells** (*bool*) – if True (default), duplicate cells are kept, otherwise only the last entry is kept.
- **keep_null_columns** (*bool*) – if True (default), columns that contain all NA values are kept, otherwise they are removed.
- **last_data_only** (*bool*) – If True, only the last entry of well data in the file are used in the dataframe, otherwise all of the well data are used at different times.

Returns

Pandas dataframe of all well data for a specific well name or None if all the data are NA.

resqpy.olio.wellspec_keywords.get_well_data

resqpy.olio.wellspec_keywords.**get_well_data**(*file: TextIO*, *well_name: str*, *pointer: int*, *column_list: List[str] = []*, *selecting: bool = False*, *keep_duplicate_cells: bool = True*, *keep_null_columns: bool = True*, *date: Optional[str] = None*) → Optional[DataFrame]

Creates a dataframe of the well data for a given well name and at a specific time in the wellspec file.

The pointer argument is used to go to the file location where the well dataset is located.

Parameters

- **file** (*TextIO*) – the opened wellspec file object.
- **well_name** (*str*) – name of the well.

- **pointer** (*int*) – the file object's start position of the well data represented as number of bytes from the beginning of the file.
- **column_list** (*List[str]*) – if present, each dataframe returned contains these columns, in this order. If None, the resulting dictionary contains only well names as keys (each mapping to None rather than a dataframe). If an empty list (default), each dataframe contains the columns listed in the corresponding wellspec header, in the order found in the file.
- **selecting** (*bool*) – True if the column_list contains at least one column name, False otherwise (default).
- **keep_duplicate_cells** (*bool*) – if True (default), duplicate cells are kept, otherwise only the last entry is kept.
- **keep_null_columns** (*bool*) – if True (default), columns that contain all NA values are kept, otherwise they are removed.
- **date** (*str, optional*) – the well date which is provided by the get_well_pointers function along with the well pointers.

Returns

Pandas dataframe of the well data or None if all the data are NA.

resqpy.olio.wellspec_keywords.get_well_pointers

resqpy.olio.wellspec_keywords.**get_well_pointers**(*wellspec_file: str*, *usa_date_format: bool = False*, *no_date_replacement: Optional[date] = None*) → Dict[str, List[Tuple[int, Union[None, str]]]]

Gets the file locations of each well in the wellspec file for optimised processing of the data.

Parameters

- **wellspec_file** (*str*) – file path of ascii input file containing wellspec keywords.
- **usa_date_format** (*bool*) – if True, the date taken from the wellspec file is in the format MM/DD/YYYY, otherwise it is in the format DD/MM/YYYY.
- **no_date_replacement** (*datetime.date, optional*) – if there is no date given for a well, this date is used.

Returns

well_pointers (Dict[str, List[Tuple[int, None/str]]]) –

mapping each well name found in
the wellspec file to a list of their file locations and dates as tuples. If there is no date before the well data in the file, the date is None. If there is a FileNotFoundError then None is returned.

resqpy.olio.wellspec_keywords.increment_complaints

resqpy.olio.wellspec_keywords.**increment_complaints**(*keyword*)

Increments the count of complaints (warnings) associated with the keyword.

resqpy.olio.wellspec_keywords.known_keyword

resqpy.olio.wellspec_keywords.**known_keyword**(*keyword*)

Returns True if the keyword exists in the wellspec dictionary.

resqpy.olio.wellspec_keywords.length_unit_conversion_applicable

resqpy.olio.wellspec_keywords.**length_unit_conversion_applicable**(*keyword*)

Returns True if the keyword has a quantity class of length.

resqpy.olio.wellspec_keywords.load_wellspecs

resqpy.olio.wellspec_keywords.**load_wellspecs**(*wellspec_file: str*, *well: Optional[str] = None*, *column_list: Optional[List[str]] = []*, *keep_duplicate_cells: bool = False*, *keep_null_columns: bool = True*, *last_data_only: bool = True*, *usa_date_format: bool = False*, *return_dates_list: bool = False*)

Reads the Nexus wellspec file returning a dictionary of well name to pandas dataframe.

Parameters

- **wellspec_file** (*str*) – file path of ascii input file containing wellspec keywords.
- **well** (*str, optional*) – if present, only the data for the named well are loaded. If None, data for all wells are loaded.
- **column_list** (*List[str]/None*) – if present, each dataframe returned contains these columns, in this order. If None, the resulting dictionary contains only well names as keys (each mapping to None rather than a dataframe). If an empty list (default), each dataframe contains the columns listed in the corresponding wellspec header, in the order found in the file.
- **keep_duplicate_cells** (*bool*) – if True (default), duplicate cells are kept, otherwise only the last entry is kept.
- **keep_null_columns** (*bool*) – if True (default), columns that contain all NA values are kept, otherwise they are removed.
- **last_data_only** (*bool*) – If True, only the last entry of well data in the file are used in the dataframe, otherwise all of the well data are used at different times.
- **usa_date_format** (*bool*) – If True, wellspec file is expected to contain date formats in MM/DD/YYYY. if False, DD/MM/YYYY.
- **return_dates_list** (*bool, default False*) – if True, a sorted list of unique dates present in the wellspec file is also returned, with dates in iso format

Returns

well_dict (Dict[str, Union[pd.DataFrame, None]]) –

mapping each well name found in the
wellspec file to a dataframe containing the wellspec data

or (well_dict, dates_list): where dates list is a sorted list of all dates present in the
wellspec file (including those not relevant to a specific well), in iso format

Note: if return_dates_list is True, the returned list always contains all dates from the wellspec file that applied to any entry, regardless of the well and last_data_only arguments; the dates list will not include a null entry, even if there are wellspec data before the first timestamp

resqpy.olio.wellspec_keywords.required_out_list

resqpy.olio.wellspec_keywords.**required_out_list**()

Returns a list of keywords that are required.

7.18.28 resqpy.olio.write_data

Array writing functions.

Functions

write_array_to_ascii_file	Writes a 3D array of data to an ascii file.
write_pure_binary_data	Writes a numpy array to a file in 'pure binary' format.

resqpy.olio.write_data.write_array_to_ascii_file

resqpy.olio.write_data.**write_array_to_ascii_file**(*file_name*, *extent_kji*, *a*, *headers=True*, *keyword=None*, *columns=20*, *data_type='real'*, *decimals=3*, *target_simulator='nexus'*, *blank_line_after_i_block=True*, *blank_line_after_j_block=False*, *space_separated=False*, *append=False*, *use_binary=False*, *binary_only=False*, *nan_substitute_value=None*)

Writes a 3D array of data to an ascii file.

resqpy.olio.write_data.write_pure_binary_data

resqpy.olio.write_data.**write_pure_binary_data**(*binary_file_name*, *numpy_array*)

Writes a numpy array to a file in 'pure binary' format.

7.18.29 resqpy.olio.write_hdf5

write_hdf5.py: Class to write a resqml hdf5 file and functions for copying hdf5 data.

Classes

H5Register	Class for registering arrays and then writing to an hdf5 file.

resqpy.olio.write_hdf5.H5Register

class resqpy.olio.write_hdf5.**H5Register**(*model*, *default_chunks=None*, *default_compression=None*)

Bases: object

Class for registering arrays and then writing to an hdf5 file.

Methods:

__init__(model[, default_chunks, ...])	Create a new, empty register of arrays to be written to an hdf5 file.
register_dataset(object_uuid, group_tail, a)	Register an array to be included as a dataset in the hdf5 file.
write_fp(fp[, use_int32])	Write or append to an hdf5 file, writing the pre-registered datasets (arrays).
write([file, mode, release_after, use_int32])	Create or append to an hdf5 file, writing the pre-registered datasets (arrays).

__init__(*model*, *default_chunks=None*, *default_compression=None*)

Create a new, empty register of arrays to be written to an hdf5 file.

register_dataset(*object_uuid*, *group_tail*, *a*, *dtype=None*, *hdf5_internal_path=None*, *copy=False*, *chunks=None*, *compression=None*)

Register an array to be included as a dataset in the hdf5 file.

Parameters

- **object_uuid** (*uuid.UUID*) – the uuid of the object (part) that this array is for
- **group_tail** (*string*) – the remainder of the hdf5 internal path (following RESQML and uuid elements)
- **a** (*numpy array*) – the dataset (array) to be registered for writing
- **dtype** (*type or string*) – the required type of the individual elements within the dataset; special value of 'pack' may be used to cause a bool array to be packed before writing
- **hdf5_internal_path** (*string, optional*) – if present, a full hdf5 internal path to use instead of the default generated from the uuid
- **copy** (*boolean, default False*) – if True, a copy of the array will be made at the time of registering, otherwise changes made to the array before the write() method is called are likely to be in the data that is written
- **chunks** (*str or tuple of ints, optional*) – if not None, chunked hdf5 storage will be used; if str, options are 'auto', 'all', 'slice'
- **compression** (*str, optional*) – if not None, either 'gzip' or 'lzf'

Returns
None

Notes

several arrays might belong to the same object; if a dtype is given and necessitates a conversion of the array data, the behaviour will be as if the copy argument is True regardless of its setting; the use of 'pack' as dtype will result in hdf5 data that will not generally be readable by non-resqpy applications; when reading packed data, the required shape must be specified; packing only takes place over the last axis; do not use packing if the array needs to be read or updated in slices, or read a single value at a time with index values; if chunks is set to a tuple, it must have the same ndim as a and the shape of a must be a mulitple of the entries in the chunks tuple, in each dimension; if chunks is 'all', the shape of a will be used as the tuple; if 'auto' then hdf5 auto chunking will be used; if 'slice' and a has more than one dimension, then the chunks tuple will be the shape of a with the first entry replaced with 1

`write_fp`(*fp*, *use_int32=None*)

Write or append to an hdf5 file, writing the pre-registered datasets (arrays).

Parameters
fp – an already open h5py._hl.files.File object

Returns
None

Note: the file handle fp must have been opened with mode 'w' or 'a'

`write`(*file=None*, *mode='w'*, *release_after=True*, *use_int32=None*)

Create or append to an hdf5 file, writing the pre-registered datasets (arrays).

Parameters

- **file** – either a string being the file path, or an already open h5py._hl.files.File object; if None (recommended), the file is opened through the model object's hdf5 management functions
- **mode** (*string, default 'w'*) – the mode to open the file in; only relevant if file is a path; must be 'w' or 'a' for (over)write or append
- **release_after** (*bool, default True*) – if True, h5_release() is called after the write
- **use_int32** (*bool, optional*) – if True, int64 arrays will be written as int32; if None, global default will be used (currently True); if False, int64 arrays will be written as such

Returns
None

Functions

change_uuid	Changes hdf5 internal path (group name) for part, switching from old to new uuid.
copy_h5	Create a copy of an hdf5, optionally including or excluding arrays with specified uuids.
copy_h5_path_list	Create a copy of some hdf5 datasets (or groups), identified as a list of hdf5 internal paths.
set_global_default_chunks_and_compression	Set global default values for hdf5 chunks and compression.

resqpy.olio.write_hdf5.change_uuid

resqpy.olio.write_hdf5.**change_uuid**(*file*, *old_uuid*, *new_uuid*)

Changes hdf5 internal path (group name) for part, switching from old to new uuid.

Notes

this is low level functionality not usually called directly; the function assumes that hdf5 internal path names conform to the format that resqpy uses when writing data, namely /RESQML/uuid/tail…

resqpy.olio.write_hdf5.copy_h5

resqpy.olio.write_hdf5.**copy_h5**(*file_in*, *file_out*, *uuid_inclusion_list=None*, *uuid_exclusion_list=None*, *mode='w'*)

Create a copy of an hdf5, optionally including or excluding arrays with specified uuids.

Parameters

- **file_in** (*string*) – path of existing hdf5 file to be duplicated
- **file_out** (*string*) – path of output hdf5 file to be created or appended to (see mode)
- **uuid_inclusion_list** (*list of uuid.UUID, optional*) – if present, the uuids to be included in the output file
- **uuid_exclusion_list** (*list of uuid.UUID, optional*) – if present, the uuids to be excluded from the output file
- **mode** (*string, default 'w'*) – mode to open output file with; must be ‘w’ or ‘a’ for (over)write or append respectively

Returns

number of hdf5 groups (uuid’s) copied

Notes

at most one of uuid_inclusion_list and uuid_exclusion_list should be passed; if neither are passed, all the datasets (arrays) in the input file are copied to the output file

resqpy.olio.write_hdf5.copy_h5_path_list

resqpy.olio.write_hdf5.**copy_h5_path_list**(*file_in*, *file_out*, *hdf5_path_list*, *mode='w'*, *chunks=None*, *compression=None*)

Create a copy of some hdf5 datasets (or groups), identified as a list of hdf5 internal paths.

Parameters

- **file_in** (*string*) – path of existing hdf5 file to be copied from
- **file_out** (*string*) – path of output hdf5 file to be created or appended to (see mode)
- **hdf5_path_list** (*list of string*) – the hdf5 internal paths of the datasets (or groups) to be copied
- **mode** (*string, default 'w'*) – mode to open output file with; must be 'w' or 'a' for (over)write or append respectively
- **chunks** (*string, optional*) – if present, one of 'auto', 'all', 'slice'; if None, global default will be used; any of the valid strings will actually be treated as 'auto'
- **compression** (*string, optional*) – if present, either 'gzip' or 'lzf'; if None, global default will be used

Returns

number of hdf5 datasets (or groups) copied

resqpy.olio.write_hdf5.set_global_default_chunks_and_compression

resqpy.olio.write_hdf5.**set_global_default_chunks_and_compression**(*chunks*, *compression*)

Set global default values for hdf5 chunks and compression.

Parameters

- **chunks** (*str, or None*) – if str, one of 'auto', 'all', or 'slice'
- **compression** (*str, or None*) – if str, either 'gzip' or 'lzf'

7.18.30 resqpy.olio.xml_et

xml_et.py: Resqml xml element tree utilities module.

Functions

bool_from_text	Returns boolean value for string 'true' or 'false'; anything else results in None.
citation_title_for_node	Looks for a citation node as a child of node and returns the title text.
colon_prefixed	Returns a version of an xml tag with {url} prefix replaced with nsi: equivalent; also returns the nsi prefix.
content_type	Returns the actual type, as embedded in an xml Content-Type attribute; application and version are disregarded.
count_tag	Returns the number of children in xml node with a (prefix-stripped) tag matching given tag name.
create_metadata_xml	Writes the xml for the given metadata dictionary.
creation_date_for_node	Looks for a citation node as a child of node and returns the creation (date-time) text.
cut_extra_metadata	Removes all the extra metadata children under root node.
cut_nodes_of_types	Deletes any nodes of a type matching one in the given list.
cut_obj_references	Deletes any object reference nodes to uuids in given list.
find_in_ordered_data	Returns the index in the ordered list-like array of value; or None if not present.
find_nested_tags	Follows a list of tags in a nested xml hierarchy, returning the node at the deepest level.
find_nested_tags_bool	Return stripped text of node at deepest level of xml hierarchy as a bool.
find_nested_tags_cast	Return value of nested tags as desired dtype.
find_nested_tags_float	Return stripped text of node at deepest level of xml hierarchy as a float.
find_nested_tags_int	Return stripped text of node at deepest level of xml hierarchy as an int.
find_nested_tags_text	Return stripped text of node at deepest level of xml hierarchy.
find_tag	Finds the first child in xml node with a (prefix-stripped) tag matching given tag name.
find_tag_bool	Finds the first child in xml node with a tag matching given tag name; returns stripped text field as bool.
find_tag_float	Finds the first child in xml node with a tag matching given tag name; returns stripped text field as float.
find_tag_int	Finds the first child in xml node with a tag matching given tag name; returns stripped text field as int.
find_tag_text	Finds the first child in xml node with a tag matching given tag name; returns stripped text field.
ijk_handedness	Returns ijk true handedness as 'left', 'right' or 'unknown'.
is_node	Returns True if type of object is element tree node; False otherwise.
length_units_from_node	Returns standard length units string based on node text, or 'unknown'.
list_obj_references	Returns list of nodes of type DataObjectReference.
list_of_descendant_tag	Returns a list of descendants in xml node tree with a (prefix-stripped) tag matching given tag name.

continues on next page

Table 7 – continued from previous page

list_of_tag	Returns a list of children in xml node with a (prefix-stripped) tag matching given tag name.
load_metadata_from_xml	Loads the ExtraMetaData stored in a RESQML part as a dictionary.
match	Returns True if the xml_name stripped of prefix matches name.
node_bool	Returns stripped node text as bool, or None.
node_float	Returns stripped node text as float, or None.
node_int	Returns stripped node text as int, or None.
node_text	Returns stripped node text or 'unknown' if node is None or text is blank or newline.
node_type	Returns the type as held in attributes of xml node; defining authority is stripped out.
part_name_for_object	Returns the standard part name comprised of the object type, uuid and .xml extension.
part_name_for_part_root	Returns the part name given the root node for the part's xml.
patch_uuid_in_part_root	Returns modified part name with uuid swapped to uuid argument; root attrib is also changed.
print_xml_tree	Print an xml tree in an indented semi-readable format; return accumulated number of lines.
rels_part_name_for_part	Returns the paired relationships part name for the given part name.
simplified_data_type	Returns a simplified string version of the elemental data type (typically for a numpy or hdf5 array).
strip_path	Returns the filename part of full_path with any directory path removed.
stripped_of_prefix	Returns a simplified version of an xml tag or other str with any {xsd defining prefix} stripped off.
time_units_from_node	Returns standard time units string based on node text, or 'unknown'.
uuid_for_part_root	Returns uuid as stored in xml attribs for root.
uuid_in_part_name	Returns uuid as embedded in part name.
write_xml	Write an xml tree to file in an indented format; gSOAP/FESAPI compatible; return number of nodes written.
write_xml_node	Recursively write an xml node to an open file; return number of nodes written.
xyz_handedness	Return xyz true handedness as 'left', 'right' or 'unknown'.

resqpy.olio.xml_et.bool_from_text

resqpy.olio.xml_et.**bool_from_text**(*text*)

Returns boolean value for string 'true' or 'false'; anything else results in None.

resqpy.olio.xml_et.citation_title_for_node

resqpy.olio.xml_et.**citation_title_for_node**(*node*)

Looks for a citation node as a child of node and returns the title text.

resqpy.olio.xml_et.colon_prefixed

resqpy.olio.xml_et.**colon_prefixed**(*curly_prefixed*)

Returns a version of an xml tag with {url} prefix replaced with nsi: equivalent; also returns the nsi prefix.

resqpy.olio.xml_et.content_type

resqpy.olio.xml_et.**content_type**(*content_type_str*)

Returns the actual type, as embedded in an xml ContentType attribute; application and version are disregarded.

resqpy.olio.xml_et.count_tag

resqpy.olio.xml_et.**count_tag**(*root*, *tag_name*)

Returns the number of children in xml node with a (prefix-stripped) tag matching given tag name.

resqpy.olio.xml_et.create_metadata_xml

resqpy.olio.xml_et.**create_metadata_xml**(*node*, *extra_metadata*)

Writes the xml for the given metadata dictionary.

resqpy.olio.xml_et.creation_date_for_node

resqpy.olio.xml_et.**creation_date_for_node**(*node*)

Looks for a citation node as a child of node and returns the creation (date-time) text.

resqpy.olio.xml_et.cut_extra_metadata

resqpy.olio.xml_et.**cut_extra_metadata**(*root*)

Removes all the extra metadata children under root node.

resqpy.olio.xml_et.cut_nodes_of_types

resqpy.olio.xml_et.**cut_nodes_of_types**(*root*, *types_to_be_cut*)

Deletes any nodes of a type matching one in the given list.

resqpy.olio.xml_et.cut_obj_references

resqpy.olio.xml_et.**cut_obj_references**(*root*, *uuids_to_be_cut*)

Deletes any object reference nodes to uuids in given list.

resqpy.olio.xml_et.find_in_ordered_data

resqpy.olio.xml_et.**find_in_ordered_data**(*value*, *array_1d*)

Returns the index in the ordered list-like array of value; or None if not present.

resqpy.olio.xml_et.find_nested_tags

resqpy.olio.xml_et.**find_nested_tags**(*root*, *tag_list*)

Follows a list of tags in a nested xml hierarchy, returning the node at the deepest level.

resqpy.olio.xml_et.find_nested_tags_bool

resqpy.olio.xml_et.**find_nested_tags_bool**(*root*, *tag_list*)

Return stripped text of node at deepest level of xml hierarchy as a bool.

Parameters

tag_list (*list of str*) – list of tags in a nested xml hierarchy

resqpy.olio.xml_et.find_nested_tags_cast

resqpy.olio.xml_et.**find_nested_tags_cast**(*root*, *tag_list*, *dtype=None*)

Return value of nested tags as desired dtype.

Follows a list of tags in a nested xml hierarchy, returning the stripped text of the node at the deepest level.

resqpy.olio.xml_et.find_nested_tags_float

resqpy.olio.xml_et.**find_nested_tags_float**(*root*, *tag_list*)

Return stripped text of node at deepest level of xml hierarchy as a float.

Parameters

tag_list (*list of str*) – list of tags in a nested xml hierarchy

resqpy.olio.xml_et.find_nested_tags_int

resqpy.olio.xml_et.**find_nested_tags_int**(*root*, *tag_list*)

Return stripped text of node at deepest level of xml hierarchy as an int.

Parameters

tag_list (*list of str*) – list of tags in a nested xml hierarchy

resqpy.olio.xml_et.find_nested_tags_text

resqpy.olio.xml_et.**find_nested_tags_text**(*root*, *tag_list*)

Return stripped text of node at deepest level of xml hierarchy.

Parameters

tag_list (*list of str*) – list of tags in a nested xml hierarchy

resqpy.olio.xml_et.find_tag

resqpy.olio.xml_et.**find_tag**(*root*, *tag_name*, *must_exist=False*)

Finds the first child in xml node with a (prefix-stripped) tag matching given tag name.

resqpy.olio.xml_et.find_tag_bool

resqpy.olio.xml_et.**find_tag_bool**(*root*, *tag_name*, *must_exist=False*)

Finds the first child in xml node with a tag matching given tag name; returns stripped text field as bool.

resqpy.olio.xml_et.find_tag_float

resqpy.olio.xml_et.**find_tag_float**(*root*, *tag_name*, *must_exist=False*)

Finds the first child in xml node with a tag matching given tag name; returns stripped text field as float.

resqpy.olio.xml_et.find_tag_int

resqpy.olio.xml_et.**find_tag_int**(*root*, *tag_name*, *must_exist=False*)

Finds the first child in xml node with a tag matching given tag name; returns stripped text field as int.

resqpy.olio.xml_et.find_tag_text

resqpy.olio.xml_et.**find_tag_text**(*root*, *tag_name*, *must_exist=False*)

Finds the first child in xml node with a tag matching given tag name; returns stripped text field.

resqpy.olio.xml_et.ijk_handedness

resqpy.olio.xml_et.**ijk_handedness**(*geom_node*)

Returns ijk true handedness as 'left', 'right' or 'unknown'.

Parameters

geom_node – GridIsRightHanded node in grid geometry node.

resqpy.olio.xml_et.is_node

resqpy.olio.xml_et.**is_node**(*obj*)

Returns True if type of object is element tree node; False otherwise.

resqpy.olio.xml_et.length_units_from_node

resqpy.olio.xml_et.**length_units_from_node**(*node*)

Returns standard length units string based on node text, or 'unknown'.

resqpy.olio.xml_et.list_obj_references

resqpy.olio.xml_et.**list_obj_references**(*root*, *skip_hdf5=True*)

Returns list of nodes of type DataObjectReference.

resqpy.olio.xml_et.list_of_descendant_tag

resqpy.olio.xml_et.**list_of_descendant_tag**(*root*, *tag_name*)

Returns a list of descendants in xml node tree with a (prefix-stripped) tag matching given tag name.

resqpy.olio.xml_et.list_of_tag

resqpy.olio.xml_et.**list_of_tag**(*root*, *tag_name*)

Returns a list of children in xml node with a (prefix-stripped) tag matching given tag name.

resqpy.olio.xml_et.load_metadata_from_xml

resqpy.olio.xml_et.**load_metadata_from_xml**(*node*)

Loads the ExtraMetaData stored in a RESQML part as a dictionary.

resqpy.olio.xml_et.match

resqpy.olio.xml_et.**match**(*xml_name*, *name*)

Returns True if the xml_name stripped of prefix matches name.

resqpy.olio.xml_et.node_bool

resqpy.olio.xml_et.**node_bool**(*node*)

Returns stripped node text as bool, or None.

resqpy.olio.xml_et.node_float

resqpy.olio.xml_et.**node_float**(*node*)

Returns stripped node text as float, or None.

resqpy.olio.xml_et.node_int

resqpy.olio.xml_et.**node_int**(*node*)

Returns stripped node text as int, or None.

resqpy.olio.xml_et.node_text

resqpy.olio.xml_et.**node_text**(*node*, *unknown_if_none=False*)

Returns stripped node text or 'unknown' if node is None or text is blank or newline.

resqpy.olio.xml_et.node_type

resqpy.olio.xml_et.**node_type**(*node*, *is_rels=False*, *strip_obj=False*)

Returns the type as held in attributes of xml node; defining authority is stripped out.

resqpy.olio.xml_et.part_name_for_object

resqpy.olio.xml_et.**part_name_for_object**(*obj_type*, *uuid*, *prefixed=False*, *epc_subdir=None*)

Returns the standard part name comprised of the object type, uuid and .xml extension.

resqpy.olio.xml_et.part_name_for_part_root

resqpy.olio.xml_et.**part_name_for_part_root**(*root*, *is_rels=False*, *epc_subdir=None*)

Returns the part name given the root node for the part's xml.

resqpy.olio.xml_et.patch_uuid_in_part_root

resqpy.olio.xml_et.**patch_uuid_in_part_root**(*root*, *uuid*)

Returns modified part name with uuid swapped to uuid argument; root attrib is also changed.

resqpy.olio.xml_et.print_xml_tree

resqpy.olio.xml_et.**print_xml_tree**(*root*, *level=0*, *max_level=None*, *strip_tag_refs=True*, *to_log=False*, *log_level=None*, *max_lines=0*, *line_count=0*)

Print an xml tree in an indented semi-readable format; return accumulated number of lines.

resqpy.olio.xml_et.rels_part_name_for_part

resqpy.olio.xml_et.**rels_part_name_for_part**(*part_name*)

Returns the paired relationships part name for the given part name.

resqpy.olio.xml_et.simplified_data_type

resqpy.olio.xml_et.**simplified_data_type**(*array_dtype*)

Returns a simplified string version of the elemental data type (typically for a numpy or hdf5 array).

resqpy.olio.xml_et.strip_path

resqpy.olio.xml_et.**strip_path**(*full_path*)

Returns the filename part of full_path with any directory path removed.

resqpy.olio.xml_et.stripped_of_prefix

resqpy.olio.xml_et.**stripped_of_prefix**(*s*)

Returns a simplified version of an xml tag or other str with any {xsd defining prefix} stripped off.

resqpy.olio.xml_et.time_units_from_node

resqpy.olio.xml_et.**time_units_from_node**(*node*)

Returns standard time units string based on node text, or 'unknown'.

resqpy.olio.xml_et.uuid_for_part_root

resqpy.olio.xml_et.**uuid_for_part_root**(*root*)

Returns uuid as stored in xml attribs for root.

resqpy.olio.xml_et.uuid_in_part_name

resqpy.olio.xml_et.**uuid_in_part_name**(*part_name*, *return_uuid_str=False*)

Returns uuid as embedded in part name.

resqpy.olio.xml_et.write_xml

resqpy.olio.xml_et.**write_xml**(*xml_fp*, *tree*, *standalone=None*)

Write an xml tree to file in an indented format; gSOAP/FESAPI compatible; return number of nodes written.

resqpy.olio.xml_et.write_xml_node

resqpy.olio.xml_et.**write_xml_node**(*xml_fp*, *root*, *level=0*, *namespace_keys=[]*)

Recursively write an xml node to an open file; return number of nodes written.

resqpy.olio.xml_et.xyz_handedness

resqpy.olio.xml_et.**xyz_handedness**(*xy_axes: str*, *z_inc_down: bool*)

Return xyz true handedness as 'left', 'right' or 'unknown'.

7.18.31 resqpy.olio.xml_namespaces

xml_namespaces.py: Module defining constant resqml xml namespaces.

Functions

colon_namespace	Returns the short form namespace for the url, complete with colon suffix.

resqpy.olio.xml_namespaces.colon_namespace

resqpy.olio.xml_namespaces.**colon_namespace**(*url*)

Returns the short form namespace for the url, complete with colon suffix.

7.18.32 resqpy.olio.zmap_reader

Functions for reading zmap and roxar format files.

Functions

read_mesh	Reads a mesh (lattice) from a zmap or roxar format file.
read_rms_text_mesh	Read RMS text format surface mesh; returns triple (x, y, z) 2D arrays.
read_roxar_header	Reads header lines from a roxar format file.
read_roxar_mesh	Read RMS text format surface mesh; returns triple (x, y, z) 2D arrays.
read_zmap_header	Reads header lines from a zmap format file.
read_zmapplusgrid	Read zmapplus grid (surface mesh); returns triple (x, y, z) 2D arrays.

resqpy.olio.zmap_reader.read_mesh

resqpy.olio.zmap_reader.**read_mesh**(*inputfile*, *dtype=<class 'numpy.float64'>*, *format=None*)

Reads a mesh (lattice) from a zmap or roxar format file.

Returns

x, y, f – each a numpy float array of shape (no_rows, no_cols)

resqpy.olio.zmap_reader.read_rms_text_mesh

resqpy.olio.zmap_reader.**read_rms_text_mesh**(*inputfile*, *dtype=<class 'numpy.float64'>*)

Read RMS text format surface mesh; returns triple (x, y, z) 2D arrays.

Note: the RMS text format was previously known as the Roxar format

resqpy.olio.zmap_reader.read_roxar_header

resqpy.olio.zmap_reader.**read_roxar_header**(*inputfile*)

Reads header lines from a roxar format file.

Returns

header_lines_count, no_rows, no_cols, minx, maxx, miny, maxy, null_value

resqpy.olio.zmap_reader.read_roxar_mesh

resqpy.olio.zmap_reader.**read_roxar_mesh**(*inputfile*, *dtype=<class 'numpy.float64'>*)

Read RMS text format surface mesh; returns triple (x, y, z) 2D arrays.

Note: the RMS text format was previously known as the Roxar format

resqpy.olio.zmap_reader.read_zmap_header

resqpy.olio.zmap_reader.**read_zmap_header**(*inputfile*)

Reads header lines from a zmap format file.

Returns

header_lines_count, no_rows, no_cols, minx, maxx, miny, maxy, null_value

resqpy.olio.zmap_reader.read_zmapplusgrid

resqpy.olio.zmap_reader.**read_zmapplusgrid**(*inputfile*, *dtype=<class 'numpy.float64'>*)

Read zmapplus grid (surface mesh); returns triple (x, y, z) 2D arrays.

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z