The valjean documentation

Release 0.10.0

Ève Le Ménédeu, Davide Mancusi

Jan 27, 2023

Contents:

Fig. 1: Jean Valjean avec ses chandeliers, illustré par Émile Bayard (1862).

1

What is valjean?

valjean is a framework for building test suites for scientific computing. It can help with the automation of most of the steps of a typical test suite, including checkout and compilation of the source code, execution of calculations, post-processing of calculations results, hypothesis testing and generation of test reports.

valjean is developed by the French Alternative Energies and Atomic Energy Commission (CEA) as a tool for the verification and validation (V&V) suite of the Tripoli-4® Monte-Carlo particle-transport code. It includes a parser for Tripoli-4 calculation results, but its aim is to be more broadly useful to the whole community of scientific computing.

valjean is available on GitHub.

2 Getting started

2.1 Installing `valjean`

2.1.1 Requirements

valjean requires Python >= 3.6. The package dependencies are handled by the build system.

2.1.2 Getting the code

valjean is available on GitHub. It can be downloaded using:

```
$ git clone https://github.com/valjean-framework/valjean.git
```

2.1.3 Quick start

For those in a rush:

```
$ python3 -m venv ~/venv-valjean
$ source ~/venv-valjean/bin/activate
(venv-valjean) $ pip install --upgrade pip
(venv-valjean) $ pip install /path/to/valjean
```

Note: `/path/to/valjean` is the location of the package sources.

2.1.4 Using virtual environments

The *valjean* package can be installed like a normal Python package, using the pip package manager.

The recommended way to install *valjean* is to use a *virtual environment*. At the time of writing, the preferred solution to create virtual environments is the venv module from the standard Python distribution:

```
$ python3 -m venv ~/venv-valjean
$ source ~/venv-valjean/bin/activate
(valjean) $ pip install /path/to/valjean-x.y.z
```

Note: Ancient versions of `pip` (<19.0) will not be able to install *valjean*, because *valjean* uses a `pyproject.toml` file to describe the build, as specified in the **PEP 517** and **PEP 518** formats. If you are using an old version of `pip`, you should upgrade it (after activating the virtual environment) with:

```
(venv-valjean) $ pip install --upgrade pip
```

2.1.5 Prerequisites

valjean depends on a number of Python packages. You don't have to do anything special to install most of these packages, since `pip` should take care of everything. The one exception for the moment is h5py, which will not work unless the HDF5 library is installed on your machine.

2.1.6 Using conda

It is also possible to use *valjean* from a `conda` environment. The first step is to install miniconda or anaconda. The former is a light installation of python and only required packages will be installed. The latter is a full installation and can be used offline. Once the installation is done, you should run:

```
$ source PATH/TO/CONDA/bin/activate
```

unless you have set up your shell to do that automatically for you.

The recommended way to install *valjean* with `conda` is to create a `conda` environment for the package and all of its dependencies:

```
(base) $ conda create -n MY_ENV python=PY_VERSION
(base) $ conda activate MY_ENV
(MY_ENV) $ conda install -c file://PATH/TO/valjean-DETAILS.tar.bz2 --use-local valjean
```

`DETAILS` stands for v*VERSION-NUMBER_HASH_pyPY_VERSION* with:

- `VERSION`: last tag from valjean in the branch used to build the archive
- `NUMBER`: number of commits since this tag
- `HASH`: short hash of the commit used

- `PY_VERSION`: python version used to build the archive, the version used for the installation should be the same.

This procedure should allow to use *valjean* from the python interpreter, from a jupyter notebook or directly with the `valjean` command.

Note: only the *valjean* package is installed at that step, the others (*numpy*, *pyparsing*, ...) will be installed when running *valjean*. If you want to use *valjean* directly in python you'll probably need to install the required packages using `conda install PACKAGE`.

An offline installation is possible adding the `--offline` option in the installation command line. As a consequence updates of packages won't be possible, i.e. they will come from the available ones in the local installation of `conda`. The Python version of the package should probably be the default one of `conda`.

The conda package is not editable. If you prefer to use an editable version of *valjean* associated with conda it would probably be easier to use conda to get the python version, then install the package via `pip` or `poetry`. Virtual environnment have to be treated carefully in that case. To get an editable version offline fully working it might be necessary to install all (direct and indirect) dependencies.

2.1.7 Checking package integrity

The `md5sum` of the archives (`pip` or `conda` installation) are given in Tuleap. To check them, just type `md5sum MY_ARCHIVE` and compare the obtained hash with the one stored on Tuleap.

2.2 Documentation

If you are reading this, chances are high that you already know where to look for the documentation, but anyway: the documentation is distributed along with the sources of *valjean*, in the `doc` folder. You will find the documentation in reStructuredText in `doc/src`, and the HTML version in `doc/build/html/index.html`.

You can also find it in the valjean package release on Tuleap under the *Fichiers* area in the valjean project.

If, for some reason, the HTML documentation isl missing, you can generate it by running

```
$ cd /path/to/valjean-x.y.z
$ ./setup.py build_sphinx
```

or

```
$ cd /path/to/valjean-x.y.z/doc/src
$ make html
```

2.3 The `valjean` executable

Todo: Write documentation for the executable.

3 Examples (in French)

3.1 Notebooks

3.1.1 Lecture des fichiers *rates* d'Apollo3 grâce au `Picker`

La lecture des fichiers *rates* d'Apollo3 (réseau) est possible dans *valjean*, ainsi que les différents autres formats HDF5.

Deux modes de lecture sont possibles :

- lecture de tout le fichier HDF5 et stockage des résultats dans un `Browser`, récupération des résultats sous forme de `Dataset` grâce au `Browser` -> utilisation du `Reader`
- lecture d'un résultat ou de plusieurs résultats donnés à partir du HDF5 pour une utilisation directe sous forme de `Dataset` -> `Picker`

La seconde méthode est bien plus rapide car elle ne nécessite pas de charger l'intégralité du fichier et bénéficie des accès de lecture du HDF5. Cet exemple se concentre sur cette seconde méthode.

Le `Picker` : résumé

À utiliser notamment si vous connaissez les métadonnées du résultat qui vous intéresse : `'zone'`, `'output'`, `'result_name'`, etc.

[1]:
```
from valjean.eponine.apollo3.hdf5_picker import Picker
```

[2]:
```
ap3p = Picker("Mosteller.hdf")
```

```
* Reading Mosteller.hdf
* HDF5 loading done in 0.001324 s
```

La méthode à utiliser pour récupérer les résultats est `pick_standard_value` :

[3]:
```
keff = ap3p.pick_standard_value(output='output_1', zone='totaloutput', result_name=
↪'KEFF')
```

Attention : les strings correspondant aux différents arguments (``'output'``, ``'result_name'``, etc) sont celles stockées dans le fichier, en capitales pour les résultats la plupart du temps.

Les résultats correspondants à des taux microscopiques (par isotopes) sont souvent en minuscules avec la première lettre en majuscules.

Inspection du fichier HDF5

Il est tout de même possible d'inspecter le fichier (mais cela peut prendre un peu de temps).

[4]:
```
print(ap3p.zones(output='output_0'))
print(ap3p.zones(output='output_2'))
```

```
['1', '2', '3', 'totaloutput']
['1', '2', '3', 'totaloutput']
```

Illustration pour l'output `'output_0'`.

[5]:
```
print(f"isotopes dans la zone 3: {ap3p.isotopes(output='output_0', zone='3')}")
print(f"isotopes dans la zone 1: {ap3p.isotopes(output='output_0', zone='1')}")
```

```
isotopes dans la zone 3: ['H2O', 'B10', 'B11', 'macro']
isotopes dans la zone 1: ['U234', 'U235', 'U238', 'Pu239', 'Pu241', 'Pu240', 'Pu242',
↪'O16', 'macro']
```

Recherche des résultats macroscopiques (pas d'isotope de mentionné) :

[6]:
```
print(f"résultats dans la zone 1: {ap3p.results(output='output_0', zone='1')}")
print(f"résultats pour totaloutput: {ap3p.results(output='output_0', zone='totaloutput
↪')}")
```

```
résultats dans la zone 1: ['FLUX']
résultats pour totaloutput: ['ABSORPTION', 'FLUX', 'KEFF', 'PRODUCTION']
```

Recherche des résultats macroscopiques et microscopiques (isotope requis) :

[7]:
```
print(f'resultats sans isotopes: {ap3p.results(output="output_0", zone="3")}')
print(f'resultats pour B10: {ap3p.results(output="output_0", zone="3", isotope="B10")}
↪')
```

```
resultats sans isotopes: ['FLUX']
resultats pour B10: ['Absorption', 'concentration']
```

Sélection des résultats

Récupération d'un résultat sous forme de `Dataset` : utilisation de la méthode `pick_standard_value` pour les fichiers *rates* standards

Exemple sur les k_{eff}

[8]:
```
keff_o0 = ap3p.pick_standard_value(output='output_0', zone='totaloutput', result_name=
↪'KEFF')
print(keff_o0)
```

```
value: 1.235028e+00, error:    nan, bins: OrderedDict(), type: <class 'numpy.float32'>
↪,name: , what: keff
```

[9]:
```
keff_o1 = ap3p.pick_standard_value(output='output_1', zone='totaloutput', result_name=
↪'KEFF')
print(keff_o1)
```

```
value: 1.247908e+00, error:    nan, bins: OrderedDict(), type: <class 'numpy.float32'>
↪,name: , what: keff
```

[10]:
```
print(f"Difference between keffs (output_0, output1) = "
      f"{(1/keff_o0.value - 1/keff_o1.value)*1e5:.0f} pcm")
```

```
Difference between keffs (output_0, output1) = 836 pcm
```

Exemple sur les flux (arrays)

[11]:
```
flux = ap3p.pick_standard_value(output='output_0', zone='1', result_name='FLUX')
print(flux)
print(flux.value)
flux.name = 'output 0'
```

```
shape: (10,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['groups: [0 1 2 3 4 5 6 7␣
↪8 9]'], name: , what: flux
[  5.168701    45.47962    174.16776      3.8427184    0.37652907
   0.4066852    0.73074013   1.4990817    3.1569755    1.5008503 ]
```

[12]:
```
absB10 = ap3p.pick_standard_value(output='output_0', zone='3', result_name='Absorption
↪', isotope='B10')
print(absB10)
print(absB10.value)
```

```
shape: (10,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['groups: [0 1 2 3 4 5 6 7␣
↪8 9]'], name: , what: absorption
[3.4434328e-05 3.5456420e-04 1.6647208e-01 6.4293027e-02 9.4167180e-03
 1.5479327e-02 4.2232495e-02 6.3232511e-02 1.6776539e-01 1.6186915e-01]
```

Comparaison de résultats

Les tests disponibles dans *valjean* peuvent tout à fait être appliqués aux résultats issus du `Reader` ou du `Picker`.

Ici `TestApproxEqual` : les résultats sont des `float` sans erreur associée (mise à `nan` par défaut).

[13]:
```
flux_1 = ap3p.pick_standard_value(output='output_1', zone='1', result_name='FLUX')
flux_1.name = 'output 1'
```

[14]:
```
flux_2 = ap3p.pick_standard_value(output='output_2', zone='1', result_name='FLUX')
flux_2.name = 'output 2'
```

Import du test et des représentations (voir les autres notebooks)

[15]:
```
from valjean.gavroche.test import TestApproxEqual
from valjean.javert.representation import FullRepresenter
from valjean.javert.rst import RstFormatter
from valjean.javert.mpl import MplPlot
from valjean.javert.verbosity import Verbosity

frepr = FullRepresenter()
rstformat = RstFormatter()
```

[16]:
```
taeq_res = TestApproxEqual(flux, flux_1, flux_2, name='TestApproxEqual',
                           description='Test le TestApproxEqual sur les flux',␣
↪rtol=1e-2).evaluate()
print(bool(taeq_res))  # expected: False
```

```
False
```

[17]:
```
aeqrepr = frepr(taeq_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une liste␣
↪de templates
aeqrst = rstformat.template(aeqrepr[1])
print(aeqrst)
```

```
.. role:: hl

.. table::
    :widths: auto

    ======  ===========  ===========  ======================  ===========  ␣
↪=======================
    groups   output 0     output 1    approx equal(output 1)?   output 2    approx␣
↪equal(output 2)?
    ======  ===========  ===========  ======================  ===========  ␣
↪=======================
         0       5.1687       5.1609                     True      4.78112           ␣
↪  :hl:`False`
         1      45.4796      45.4164                     True      46.6316           ␣
↪  :hl:`False`
         2      174.168      174.516                     True      188.355           ␣
↪  :hl:`False`
         3      3.84272      3.89218             :hl:`False`      5.57925           ␣
↪  :hl:`False`
         4     0.376529     0.383804             :hl:`False`      0.87789           ␣
↪  :hl:`False`
         5     0.406685     0.413455             :hl:`False`      1.69239           ␣
↪  :hl:`False`
         6      0.73074     0.733685                     True      5.25886           ␣
↪  :hl:`False`
         7      1.49908      1.51683             :hl:`False`      6.91474           ␣
↪  :hl:`False`
         8      3.15698      3.20994             :hl:`False`      13.5553           ␣
↪  :hl:`False`
         9      1.50085      1.53149             :hl:`False`      6.58862           ␣
↪  :hl:`False`
```

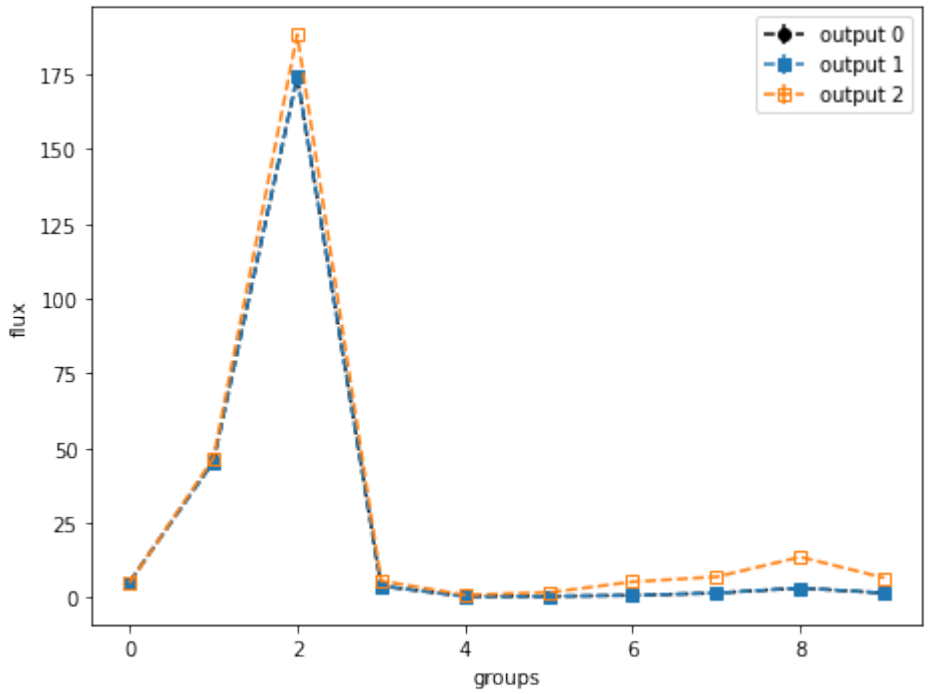
(continues on next page)

(continued from previous page)

```
    ======   ===========   ===========   ======================   ===========
↪========================
```

[18]:
```
mpl = MplPlot(aeqrepr[0]).draw()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/python3.8/
↪site-packages/matplotlib/axes/_base.py:2532: UserWarning: Warning: converting a
↪masked element to nan.
  xys = np.asarray(xys)
```



3.1.2 Lecture des fichiers *rates* d'Apollo3 : le Reader

La lecture des fichiers *rates* d'Apollo3 (réseau) est possible dans *valjean*, ainsi que les différents autres formats HDF5.

Deux modes de lecture sont possibles :

- lecture de tout le fichier HDF5 et stockage des résultats dans un `Browser`, récupération des résultats sous forme de `Dataset` grâce au `Browser` -> utilisation du `Reader`
- lecture d'un résultat ou de plusieurs résultats donnés à partir du HDF5 pour une utilisation directe sous forme de `Dataset` -> `Picker`

La seconde méthode est bien plus rapide car elle ne nécessite pas de charger l'intégralité du fichier et bénéficie des accès de lecture du HDF5.

Cet exemple se concentre sur la première méthode.

Le Reader : résumé

[1]:
```
from valjean.eponine.apollo3.hdf5_reader import Reader
```

Lecture d'un fichier *rates* typique : cas Mosteller, avec isotopes particularisés.

[2]:
```
ap3r = Reader("full_rates.hdf")
```

```
* Reading full_rates.hdf
* hdf5 loading done in 0.000718 s
* Successful reading in 0.018925 s
```

Transformation en `Browser`

[3]:
```
ap3b = ap3r.to_browser()
```

[4]:
```
print(ap3b)
```

```
Browser object -> Number of content items: 39, data key: 'results', available␣
↪metadata keys: ['index', 'isotope', 'output', 'result_name', 'zone']
               -> Number of globals: 2
```

39 = nombre de résultats total dans le fichier.

Il est possible de faire les deux précédentes étapes en une seule :

[5]:
```
from valjean.eponine.apollo3.hdf5_reader import hdf_to_browser
htb = hdf_to_browser("full_rates.hdf")
```

```
* Reading full_rates.hdf
* hdf5 loading done in 0.000338 s
* Successful reading in 0.016541 s
```

Inspection du fichier

On peut inspecter davantage le fichier grâce au `Browser` et connaître notamment les valeurs possibles des clefs du `Browser`.

[6]:
```
print(f'isotopes: {list(ap3b.available_values("isotope"))}')
print(f'resultats: {list(ap3b.available_values("result_name"))}')
print(f'zones: {list(ap3b.available_values("zone"))}')
print(f'outputs: {list(ap3b.available_values("output"))}')
```

```
isotopes: ['TotalResidual_reduced_chain', 'I135', 'Sm149', 'U238', 'Xe135', 'macro']
resultats: ['concentration', 'flux', 'absorption', 'diffusion', 'nexcess', 'fission',
↪'fissionspectrum', 'nufission', 'total', 'current', 'keff', 'kinf', 'production',
↪'surfflux']
zones: ['q', 'totaloutput']
outputs: ['output_0']
```

Toutes les combinaisons ne sont pas possibles.

Pour sélectionner un résultat :

- si on ne sélectionne qu'un seul résultat : méthode `select_by` → dictionnaire correspondant au résultat demandé

- si on en sélectionne plusieurs en vue d'une sélection plus raffinée ensuite : `filter_by` → sous-`Browser` dont la liste de résultats a été réduite à ceux correspondant à la sélection demandée

Le résultat, sous forme de `Dataset`, se trouve sous la clef `'results'`.

[7]:
```
sb_totout = ap3b.filter_by(zone='totaloutput')
print(sb_totout)
print(f'resultats: {list(sb_totout.available_values("result_name"))}')
print(f'outputs: {list(sb_totout.available_values("output"))}')
```

```
Browser object -> Number of content items: 7, data key: 'results', available metadata␣
↪keys: ['index', 'output', 'result_name', 'zone']
               -> Number of globals: 2
resultats: ['absorption', 'current', 'flux', 'keff', 'kinf', 'production', 'surfflux']
outputs: ['output_0']
```

[8]:
```
keffs = sb_totout.filter_by(result_name="keff")
print(len(keffs))
```

```
1
```

[9]:
```
keff = sb_totout.select_by(result_name="keff")
print(keff)
```

```
{'result_name': 'keff', 'results': class: <class 'valjean.eponine.dataset.Dataset'>,␣
↪data type: <class 'numpy.float32'>
value: 1.321053e+00, error:    nan, bins: OrderedDict()
name: '', what: 'keff'
, 'zone': 'totaloutput', 'output': 'output_0', 'index': 3}
```

[10]:
```
sb_q = ap3b.filter_by(zone='q')
print(sb_q)
```

```
Browser object -> Number of content items: 32, data key: 'results', available␣
↪metadata keys: ['index', 'isotope', 'output', 'result_name', 'zone']
               -> Number of globals: 2
```

[11]:
```
print(f'isotopes: {list(sb_q.available_values("isotope"))}')
print(f'resultats: {list(sb_q.available_values("result_name"))}')
print(f'outputs: {list(sb_q.available_values("output"))}')
```

```
isotopes: ['TotalResidual_reduced_chain', 'I135', 'Sm149', 'U238', 'Xe135', 'macro']
resultats: ['concentration', 'flux', 'absorption', 'diffusion', 'nexcess', 'fission',
↪'fissionspectrum', 'nufission', 'total']
outputs: ['output_0']
```

[12]:
```
sb_xe135 = sb_q.filter_by(isotope='Xe135')
print(f'resultats pour Xe135: {list(sb_xe135.available_values("result_name"))}')
```

```
resultats pour Xe135: ['concentration', 'absorption', 'diffusion']
```

[13]:
```
sb_u238 = sb_q.filter_by(isotope='U238')
print(f'resultats pour U238: {list(sb_u238.available_values("result_name"))}')
```

```
resultats pour U238: ['concentration', 'absorption', 'diffusion', 'fission',
↪'fissionspectrum', 'nexcess', 'nufission']
```

Sélection des résultats

La liste des résultats est disponible dans `content` dans le `Browser` ou directement grâce à la méthode `select_by` si la sélection est réduite à un résultat unique.

Les résultats sont sotckés sous la clef `'results'` sous forme de `Dataset`.

La plupart des résultats sont donnés par groupes d'énergie. Les intervalles (`bins` dans le `Dataset`) correspondent à l'index des groupes.

Certaines quantités ont différents axes :

- k_{eff} et k_{inf} : grandeurs scalaires, pas d'axes
- diffusion : le nombre d'anisotropies sur lequel le calcul est fait est pris en compte, s'il est différent de 1 les axes seront (anisotropies, groupes)
- flux surfacique : les axes sont (groupes, surfaces), surfaces contenant l'index des surfaces
- courant : les axes sont (groupes, surfaces, direction), surfaces contenant l'index des surfaces et direction (*incoming* et *leaving*)

[14]:
```
xe135_abs = sb_q.select_by(isotope='Xe135', result_name='absorption')
print(list(xe135_abs.keys()))
xe135_abs = xe135_abs['results']
print(xe135_abs.shape)  # 26 groupes
print(xe135_abs.what)
xe135_abs.name = '$^{135}$Xe'
```

```
['result_name', 'results', 'isotope', 'zone', 'output', 'index']
(26,)
absorption
```

[15]:
```
macro_diff = sb_q.select_by(isotope='macro', result_name='diffusion')
print(macro_diff['results'].what)
print(macro_diff['results'].shape)
print(list(macro_diff['results'].bins.keys()))
```

```
diffusion
(4, 26)
['anisotropies', 'groups']
```

[16]:
```
surf_flux = sb_totout.select_by(result_name='surfflux')
print(surf_flux['results'].what)
print(surf_flux['results'].shape)
print(list(surf_flux['results'].bins.keys()))
```

```
surfflux
(26, 220)
['groups', 'surfaces']
```

[17]:
```
current = sb_totout.select_by(result_name='current')['results']
print(current.what)
print(current.shape)
print(current.bins)
```

```
current
(26, 220, 2)
OrderedDict([('groups', array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
```

(continues on next page)

(continued from previous page)

```
↪ 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25])), ('surfaces', array([  0,   1,   2,   3,␣
↪  4,   5,   6,   7,   8,   9,  10,  11,  12,
        13,  14,  15,  16,  17,  18,  19,  20,  21,  22,  23,  24,  25,
        26,  27,  28,  29,  30,  31,  32,  33,  34,  35,  36,  37,  38,
        39,  40,  41,  42,  43,  44,  45,  46,  47,  48,  49,  50,  51,
        52,  53,  54,  55,  56,  57,  58,  59,  60,  61,  62,  63,  64,
        65,  66,  67,  68,  69,  70,  71,  72,  73,  74,  75,  76,  77,
        78,  79,  80,  81,  82,  83,  84,  85,  86,  87,  88,  89,  90,
        91,  92,  93,  94,  95,  96,  97,  98,  99, 100, 101, 102, 103,
       104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116,
       117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129,
       130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142,
       143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155,
       156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 168,
       169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181,
       182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194,
       195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207,
       208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219])), ('direction',␣
↪array(['incoming', 'leaving'], dtype='<U8'))])
```

Comparaison de résultats

Les tests disponibles dans *valjean* peuvent tout à fait être appliqués aux résultats issus du `Reader` ou du `Picker`.

Ici `TestApproxEqual` : les résultats sont des `float` sans erreur associée (mise à `nan` par défaut).

Import du test et des représentations (voir les autres notebooks)

[18]:
```
from valjean.gavroche.test import TestApproxEqual
from valjean.javert.representation import FullRepresenter
from valjean.javert.rst import RstFormatter
from valjean.javert.mpl import MplPlot
from valjean.javert.verbosity import Verbosity

frepr = FullRepresenter()
rstformat = RstFormatter()
```

Comparaison des aborptions pour différents isotopes

[19]:
```
sm149_abs = sb_q.select_by(isotope='Sm149', result_name='absorption')['results']
sm149_abs.name = '$^{149}$Sm'
i135_abs = sb_q.select_by(isotope='I135', result_name='absorption')['results']
i135_abs.name = '$^{135}$I'
```

[20]:
```
taeq_res = TestApproxEqual(xe135_abs, sm149_abs, i135_abs, name='TestApproxEqual',
                           description='Test le TestApproxEqual sur les absorptions',␣
```

(continues on next page)

(continued from previous page)

```
↪rtol=1e-2).evaluate()
print(bool(taeq_res))  # expected: False
```

```
False
```

Ajout d'une fonction pour passer en échelle logarithmique l'axe des ordonnées vu les variations du spectre.

[21]:
```
from valjean.javert import plot_repr as pltr
def log_post(templates, tres):
    pltr.post_treatment(templates, tres)
    for templ in templates:
        templ.subplots[0].attributes.logy = True
    return templates
```

[22]:
```
logaeqrepr = FullRepresenter(post=log_post)(taeq_res, verbosity=Verbosity.FULL_
↪DETAILS)
aeqrst = rstformat.template(logaeqrepr[1])
print(aeqrst)
```

```
.. role:: hl

.. table::
    :widths: auto

    ======  ===========  ===========  ========================  ===========
↪=========================
    groups  $^{135}$Xe   $^{149}$Sm   approx equal($^{149}$Sm)?   $^{135}$I   approx
↪equal($^{135}$I)?
    ======  ===========  ===========  ========================  ===========
↪=========================
         0      331.667      100.186                :hl:`False`      620.565
↪     :hl:`False`
         1      635.539      1311.99                :hl:`False`      2837.33
↪     :hl:`False`
         2      382.903      2705.19                :hl:`False`      1271.12
↪     :hl:`False`
         3      1159.48      15132.2                :hl:`False`      1280.17
↪     :hl:`False`
         4      1520.81      25141.5                :hl:`False`      1067.43
↪     :hl:`False`
         5      1714.74      27434.2                :hl:`False`      1458.82
↪     :hl:`False`
         6      2251.86      28455.9                :hl:`False`      2287.66
↪     :hl:`False`
         7      3895.33      47164.8                :hl:`False`       2626.3
↪     :hl:`False`
         8      15479.9       181483                :hl:`False`      116.592
↪     :hl:`False`
         9      42628.9       490062                :hl:`False`      228.763
↪     :hl:`False`
        10       146457  3.32796e+06                :hl:`False`      679.754
↪     :hl:`False`
        11  9.26102e+06  7.66008e+06                :hl:`False`      2146.12
↪     :hl:`False`
        12  1.89378e+08       492069                :hl:`False`      2541.36
↪     :hl:`False`
```

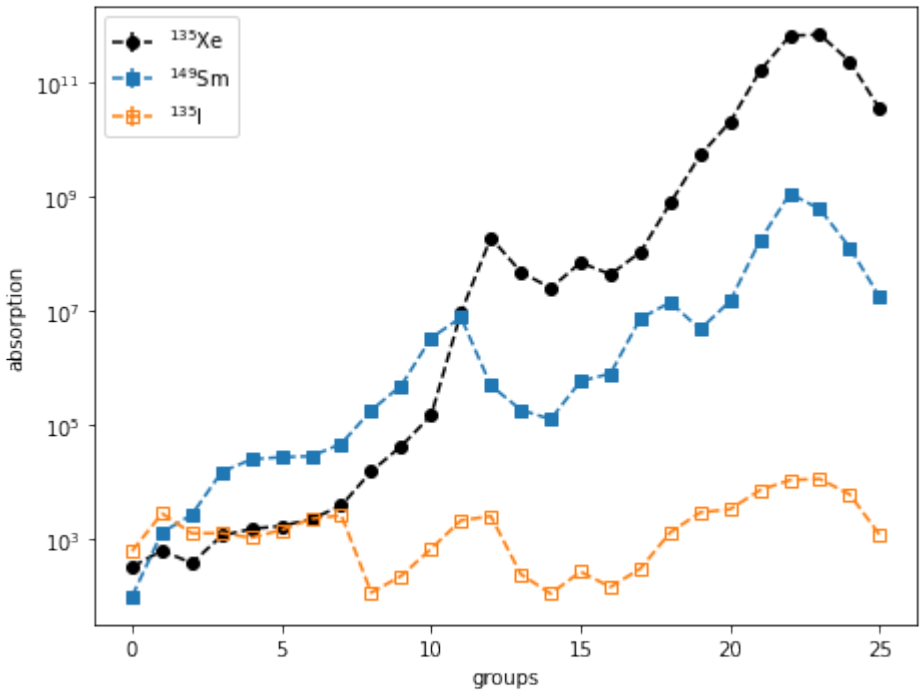
(continues on next page)

(continued from previous page)

```
        13  4.80216e+07       184737                :hl:`False`       245.923
↪     :hl:`False`
        14  2.49171e+07       126140                :hl:`False`       111.619
↪     :hl:`False`
        15  6.97439e+07       587325                :hl:`False`       271.223
↪     :hl:`False`
        16  4.34587e+07       794446                :hl:`False`        146.31
↪     :hl:`False`
        17  1.05815e+08  7.34886e+06                :hl:`False`       306.338
↪     :hl:`False`
        18  7.63633e+08  1.40516e+07                :hl:`False`        1322.4
↪     :hl:`False`
        19   5.2979e+09  4.73993e+06                :hl:`False`       2953.29
↪     :hl:`False`
        20  2.02055e+10  1.47097e+07                :hl:`False`       3355.33
↪     :hl:`False`
        21  1.57765e+11  1.68924e+08                :hl:`False`       7251.37
↪     :hl:`False`
        22  6.50512e+11   1.1291e+09                :hl:`False`         10870
↪     :hl:`False`
        23  6.84439e+11  6.02307e+08                :hl:`False`       11398.9
↪     :hl:`False`
        24    2.319e+11  1.28561e+08                :hl:`False`       6121.15
↪     :hl:`False`
        25  3.42816e+10  1.73609e+07                :hl:`False`       1198.21
↪     :hl:`False`
    ======  ===========  ===========  ========================  ===========
↪========================
```

[23]:
```
mpl = MplPlot(logaeqrepr[0]).draw()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/python3.8/
↪site-packages/matplotlib/axes/_base.py:2532: UserWarning: Warning: converting a
↪masked element to nan.
  xys = np.asarray(xys)
```



Comparaison de la diffusion pour les différentes valeurs d'anisotropie

La méthode `squeeze` permet de supprimer les dimensions triviales d'un `Dataset` (on ne comparera que des `Datasets` à une dimension ici).

[24]:
```
macro_diff_aniso = [macro_diff['results'][:1, :].squeeze()]
macro_diff_aniso[-1].name = 'anisotropie = 0'
macro_diff_aniso[-1].what = 'diffusion (macro)'
print(macro_diff_aniso[-1])
```

```
shape: (26,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['groups: [ 0  1  2  3  4␣
↪5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25]'], name: anisotropie␣
↪= 0, what: diffusion (macro)
```

[25]:
```
# anisotropie d'ordre 1
macro_diff_aniso.append(macro_diff['results'][1:2, :].squeeze())
macro_diff_aniso[-1].name = 'anisotropie = 1'
macro_diff_aniso[-1].what = 'diffusion (macro)'
# anisotropie d'ordre 2
macro_diff_aniso.append(macro_diff['results'][2:3, :].squeeze())
macro_diff_aniso[-1].name = 'anisotropie = 2'
macro_diff_aniso[-1].what = 'diffusion (macro)'
# anisotropie d'ordre 3
macro_diff_aniso.append(macro_diff['results'][3:, :].squeeze())
macro_diff_aniso[-1].name = 'anisotropie = 3'
macro_diff_aniso[-1].what = 'diffusion (macro)'
```

[26]:
```
taeq_res = TestApproxEqual(*macro_diff_aniso, name='TestApproxEqual',
                           description=("Test le TestApproxEqual sur la diffusion␣
↪macroscopique aux différents "
```

(continues on next page)

(continued from previous page)

```
                                        "ordres d'anisotropie"),
                           rtol=1e-2).evaluate()
print(bool(taeq_res))  # expected: False
```

```
False
```

[27]:
```
aeqrepr = frepr(taeq_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une liste␣
↪de templates
aeqrst = rstformat.template(aeqrepr[1])
print(aeqrst)
```

```
.. role:: hl

.. table::
    :widths: auto

    ======  ===============  ===============  ============================== ␣
↪===============  ==============================  =============== ␣
↪==============================
    groups  anisotropie = 0  anisotropie = 1  approx equal(anisotropie = 1)? ␣
↪anisotropie = 2  approx equal(anisotropie = 2)?  anisotropie = 3  approx␣
↪equal(anisotropie = 3)?
    ======  ===============  ===============  ============================== ␣
↪===============  ==============================  =============== ␣
↪==============================
         0      3.03941e+13      1.43217e+13                     :hl:`False`      8.
↪72864e+12                     :hl:`False`      5.80013e+12                     :hl:
↪`False`
         1      2.12017e+14      9.70531e+13                     :hl:`False`      5.
↪52561e+13                     :hl:`False`      2.46676e+13                     :hl:
↪`False`
         2      2.12051e+14       8.9062e+13                     :hl:`False`      4.
↪40241e+13                     :hl:`False`      1.25229e+13                     :hl:
↪`False`
         3      6.02824e+14      2.45831e+14                     :hl:`False`      1.
↪07446e+14                     :hl:`False`      8.16862e+12                     :hl:
↪`False`
         4      5.65327e+14      2.08306e+14                     :hl:`False`      8.
↪53618e+13                     :hl:`False`      2.39636e+12                     :hl:
↪`False`
         5       5.0581e+14      2.12395e+14                     :hl:`False`      7.
↪79372e+13                     :hl:`False`      1.03219e+11                     :hl:
↪`False`
         6      4.21794e+14      1.89427e+14                     :hl:`False`       7.
↪0032e+13                     :hl:`False`      2.68827e+10                     :hl:
↪`False`
         7      4.03747e+14      1.88263e+14                     :hl:`False`      6.
↪97694e+13                     :hl:`False`      8.80467e+09                     :hl:
↪`False`
         8      6.12812e+14      2.84296e+14                     :hl:`False`       1.
↪0541e+14                     :hl:`False`      6.97243e+09                     :hl:
↪`False`
         9      5.57152e+14      2.66414e+14                     :hl:`False`      9.
↪87652e+13                     :hl:`False`      2.07365e+09                     :hl:
↪`False`
        10      6.82393e+14      3.26437e+14                     :hl:`False`      1.
↪21391e+14                     :hl:`False`      1.53146e+12                     :hl:
```

(continues on next page)

(continued from previous page)

```
↪`False`
       11      6.89654e+14      3.37509e+14                    :hl:`False`      1.
↪28241e+14                    :hl:`False`      7.27767e+12                    :hl:
↪`False`
       12      3.07237e+14      1.40882e+14                    :hl:`False`      5.
↪52886e+13                    :hl:`False`      1.30106e+13                    :hl:
↪`False`
       13      2.30713e+13      1.00686e+13                    :hl:`False`      3.
↪84214e+12                    :hl:`False`      1.08319e+12                    :hl:
↪`False`
       14      1.03404e+13      4.49361e+12                    :hl:`False`      1.
↪70658e+12                    :hl:`False`      4.87285e+11                    :hl:
↪`False`
       15      2.41663e+13      1.04121e+13                    :hl:`False`       3.
↪9247e+12                     :hl:`False`      1.13473e+12                     :hl:
↪`False`
       16      1.25264e+13      5.34194e+12                    :hl:`False`      2.
↪00096e+12                    :hl:`False`      5.86095e+11                    :hl:
↪`False`
       17      2.53325e+13      1.07076e+13                    :hl:`False`      3.
↪97288e+12                    :hl:`False`      1.17618e+12                    :hl:
↪`False`
       18      9.91162e+13      4.06126e+13                    :hl:`False`       1.
↪4576e+13                     :hl:`False`      4.41702e+12                     :hl:
↪`False`
       19      1.87319e+14      7.15993e+13                    :hl:`False`      2.
↪33575e+13                    :hl:`False`      7.47564e+12                    :hl:
↪`False`
       20      1.88516e+14      6.65679e+13                    :hl:`False`      1.
↪92102e+13                    :hl:`False`      7.18871e+12                    :hl:
↪`False`
       21      3.54498e+14      1.13313e+14                    :hl:`False`       2.
↪8835e+13                     :hl:`False`      1.29191e+13                     :hl:
↪`False`
       22       4.7715e+14      1.33495e+14                    :hl:`False`       2.
↪8788e+13                     :hl:`False`      1.47012e+13                     :hl:
↪`False`
       23      4.67324e+14      1.06575e+14                    :hl:`False`      1.
↪65194e+13                    :hl:`False`      9.26162e+12                    :hl:
↪`False`
       24      2.54186e+14      3.85002e+13                    :hl:`False`       2.
↪0665e+12                     :hl:`False`      2.26773e+12                     :hl:
↪`False`
       25      5.54342e+13      3.54245e+12                    :hl:`False`      -1.
↪4409e+11                     :hl:`False`      3.48981e+11                     :hl:
↪`False`
   ======  ===============  ===============  ============================== ␣
↪===============  ==============================  =============== ␣
↪==============================
```

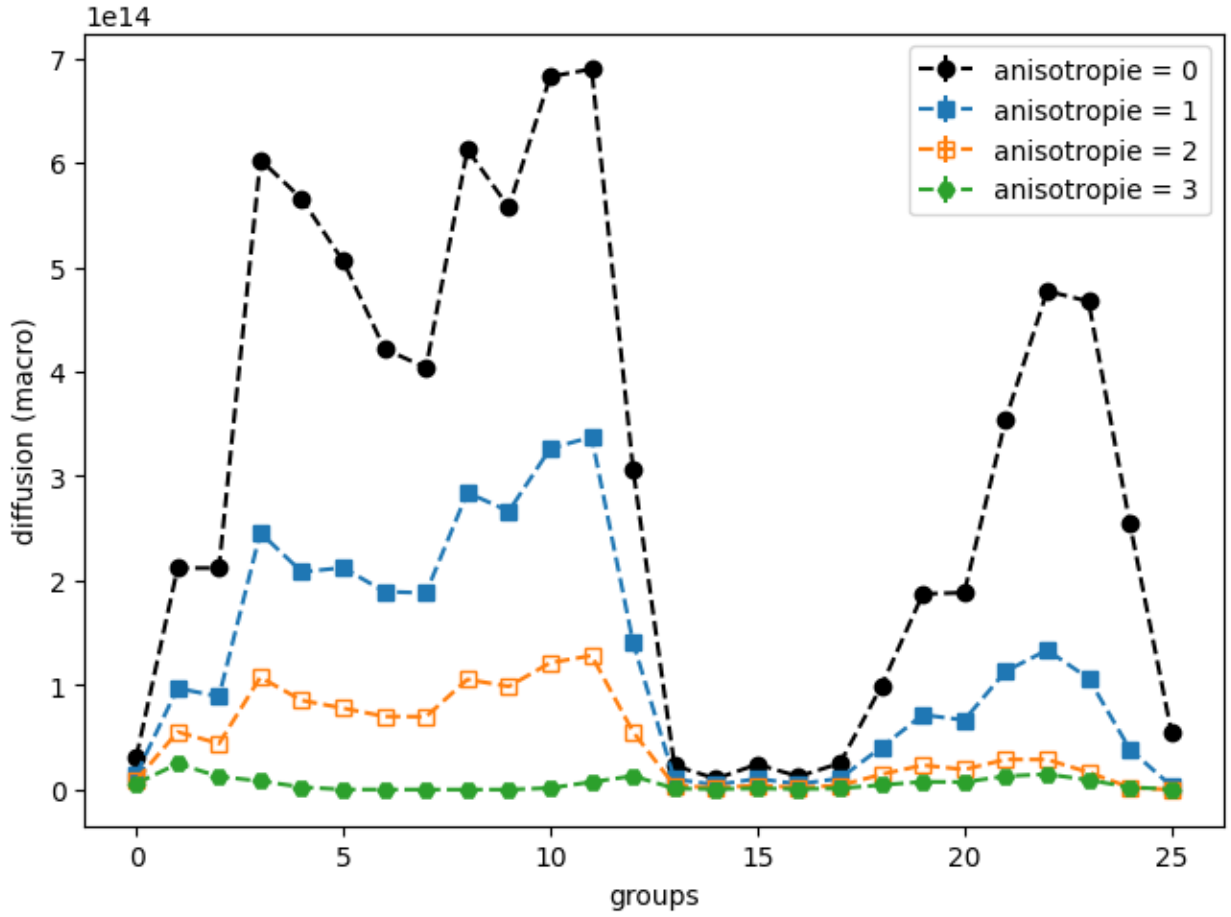
[28]:
```
mpl = MplPlot(aeqrepr[0]).draw()
```

```
/home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/python3.8/
↪site-packages/matplotlib/axes/_base.py:2532: UserWarning: Warning: converting a␣
↪masked element to nan.
```

(continues on next page)

(continued from previous page)

```
xys = np.asarray(xys)
```



Comparaison des courants entrants et sortants

Il s'agit d'une comparaison d'arrays en 2 dimensions, mais cela ne change pas les tests.

Pour une question de lisibilité du graphique on se restreint à 20 surfaces.

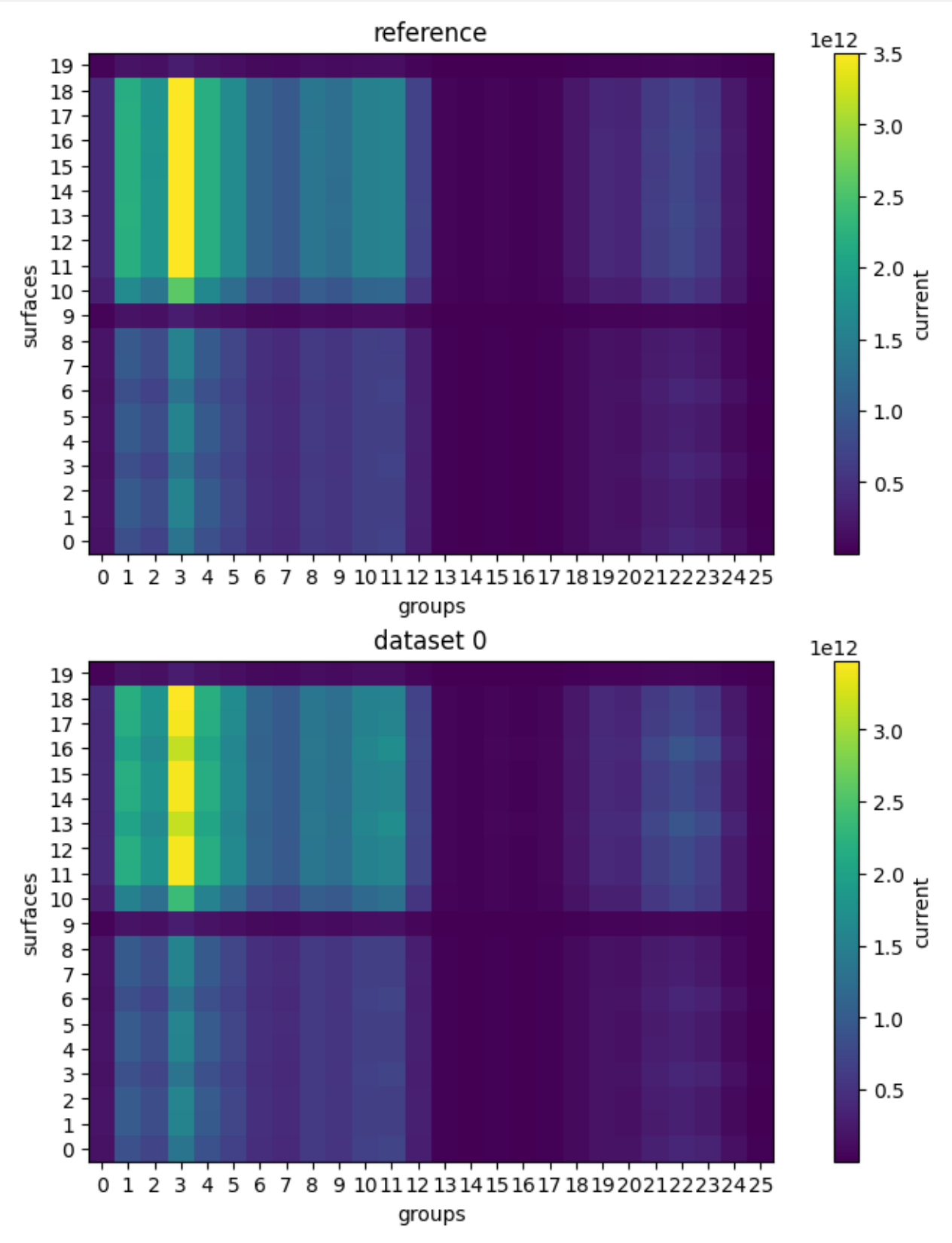
[29]:
```
current_in = current[:, :20, :1].squeeze()
current_out = current[:, :20, 1:].squeeze()
```

[30]:
```
caeq_res = TestApproxEqual(
    current_in, current_out, name='TestApproxEqual',
    description=("Test le TestApproxEqual sur la diffusion macroscopique aux␣
↪différents "
                 "ordres d'anisotropie"),
    rtol=1e-2).evaluate()
print(bool(caeq_res))  # expected: False
```

```
False
```

[31]:
```
caeqrepr = frepr(caeq_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une liste␣
↪de templates
caeqrst = rstformat.template(caeqrepr[1])
# print(caeqrst)
```

[32]:
```
mpl = MplPlot(caeqrepr[0]).draw()
```

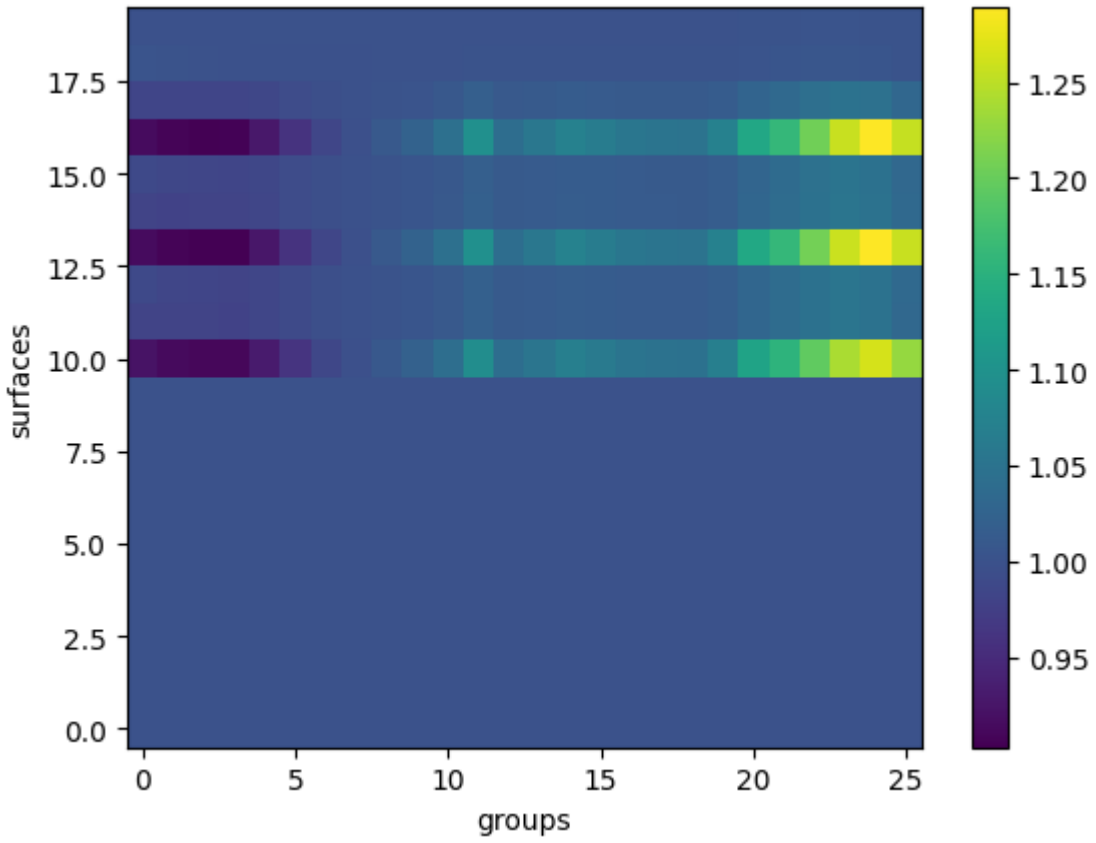

reference
surfaces
groups
current
1e12
dataset 0
surfaces
groups
current
1e12

[33]:
```
ratio = current_out / current_in
```

[34]:
```
import matplotlib.pyplot as plt
import numpy as np

fig, splts = plt.subplots()
bbins = np.broadcast_arrays(ratio.bins['groups'].reshape([26, 1]),
                            ratio.bins['surfaces'].reshape([20]))
h2d = splts.hist2d(
    bbins[0].flatten(), bbins[1].flatten(),
    bins=[np.append(ratio.bins['groups']-0.5, [ratio.bins['groups'][-1]+0.5]),
          np.append(ratio.bins['surfaces']-0.5, [ratio.bins['surfaces'][-1]+0.5])],
    weights=ratio.value.flatten())
cbar = fig.colorbar(h2d[3], ax=splts)
splts.set_xlabel('groups')
splts.set_ylabel('surfaces')
```

[34]:
```
Text(0, 0.5, 'surfaces')
```



3.1.3 Représentation de scores sur maillages

Par défaut la représentation des scores sur maillages dans valjean se fait centrée sur la maille. Si le score Tripoli4 a été tourné avec l'option `MESH_INFO` les coordonnées du centre de la maille sont en plus disponibles. Il est alors possible de faire un graphique davantage à l'échelle, même si les axes des mailles ne correspondent pas aux axes par défaut.

Différents exemples sont présentés pour illustrer ces représentations. Elles sont faites grâce à *matplotlib*, hors *valjean*.

Les imports par défaut

[1]:
```
import numpy as np
import matplotlib.pyplot as plt
from valjean.eponine.tripoli4.parse import Parser
from valjean.eponine.dataset import Dataset
```

Quelques fonctions

Certaines fonctionnalités ont été encapulées dans des fonctions pour plus de simplicité. Ce sont des exemples.

Sélection des intervalles

Cette fonction permet de sélectionner les intervalles que l'on veut représenter et permet de *squeezer* le `Dataset` et les coordonnées en même temps (suppresion des dimensions triviales). La sélection se fait sur un dictionnaire dont les clefs correspondent au nom des coordonnées à réduire et dont les valeurs correspondent aux indices des intervalles à sélectionner. Deux cas sont possibles :

- seul le deuxième intervalle est gardé sur l'axe `'u'` : `{'u': 1}`
- les intervalles d'indices 3 à 8 sont gardés sur l'axe `'v'` (qui en a au moins 9) : `{'v': (3, 9)}`

Dans le dernier cas les indices sont donnés à une `slice` (slice doc) dans la fonction.

[2]:
```
def select_space_bins(dset, coords, selection):
    """Select slice to be represented and squeeze dataset and coordinates accordingly.

    The slice can be done on one value or on a range of values if a tuple is given.

    :param Dataset dset: dataset
    :param np.ndarray coords: coordinates (structured array which names are the axis␣
↪names)
    :param dict selection: selection to be applied, keys correspond to the axis names
    :rtype: Dataset, np.ndarray
    :returns: sliced and squeezed dataset and coordinates
    """
    islices = tuple(slice(None) for _ in range(dset.ndim))
    oslices = list(islices)
    for key, val in selection.items():
        icoord = list(dset.bins.keys()).index(key)
```

(continues on next page)

(continued from previous page)

```
        if isinstance(val, int):
            oslices[icoord] = slice(val, val+1)
        elif isinstance(val, (list, tuple)):
            oslices[icoord] = slice(val[0], val[1])
    oslices = tuple(oslices)
    return dset[oslices].squeeze(), coords[oslices[:3]].squeeze()
```

Représentation des mailles individuelles

Le choix fait ici est de donner à la maille sa forme “réelle”. Pour cela le marqueur utilisé est défini comme un chemin entre les différents vertex donnés (marker doc et marker examples).

Dans les cas présentés ici le maillage est supposé régulier (une petite vérification est faite, cela suppose en général une précision dégradée sur l’estimation de la distance entre deux points consécutifs, ce qui apparaît dans l’argument `decimals` demandé par la fonction). Le chemin proposé est calculé à partir des 4 premiers points (2 sur chacune des 2 dimensions utilisées). Un parallélogramme est ainsi défini. Le sens de parcours des sommets est le sens direct pour s’assurer d’avoir un quadrilatère convexe.

Les dimensions fournies sont celles qui sont gardées.

[3]:
```
def vertices(coords, decimals, dims=('u', 'v')):
    """Compute the vertices of the elementary cell from regular coordinates.

    :param np.ndarray coords: coordinates (structured array which names are axis␣
↪names)
    :param int decimals: number of decimals to consider to identify unique␣
↪coordinates values
    :param list, tuple dims: dimension names of the corrdinated considered
    :rtype: list(tuple(float))
    :returns: verticies of the path matching the cell (4 points here)
    """
    uu = np.unique(np.diff(coords[dims[0]]).round(decimals))
    uv = np.unique(np.diff(coords[dims[1]]).round(decimals))
    if len(uu) != len(uv):
        print("Not a rectangle or parallelogram, returning arbitrary triangle")
        print(f'u: {uu}, v: {uv}')
        return [[-1, -1], [1, -1], [1, 1], [-1, -1]]
    if len(uu) > 2:
        print('Not a parallelogram, returning arbitrary triangle')
        print(uu, np.ediff1d(coords[dims[0]]).round(decimals))
        return [[-1, -1], [1, -1], [1, 1], [-1, -1]]
    f4pts = coords[:2, :2]
    meanu = np.mean([np.min(f4pts[dims[0]]), np.max(f4pts[dims[0]])])
    meanv = np.mean([np.min(f4pts[dims[1]]), np.max(f4pts[dims[1]])])
    # vertices: points rotated in direct order
    return [(f4pts[dims[0]][0, 0]-meanu, f4pts[dims[1]][0, 0]-meanv),
            (f4pts[dims[0]][1, 0]-meanu, f4pts[dims[1]][1, 0]-meanv),
            (f4pts[dims[0]][1, 1]-meanu, f4pts[dims[1]][1, 1]-meanv),
            (f4pts[dims[0]][0, 1]-meanu, f4pts[dims[1]][0, 1]-meanv)]
```

Cas d'une maille cartésienne alignée sur les axes

Lecture du jeu de données

[4]:
```
t4p_bd = Parser("box_dyn.res")
t4b_bd = t4p_bd.parse_from_index().to_browser()
```

```
* Parsing box_dyn.res
* Successful scan in 0.006736 s
* Successful parsing in 0.034147 s
```

Sélection du score (mesh)

[5]:
```
neut_flux = t4b_bd.select_by(score_name="neutron_flux_mesh_score")
print(neut_flux['results']['score_eintegrated'].shape)
nfm = neut_flux['results']['score_eintegrated'].squeeze()
print(nfm)
```

```
(3, 3, 3, 1, 1, 1, 1)
shape: (3, 3, 3), dim: 3, type: <class 'numpy.ndarray'>, bins: ['u: [0 1 2]', 'v: [0␣
↪1 2]', 'w: [0 1 2]'], name: , what: flux
```

Sélection / accès aux coordonnées associées

[6]:
```
nfc = neut_flux['results']['coordinates']
print(nfc.shape, nfc.dtype.names)
```

```
(3, 3, 3) ('u', 'v', 'w')
```

Les représentations proposées ici sont en 2 dimensions. Pour représenter complètement le score neuf graphiques sont nécessaires : 3 pour chaque couple de coordonnées. Pour simplifier seul un graphique pour chaque couple de données sera présenté.

[7]:
```
nfm_u0, nfc_u0 = select_space_bins(nfm, nfc, {'u': 0})
nfm_v0, nfc_v0 = select_space_bins(nfm, nfc, {'v': 0})
nfm_w0, nfc_w0 = select_space_bins(nfm, nfc, {'w': 0})
```

[8]:
```
print(nfm_u0)
print(nfc_u0.shape, nfc.dtype.names)
print('coordonnées restantes pour u=0')
print(nfc_u0)
print('coordonnées restantes pour w=0')
print(nfc_w0)
```

```
shape: (3, 3), dim: 2, type: <class 'numpy.ndarray'>, bins: ['v: [0 1 2]', 'w: [0 1 2]
↪'], name: , what: flux
(3, 3) ('u', 'v', 'w')
coordonnées restantes pour u=0
[[(-3.3333, -3.3333, -8.) (-3.3333, -3.3333,  0.) (-3.3333, -3.3333,  8.)]
 [(-3.3333, -0.    , -8.) (-3.3333, -0.    ,  0.) (-3.3333, -0.    ,  8.)]
 [(-3.3333,  3.3333, -8.) (-3.3333,  3.3333,  0.) (-3.3333,  3.3333,  8.)]]
coordonnées restantes pour w=0
[[(-3.3333, -3.3333, -8.) (-3.3333, -0.    , -8.) (-3.3333,  3.3333, -8.)]
 [(-0.    , -3.3333, -8.) (-0.    , -0.    , -8.) (-0.    ,  3.3333, -8.)]
 [( 3.3333, -3.3333, -8.) ( 3.3333, -0.    , -8.) ( 3.3333,  3.3333, -8.)]]
```

Calcul des sommets (vertex) du parallélogramme représentant la maille (ici un rectangle en fait).

[9]:
```
nfv_u0 = vertices(nfc_u0, 3, ['v', 'w'])
nfv_v0 = vertices(nfc_v0, 3, ['u', 'w'])
nfv_w0 = vertices(nfc_w0, 3, ['u', 'v'])
print(nfv_u0)
```

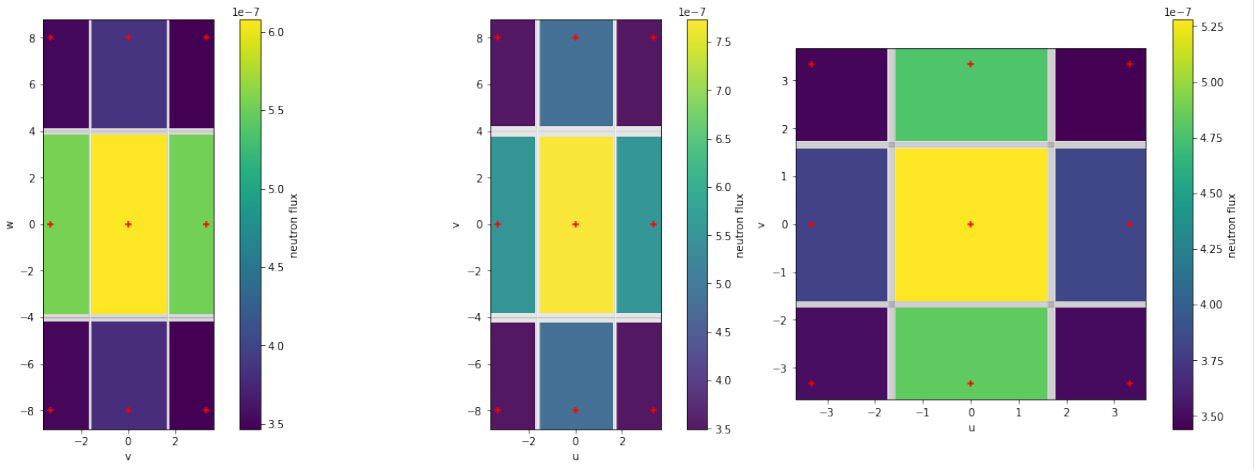
```
[(-1.66665, -4.0), (1.66665, -4.0), (1.66665, 4.0), (-1.66665, 4.0)]
```

Représentation graphique grâce à *matplotlib*.

Le graphique est représenté 3 fois :

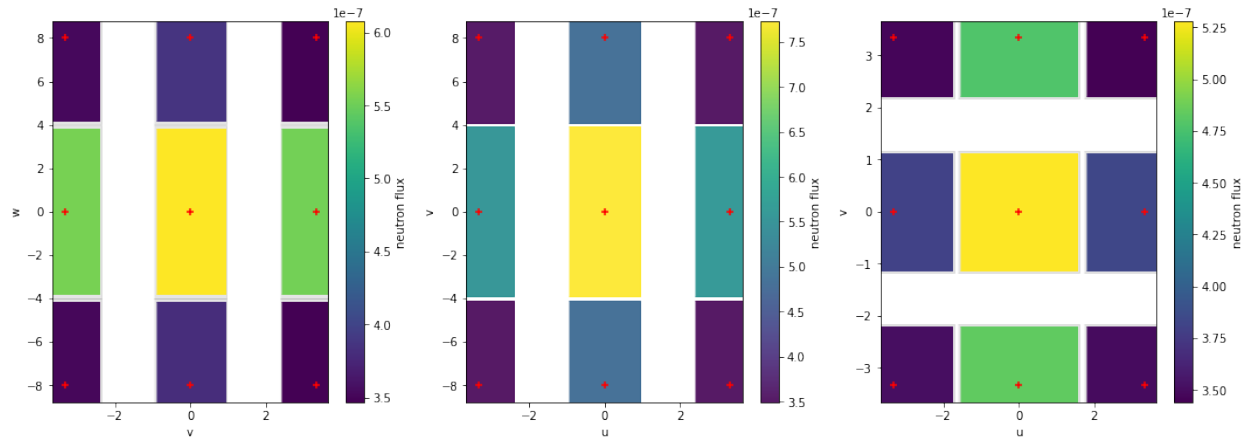
- marqueur correspondant au chemin défini plus haut en grisé transparent pour pouvoir ajuster sa taille (argument `s`), la transparence permettant de mieux visualiser les recouvrements
- marqueur correspondant au chemin défini plus haut avec le contenu attendu (argument `c`=couleur contenant la valeur du `Dataset`)
- marqueur + rouge pour visualiser le centre de la maille

[10]:
```
fig, axs = plt.subplots(1, 3, figsize=(18, 6), subplot_kw={'aspect': "equal"},␣
↪constrained_layout=True)
axs[0].scatter(nfc_u0['v'].flatten(), nfc_u0['w'].flatten(),
                   c="gray", s=30000, marker=nfv_u0, alpha=0.3)
im0 = axs[0].scatter(nfc_u0['v'].flatten(), nfc_u0['w'].flatten(),
                   c=nfm_u0.value.flatten(), s=28000, marker=nfv_u0)
axs[0].scatter(nfc_u0['v'].flatten(), nfc_u0['w'].flatten(), c="red", marker="+")
axs[0].set(xlabel='v', ylabel='w')
cbar0 = fig.colorbar(im0, ax=axs[0], label='neutron flux')
axs[1].scatter(nfc_v0['u'].flatten(), nfc_v0['w'].flatten(),
                   c="gray", s=30000, marker=nfv_v0, alpha=0.2)
im1 = axs[1].scatter(nfc_v0['u'].flatten(), nfc_v0['w'].flatten(),
                   c=nfm_v0.value.flatten(), s=27000, marker=nfv_v0, alpha=0.9)
axs[1].scatter(nfc_v0['u'].flatten(), nfc_v0['w'].flatten(), c="red", marker="+")
axs[1].set(xlabel='u', ylabel='v')
cbar1 = fig.colorbar(im1, ax=axs[1], label='neutron flux')
axs[2].scatter(nfc_w0['u'].flatten(), nfc_w0['v'].flatten(),
                 c="gray", s=24000, marker=nfv_w0, alpha=0.2,)
im2 = axs[2].scatter(nfc_w0['u'].flatten(), nfc_w0['v'].flatten(),
                   c=nfm_w0.value.flatten(), s=20000, marker=nfv_w0)
axs[2].scatter(nfc_w0['u'].flatten(), nfc_w0['v'].flatten(), c="red", marker="+")
axs[2].set(xlabel='u', ylabel='v')
cbar2 = fig.colorbar(im2, ax=axs[2], label='neutron flux')
```



Un des arguments les plus imporants ici est : `subplot_kw={'aspect': "equal"}` sans lui les axes sont optimisés et ne sont plus à l'échelle. L'effet sera plus flagrant plus loin, sur des mailles non rectangulaires.

[11]:
```
fig, axs = plt.subplots(1, 3, figsize=(18, 6))
axs[0].scatter(nfc_u0['v'].flatten(), nfc_u0['w'].flatten(),
                   c="gray", s=22000, marker=nfv_u0, alpha=0.2)
im0 = axs[0].scatter(nfc_u0['v'].flatten(), nfc_u0['w'].flatten(),
                   c=nfm_u0.value.flatten(), s=20000, marker=nfv_u0)
axs[0].scatter(nfc_u0['v'].flatten(), nfc_u0['w'].flatten(), c="red", marker="+")
axs[0].set(xlabel='v', ylabel='w')
cbar0 = fig.colorbar(im0, ax=axs[0], label='neutron flux')
im1 = axs[1].scatter(nfc_v0['u'].flatten(), nfc_v0['w'].flatten(),
                   c=nfm_v0.value.flatten(), s=21000, marker=nfv_v0, alpha=0.9)
axs[1].scatter(nfc_v0['u'].flatten(), nfc_v0['w'].flatten(), c="red", marker="+")
axs[1].set(xlabel='u', ylabel='v')
cbar1 = fig.colorbar(im1, ax=axs[1], label='neutron flux')
axs[2].scatter(nfc_w0['u'].flatten(), nfc_w0['v'].flatten(),
               c="gray", s=11000, marker=nfv_w0, alpha=0.2,)
im2 = axs[2].scatter(nfc_w0['u'].flatten(), nfc_w0['v'].flatten(),
                   c=nfm_w0.value.flatten(), s=10000, marker=nfv_w0)
axs[2].scatter(nfc_w0['u'].flatten(), nfc_w0['v'].flatten(), c="red", marker="+")
axs[2].set(xlabel='u', ylabel='v')
cbar2 = fig.colorbar(im2, ax=axs[2], label='neutron flux')
```



Cas d'une maille cartésienne dont les axes ont été tournés

Lecture du jeu de données

[12]:
```
t4p_em = Parser("extended_mesh_cartesian_info.res")
t4b_em = t4p_em.parse_from_index().to_browser()
```

```
* Parsing extended_mesh_cartesian_info.res
* Successful scan in 0.010152 s
* Successful parsing in 0.375536 s
```

Sélection du score

[13]:
```
phot_flux_m8 = t4b_em.select_by(score_name="mesh8_reg")
pfm8 = phot_flux_m8['results']['score_eintegrated']
cm8 = phot_flux_m8['results']['coordinates']
```

[14]:
```
print(pfm8.shape, cm8.shape)
```
```
(3, 2, 7, 1, 1, 1, 1) (3, 2, 7)
```

La représentation sera également faite dans le premier intervalle.

[15]:
```
pfm8u0, cm8u0 = select_space_bins(pfm8, cm8, {'u': 0})
pfm8v0, cm8v0 = select_space_bins(pfm8, cm8, {'v': 0})
pfm8w0, cm8w0 = select_space_bins(pfm8, cm8, {'w': 0})
print(pfm8u0.shape, cm8u0.shape)
```
```
(2, 7) (2, 7)
```

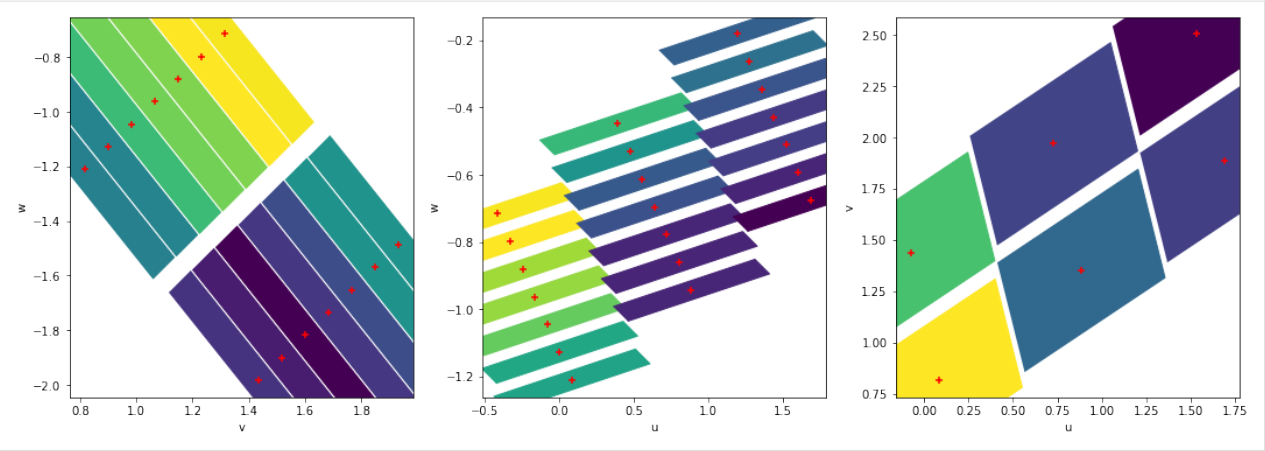
Calcul des sommets des parallélogrammes

[16]:
```
verts_u0 = vertices(cm8u0, 3, ['v', 'w'])
verts_v0 = vertices(cm8v0, 3, ['u', 'w'])
verts_w0 = vertices(cm8w0, 3, ['u', 'v'])
print(verts_u0)
print(verts_v0)
print(verts_w0)
```
```
[(-0.3498499999999999, 0.34450000000000003), (0.26735, -0.4269999999999998), (0.34985,
↪ -0.3444999999999998), (-0.26735, 0.42700000000000005)]
[(-0.35964999999999997, -0.17490000000000006), (0.44215, 0.09240000000000004), (0.
↪35964999999999997, 0.17490000000000006), (-0.44215, -0.09240000000000004)]
[(-0.32375, -0.5758500000000001), (0.47805, -0.04135000000000022), (0.32375, 0.57585),
↪ (-0.47805, 0.041349999999999776)]
```

Représentation **sans** `aspect='equal'`

[17]:
```
fig, axs = plt.subplots(1, 3, figsize=(18, 6))
axs[0].scatter(cm8u0['v'].flatten(), cm8u0['w'].flatten(),
                   c=pfm8u0.value.flatten(), s=35000, marker=verts_u0)
axs[0].scatter(cm8u0['v'].flatten(), cm8u0['w'].flatten(), c="red", marker="+")
axs[0].set(xlabel='v', ylabel='w')
axs[1].scatter(cm8v0['u'].flatten(), cm8v0['w'].flatten(),
                   c=pfm8v0.value.flatten(), s=18000, marker=verts_v0)
axs[1].scatter(cm8v0['u'].flatten(), cm8v0['w'].flatten(), c="red", marker="+")
axs[1].set(xlabel='u', ylabel='w')
axs[2].scatter(cm8w0['u'].flatten(), cm8w0['v'].flatten(),
                   c=pfm8w0.value.flatten(), s=30000, marker=verts_w0)
axs[2].scatter(cm8w0['u'].flatten(), cm8w0['v'].flatten(), c="red", marker="+")
axs[2].set(xlabel='u', ylabel='v')
```

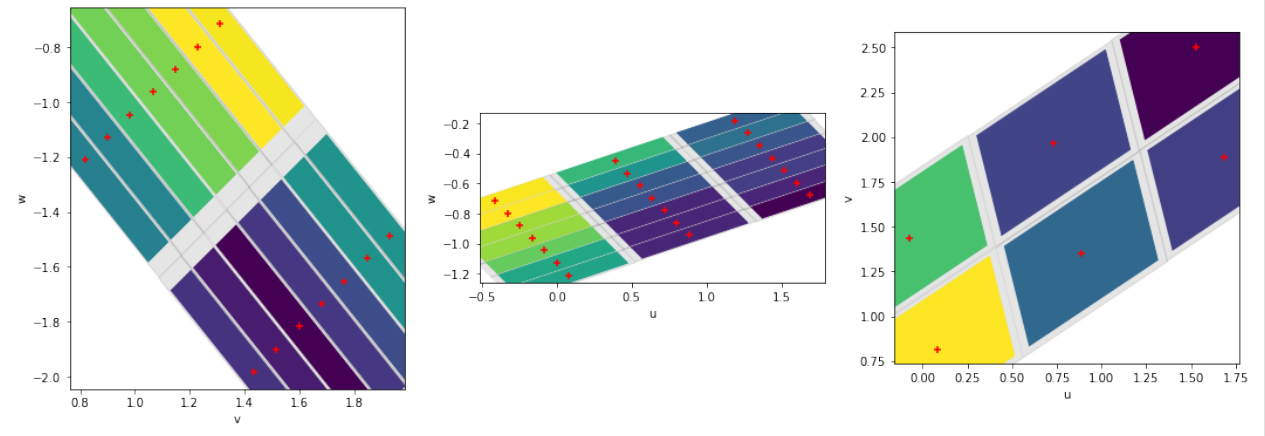
[17]:
```
[Text(0.5, 0, 'u'), Text(0, 0.5, 'v')]
```



Les parallélogrammes se chevauchent, cela ne correspond pas à ce que l'on attend -> `aspect='equal'`...

[18]:
```
fig, axs = plt.subplots(1, 3, figsize=(18, 6), subplot_kw={'aspect': "equal"})
axs[0].scatter(cm8u0['v'].flatten(), cm8u0['w'].flatten(),
                   c="gray", alpha=0.2, s=40000, marker=verts_u0)
axs[0].scatter(cm8u0['v'].flatten(), cm8u0['w'].flatten(),
                   c=pfm8u0.value.flatten(), s=30000, marker=verts_u0)
axs[0].scatter(cm8u0['v'].flatten(), cm8u0['w'].flatten(), c="red", marker="+")
axs[0].set(xlabel='v', ylabel='w')
axs[1].scatter(cm8v0['u'].flatten(), cm8v0['w'].flatten(),
                   c="gray", alpha=0.2, s=13000, marker=verts_v0)
axs[1].scatter(cm8v0['u'].flatten(), cm8v0['w'].flatten(),
                   c=pfm8v0.value.flatten(), s=10000, marker=verts_v0)
axs[1].scatter(cm8v0['u'].flatten(), cm8v0['w'].flatten(), c="red", marker="+")
axs[1].set(xlabel='u', ylabel='w')
axs[2].scatter(cm8w0['u'].flatten(), cm8w0['v'].flatten(),
                   c="gray", alpha=0.2, s=31000, marker=verts_w0)
axs[2].scatter(cm8w0['u'].flatten(), cm8w0['v'].flatten(),
                   c=pfm8w0.value.flatten(), s=25000, marker=verts_w0)
axs[2].scatter(cm8w0['u'].flatten(), cm8w0['v'].flatten(), c="red", marker="+")
axs[2].set(xlabel='u', ylabel='v')
```

[18]:
```
[Text(0.5, 0, 'u'), Text(0, 0.5, 'v')]
```



3.1.4 Godiva : analyse de k_{eff}

Le jeu de données utilisé est : heu-met-fast-001-godiva.

Les résultats à analyser sont les k_{eff} calculés par Tripoli-4.

Parsing du jeu de données et exploration des résultats

[1]:
```
from valjean.eponine.tripoli4.parse import Parser

# scan du jeu de données
t4p = Parser('heu-met-fast-001-godiva.res')
# parsing du dernier batch (par défaut, index=-1)
t4res = t4p.parse_from_index()
# clefs disponibles dans le dictionnaire de résultats
list(t4res.res.keys())
```

```
* Parsing heu-met-fast-001-godiva.res
* Successful scan in 0.021714 s
* Successful parsing in 0.030526 s
```

[1]:
```
['list_responses', 'keff_auto', 'batch_data', 'run_data']
```

Pour manipuler plus aisément les réponses de Tripoli-4 et en particulier les sélectionner on utilise un objet `Browser`.

[2]:
```
t4b = t4res.to_browser()
print(t4b)
```

```
Browser object -> Number of content items: 18, data key: 'results', available␣
↪metadata keys: ['index', 'keff_estimator', 'response_function', 'response_index',
↪'response_type']
               -> Number of globals: 6
```

Accès aux paramètres globaux du résultat :

[3]:
```
from pprint import pprint
pprint(t4b.globals)
```

```
{'batch_number': 2100,
 'edition_batch_number': 2100,
 'mean_weight_leak': {'score': 573.9424,
                      'sigma': 0.3446137,
                      'sigma%': 0.06004325},
 'name': '',
 'simulation_time': 30,
 'source_intensity': 12.56637}
```

Le contenu (soit les réponses) est caractérisé par des mots-clefs - valeurs :

[4]:
```
for k in list(t4b.keys()):
    print("{0} -> {1}".format(k, list(t4b.available_values(k))))
```

```
response_function -> ['PRODUCTION', 'ABSORPTION', 'LEAKAGE', 'LEAKAGE_INSIDE', 'NXN␣
↪EXCESS', 'FLUX TOTAL', 'ENERGY LEAKAGE', 'KEFFS']
response_type -> ['generic', 'keff', 'keff_auto']
response_index -> [0, 1, 2, 3, 4, 5, 6, 7]
index -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17]
```

(continues on next page)

(continued from previous page)

```
keff_estimator -> ['KSTEP', 'KCOLL', 'KTRACK', 'KSTEP-KCOLL', 'KSTEP-KTRACK', 'KCOLL-
↪KTRACK', 'full combination', 'MACRO KCOLL']
```

La sélection de la réponse se fait grâce aux mots-clefs - valeurs ci-dessus par les méthodes

- `filter_by` → sous-`Browser` correspondant à la sélection
- `select_by` → réponse unique correspondant à la sélection (exception s'il n'y a pas de correspondance ou si plusieurs réponses correspondent)

Sélection des k_{eff}

Deux types de résultats de k_{eff} sont disponibles :

- les k_{eff} qui apparaissent comme les réponses standard dans le listing de sortie de Tripoli-4, qui comportent normalement trois évaluations : `KSTEP`, `KCOLL` et `KTRACK` ainsi que leurs corrélations et le résultat de leur combinaison, appelés ici k_{eff} "génériques"
- les k_{eff} apparaissant le plus souvent en toute fin de listing, donc le *discard* est calculé automatiquement, de manière à en donner la meilleure estimation, appelés ici k_{eff} "automatiques"

k_{eff} "génériques"

[5]:
```
keffs = t4b.filter_by(response_function='KEFFS')
print(len(keffs))
```

```
7
```

Il y a 7 k_{eff} disponibles comme attendu (les 3 valeurs, les 3 combinaisons deux à deux et les corrélations associées et la combinaison des trois. Pour en connaître la liste précise et pouvoir les isoler le plus simple est d'utiliser la clef `'keff_estimator'` :

[6]:
```
pprint(list(keffs.available_values('keff_estimator')))
```

```
['KSTEP',
 'KCOLL',
 'KTRACK',
 'KSTEP-KCOLL',
 'KSTEP-KTRACK',
 'KCOLL-KTRACK',
 'full combination']
```

La sélection des k_{eff} avec les clefs `'response_function'` et `'keff_estimator'` donnant un unique résultat il est possible d'utiliser `select_by` pour le récupérer directement.

[7]:
```
kstep = t4b.select_by(response_function='KEFFS', keff_estimator='KSTEP')
print(list(kstep['results'].keys()))
ds_kstep = kstep['results']['keff']
ds_kstep.name='kstep'  # name is '' by default
print(ds_kstep)
```

```
['keff', 'correlation_keff', 'used_batches']
value: 9.978572e-01, error: 8.598928e-04, bins: OrderedDict(), type: <class 'numpy.
↪float64'>,name: kstep, what: keff
```

[8]:
```
kstep_kcoll = t4b.select_by(response_function='KEFFS', keff_estimator='KSTEP-KCOLL')
print(list(kstep_kcoll['results'].keys()))
print(kstep_kcoll['results']['keff'])
```

```
['keff', 'correlation_keff', 'used_batches']
value: 9.967598e-01, error: 6.801727e-04, bins: OrderedDict(), type: <class 'numpy.
↪float64'>,name: , what: keff
```

Pour obtenir la corrélation :

[9]:
```
print(kstep_kcoll['results']['correlation_keff'])
```

```
value: 7.913696e-01, error:    nan, bins: OrderedDict(), type: <class 'numpy.float64'>
↪,name: , what: correlation
```

Remarque : elle n'a pas d'erreur.

k_{eff} "automatiques"

[10]:
```
keffs = t4b.filter_by(response_type='keff_auto')
print(len(keffs))
pprint(list(keffs.available_values('keff_estimator')))
```

```
4
['KSTEP', 'KCOLL', 'KTRACK', 'MACRO KCOLL']
```

Dans ce cas il y a un autre estimateur en plus : `'MACRO_KCOLL'`. À noter également la présence du nombre de batches *discarded* puisqu'il est calculé par Tripoli-4.

Exemples de comparaison de k_{eff}

Différents tests sont disponibles dans *valjean*. Par exemple :

- `TestEqual` qui vérifie que les datasets sont égaux (ce test est plutôt prévu pour des valeurs entières comme les nombres de batches)
- `TestApproxEqual` qui vérifie que les datasets sont approximativement égaux (pertinent pour les `float` pour lesquels on n'a pas d'erreur associée, les corrélations de k_{eff} par exemple)
- `TestStudent` dans le cas où l'on veut prendre en compte les erreurs sur les valeurs

Pour tous ces tests il faut définir une référence, qui sera le premier dataset donné au test.

L'exemple ci-dessous présente la comparaison des k_{eff} obtenus grâce à la modélisation simplifiée utilisée ci-dessus à celle du modèle en couches décrit dans heu-met-fast-001-shell_model.

On utilisera les k_{eff} "génériques" en comparant tous les estimateurs un à un.

[11]:
```
sb = t4b.filter_by(response_function='KEFFS')
print('estimators values:', list(sb.available_values('keff_estimator')))
```

```
estimators values: ['KSTEP', 'KCOLL', 'KTRACK', 'KSTEP-KCOLL', 'KSTEP-KTRACK', 'KCOLL-
↪KTRACK', 'full combination']
```

Tous les k_{eff} sont récupérés et stockés dans une liste. L'estimateur sera utilisé comme `'name'` pour chaque `Dataset`.

[12]:
```
dsets = []
for keff in sb.content:
    dsets.append(keff['results']['keff'])
    dsets[-1].name=keff['keff_estimator']
# print(dsets)
```

Pour rendre le résultat plus lisible un seul `Dataset` va être construit, contenant tous les k_{eff} pour une modélisation donnée. Les bins correspondent alors aux noms des estimateurs, actuellement stockés dans la variable `'name'` des `Dataset`.

[13]:
```
from collections import OrderedDict
import numpy as np
from valjean.eponine.dataset import Dataset
```

[14]:
```
dset_simple = Dataset(value=np.array([k.value for k in dsets]),
                      error=np.array([k.error for k in dsets]),
                      bins=OrderedDict([('estimator', np.array([k.name for k in␣
↪dsets]))]),
                      name='Simple model', what='keff')
```

[15]:
```
print(dset_simple)
```

```
shape: (7,), dim: 1, type: <class 'numpy.ndarray'>, bins: ["estimator: ['KSTEP' 'KCOLL
↪' 'KTRACK' 'KSTEP-KCOLL' 'KSTEP-KTRACK' 'KCOLL-KTRACK' 'full combination']"], name:␣
↪Simple model, what: keff
```

Un `Dataset` similaire est construit pour la modélisation en couche.

[16]:
```
t4p_shell = Parser('heu-met-fast-001-shell_model.res')
t4b_shell = t4p_shell.parse_from_index().to_browser()
```

```
* Parsing heu-met-fast-001-shell_model.res
* Successful scan in 0.022095 s
* Successful parsing in 0.013342 s
```

[17]:
```
sb_shell = t4b_shell.filter_by(response_function='KEFFS')
ldsets_shell = []
for keff in sb_shell.content:
    ldsets_shell.append(keff['results']['keff'])
    ldsets_shell[-1].name=keff['keff_estimator']
```

[18]:
```
dset_shell = Dataset(value=np.array([k.value for k in ldsets_shell]),
                     error=np.array([k.error for k in ldsets_shell]),
                     bins=OrderedDict([('estimator', np.array([k.name for k in ldsets_
↪shell]))]),
                     name='Shell model', what='keff')
```

On importe les tests et la possibilité d'en faire des représentations

[19]:
```
from valjean.gavroche.stat_tests.student import TestStudent
from valjean.javert.representation import FullRepresenter
from valjean.javert.rst import RstFormatter
from valjean.javert.mpl import MplPlot
from valjean.javert.verbosity import Verbosity

frepr = FullRepresenter()
rstformat = RstFormatter()
```

La comparaison sera faite grâce à un `TestStudent` pour prendre en compte les erreurs statistiques de Tripoli-4.

[20]:
```
tstud_res = TestStudent(dset_simple, dset_shell, name='TestStudent',
                        description='Test le TestStudent sur les keff').evaluate()
print(bool(tstud_res))
```

```
True
```

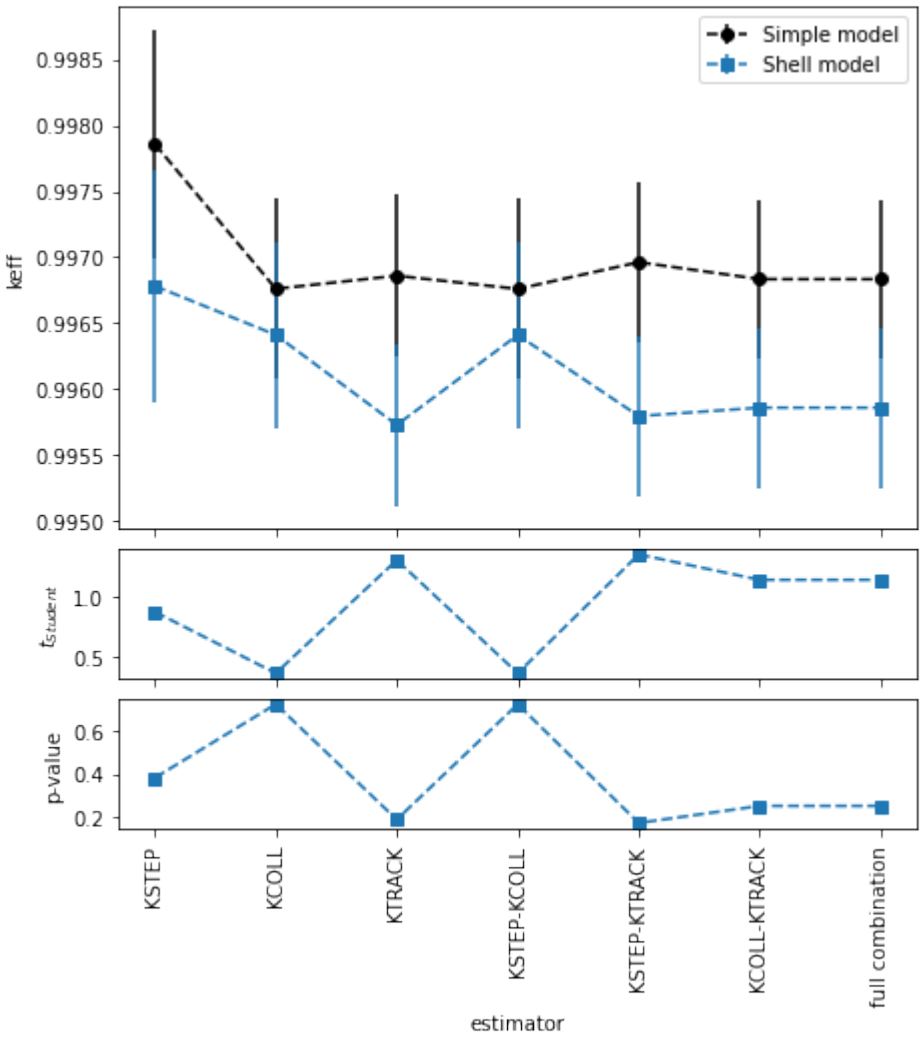
[21]:
```
stud_temp = frepr(tstud_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une␣
↪liste de templates
stud_rst = rstformat.template(stud_temp[1])
print(stud_rst)
```

```
.. role:: hl

.. table::
    :widths: auto

    ================  ===============  ===============  ============== ␣
↪==============  ===========  ========
       estimator      v(Simple model)  σ(Simple model)  v(Shell model)  σ(Shell␣
↪model)       t       Student?
    ================  ===============  ===============  ============== ␣
↪==============  ===========  ========
                KSTEP         0.997857      0.000859893        0.996782     0.
↪000882087     0.872579      True
                KCOLL         0.996761      0.000680173        0.996412     0.
↪000701962     0.357056      True
               KTRACK          0.99686      0.000611041         0.99573      0.
↪00061235      1.30649      True
         KSTEP-KCOLL          0.99676      0.000680173         0.99641      0.
↪00070196     0.357363      True
        KSTEP-KTRACK         0.996962      0.000607047        0.995797     0.
↪000610815      1.35352      True
        KCOLL-KTRACK         0.996834      0.000600633         0.99586      0.
↪00060659      1.14052      True
    full combination         0.996834      0.000600633         0.99586     0.
↪000606584      1.14053      True
    ================  ===============  ===============  ============== ␣
↪==============  ===========  ========
```

[22]:
```
mpl = MplPlot(stud_temp[0]).draw()
```



3.1.5 Sphères de Livermore

Description rapide de l'expérience

Les expériences dites "Sphères de Livermore" ont été réalisées fin des années 60 - début des années 70 au Lawrence Livermore Laboratory (États-Unis).

Une sphère creuse est placée au centre d'un bunker. En son centre on a une source de neutrons à 14 MeV (faisceau de $^2H^+$ sur une cible d'tritium, réaction $^3H(d,n)^4He$). Des détecteurs sont positionnés autour de la sphère après collimation.

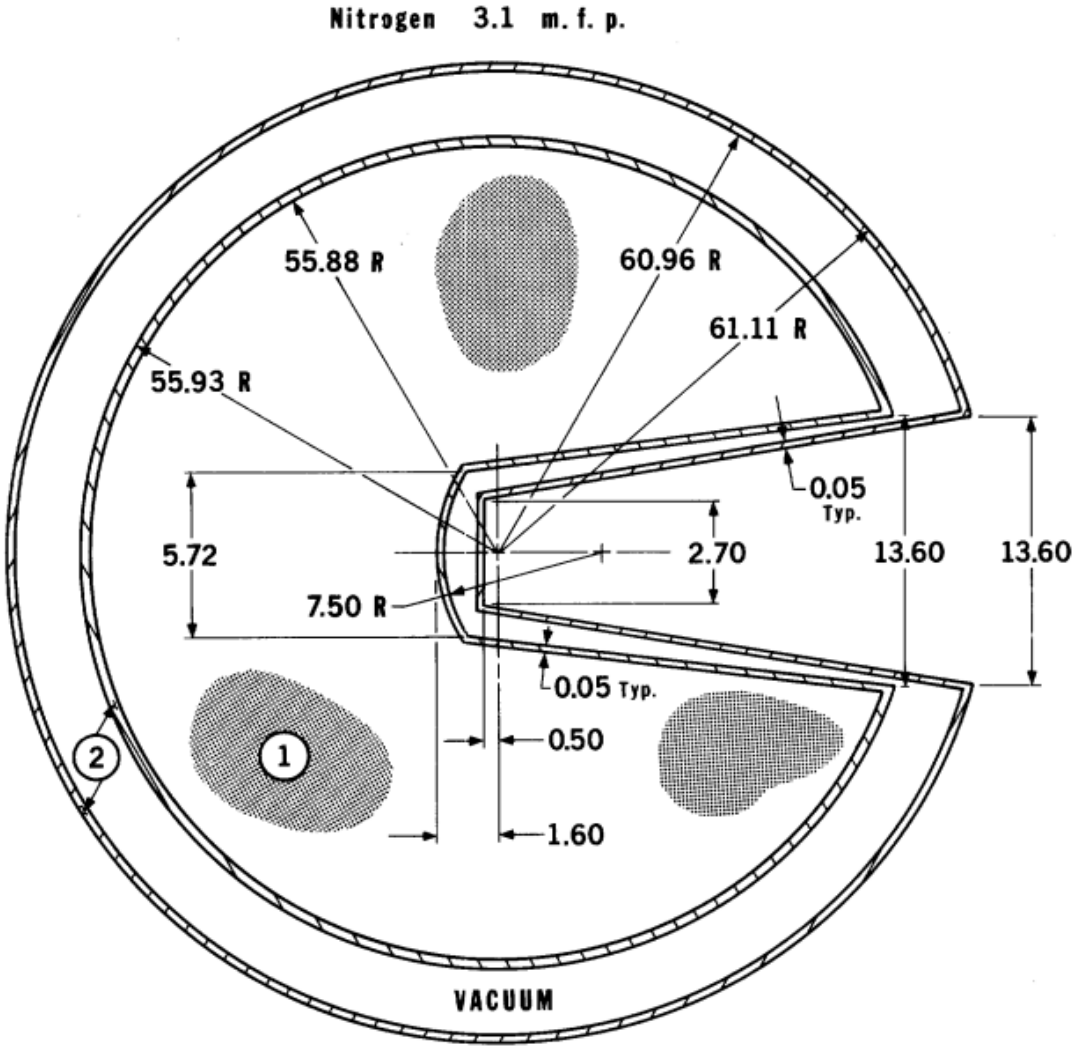
On observe le spectre en temps des neutrons qui arrivent dans les détecteurs (réponse type `REACTION` dans Tripoli-4).

Pour une sphère donnée on exécute deux fois la simulation : - avec la sphère étudiée (matériau = fer, béryllium, azote, eau, etc) - avec la même sphère dont le matériau étudié a été remplacé par de l'**air**

Cette seconde sphère nous permet de normaliser les résultats et de pouvoir nous comparer

aux données expérimentales notamment grâce à un graphique. Il y a donc deux sorties Tripoli-4 à lire et des opérations à faire sur les spectres.

Dans le cas présent la sphère considérée est celle d'azote liquide, d'une épaisseur de 3.1 mfp (libre parcours moyen), avec un spectre à 30°.



Parsing des résultats Tripoli-4 et récupération du spectre

Parser les résultats

[1]:
```
from valjean.eponine.tripoli4.parse import Parser
jdd_sphere = 'prob103_nitrogen3.1_fine_timeShifted_sphere_PARA.d.res'
jdd_air = 'prob103_nitrogen3.1_fine_timeShifted_air_PARA.d.res'
```

Le module permettant de lire, parser et récupérer les résultats de Tripoli-4 sous un format plus facilement exploitable est `Parser`.

On ne regardera ici que le résultat du dernier batch. Le parsing est effectivement fait par la méthode `parse_from_index`. L'index par défaut est `-1`, ce qui correspond au dernier batch.

Pour manipuler plus aisément les réponses de Tripoli-4 et en particulier les sélectionner on utilise un objet `Browser`.

[2]:
```
sphere_b = Parser(jdd_sphere).parse_from_index(name='sphere').to_browser()
air_b = Parser(jdd_air).parse_from_index(name='air').to_browser()
```

```
* Parsing prob103_nitrogen3.1_fine_timeShifted_sphere_PARA.d.res
* Successful scan in 0.263119 s
* Successful parsing in 1.817557 s
* Parsing prob103_nitrogen3.1_fine_timeShifted_air_PARA.d.res
* Successful scan in 0.230765 s
* Successful parsing in 1.759230 s
```

Sélection du résultat

À chaque réponse dans ce jeu de données a été associé un `SCORE NAME` unique (et explicite). C'est le moyen le plus aisé de récupérer les réponses nécessaires.

Dans le cas de la sphère (ici d'azote liquide), on récupère `score_name='neutron_response_30deg'`, soit le spectre neutron à 30 degrés (des spectres photons sont aussi disponibles). Il s'agira du numérateur (variable `nsphere`). Dans le cas de l'air, seule l'intégrale du spectre est nécessaire pour la normalisation, donc au dénominateur (variable `dair`). La sélection de la réponse se fait sur `score_name='neutron_response_integral_30deg'`.

[3]:
```
nsphere = sphere_b.select_by(score_name='neutron_response_30deg')
dair = air_b.select_by(score_name='neutron_response_integral_30deg')
```

Transformation des données en `Dataset`

Pour pouvoir manipuler les données plus aisément mais aussi pour faciliter leur manipulation, elles sont transformées en `Dataset`.

Le but de `Dataset` est d'être commun à tous les types de données (Tripoli-4, données expérimentales, PATMOS, MCNP, etc). Chaque dataset contient au moins 2 membres : `value` et `error`. Il s'agit de l'incertitude absolue (et non pas relative comme dans les résultats standard Tripoli-4). Trois autres membres sont optionnels : `bins`, `name` et `what`.

Les données sont stockées sous forme de `Dataset` accessible dans le `Browser` sous la clef `'results'`.

Pour info :

[4]:
```
test_ds = nsphere['results']['score']
print(type(test_ds))
print(test_ds)
```

```
<class 'valjean.eponine.dataset.Dataset'>
shape: (1, 1, 1, 1, 138, 1, 1), dim: 7, type: <class 'numpy.ndarray'>, bins: ['u: []',
↪ 'v: []', 'w: []', 'e: [1.e-11 2.e+01]', 't: [0.00e+00 2.38e-07 2.40e-07 2.42e-07 2.
↪44e-07 2.46e-07 2.48e-07 2.50e-07 2.52e-07 2.54e-07 2.56e-07 2.58e-07 2.60e-07 2.
↪62e-07 2.64e-07 2.66e-07 2.68e-07 2.70e-07 2.72e-07 2.74e-07 2.76e-07 2.78e-07 2.
```

(continues on next page)

(continued from previous page)

```
↪80e-07 2.82e-07 2.84e-07 2.86e-07 2.88e-07 2.90e-07 2.92e-07 2.94e-07 2.96e-07 2.
↪98e-07 3.00e-07 3.02e-07 3.04e-07 3.06e-07 3.08e-07 3.10e-07 3.12e-07 3.14e-07 3.
↪16e-07 3.18e-07 3.20e-07 3.22e-07 3.24e-07 3.26e-07 3.28e-07 3.30e-07 3.32e-07 3.
↪34e-07 3.36e-07 3.38e-07 3.40e-07 3.42e-07 3.44e-07 3.46e-07 3.48e-07 3.50e-07 3.
↪52e-07 3.54e-07 3.56e-07 3.58e-07 3.60e-07 3.62e-07 3.64e-07 3.66e-07 3.68e-07 3.
↪70e-07 3.72e-07 3.74e-07 3.76e-07 3.78e-07 3.80e-07 3.82e-07 3.84e-07 3.86e-07 3.
↪88e-07 3.90e-07 3.92e-07 3.94e-07 3.96e-07 3.98e-07 4.00e-07 4.02e-07 4.04e-07 4.
↪06e-07 4.08e-07 4.10e-07 4.12e-07 4.14e-07 4.16e-07 4.18e-07 4.20e-07 4.22e-07 4.
↪24e-07 4.26e-07 4.28e-07 4.30e-07 4.32e-07 4.34e-07 4.36e-07 4.38e-07 4.40e-07 4.
↪42e-07 4.44e-07 4.46e-07 4.48e-07 4.50e-07 4.52e-07 4.54e-07 4.56e-07 4.58e-07 4.
↪60e-07 4.62e-07 4.64e-07 4.66e-07 4.68e-07 4.70e-07 4.72e-07 4.74e-07 4.76e-07 4.
↪78e-07 4.80e-07 4.82e-07 4.84e-07 4.86e-07 4.88e-07 4.90e-07 4.92e-07 4.94e-07 4.
↪96e-07 4.98e-07 5.00e-07 5.02e-07 5.04e-07 5.06e-07 5.08e-07 5.10e-07 1.00e+01]',
↪'mu: []', 'phi: []'], name: sphere, what: reaction
```

[5]:
```
print(test_ds.squeeze().shape)
```

```
(138,)
```

Pour les manipuler plus simplement, les `Dataset` sont *squeezés* : nous récupérons des spectres en temps, les 6 autres dimensions sont donc à 1 et non utiles ici (et triviales).

[6]:
```
nspheres = nsphere['results']['score'].squeeze()
nspheres.name="azote (num)"
```

[7]:
```
dairds = dair['results']['score'].squeeze()
dairds.name='air (denom)'
print(dairds)
```

```
shape: (3,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['t: [0.00e+00 2.38e-07 5.
↪10e-07 1.00e+01]'], name: air (denom), what: reaction
```

Normalisation du spectre

Le résultat correspondant à l'intégrale du spectre n'est en réalité ici pas un score générique, mais un réel spectre. Cette différence est due aux intervalles extrêmes : - entre t=0 et le début des résultats expérientaux, à t=138 ns pour le premier - entre t=410 ns et t=10 s pour le dernier, soit entre la fin des résultats expérimentaux et la valeur maximale du temps dans Tripoli-4

Pour être plus juste, notamment au niveau du calcul des incertitudes associées, le choix a été fait de faire un spectre de 3 intervalles où seul celui du milieu nous intéresse dans le cas présent.

L'incertitude sur la norme est négligée par la suite (elle est dominée par celle sur chaque intervalle).

[8]:
```
import numpy as np
norm = dairds.value[1]
print("type(norm) = {0}, shape(norm) = {1}".format(type(norm), norm.shape))
```

```
type(norm) = <class 'numpy.float64'>, shape(norm) = ()
```

En réalité deux normalisations du spectre sont à effectuer, celle par l'intégrale de l'air et celle par la largeur des bins, appelée ici `TIME_BIN_WIDTH`, valant 2 ns.

[9]:
```
TIME_BIN_WIDTH = 2
t4ds = nsphereds / norm / TIME_BIN_WIDTH
```

Remarque : il n'aurait pas été possible d'utiliser un `Dataset` issu de `dairds` pour la normalisation.

[10]:
```
print(dairds)
test_dairds = dairds.copy()[1:-1]
print(test_dairds)
```

```
shape: (3,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['t: [0.00e+00 2.38e-07 5.
↪10e-07 1.00e+01]'], name: air (denom), what: reaction
shape: (1,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['t: [2.38e-07 5.10e-07]'],␣
↪name: air (denom), what: reaction
```

[11]:
```
test_ds = nsphereds / test_dairds / TIME_BIN_WIDTH
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[11], line 1
----> 1 test_ds = nsphereds / test_dairds / TIME_BIN_WIDTH

File ~/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/python3.8/site-
↪packages/valjean/eponine/dataset.py:952, in Dataset.__truediv__(self, other)
    948 if not isinstance(other, Dataset):
    949     return Dataset(
    950         self.value / other, self.error / other,
    951         bins=self.bins, name=self.name, what=self.what)
--> 952 self._check_datasets_consistency(other, "divide")
    953 value = self.value / other.value
    954 # RuntimeWarning can be ignored thanks to the commented line.
    955 # 'log' can be used instead of 'ignore' but did not work.
    956 # with np.errstate(divide='divide', invalid='ignore'):

File ~/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/python3.8/site-
↪packages/valjean/eponine/dataset.py:893, in Dataset._check_datasets_
↪consistency(self, other, operation)
    891 def _check_datasets_consistency(self, other, operation=""):
    892     if other.shape != self.shape:
--> 893         raise ValueError(f"Datasets to {operation} do not have same shape")
    894     if other.bins != OrderedDict():
    895         if any((s != o) for s, o in zip(self.bins, other.bins)):

ValueError: Datasets to divide do not have same shape
```

Ces deux datasets n'ont bien pas les shapes or les calculs sur les `Dataset` ne peuvent être effectués que s'ils ont la même *shape* et, dans le cas où des bins ont été fournis, si leurs bins sont équivalents.

[12]:
```
'shape(t4ds) = {0}, shape(test_dairds) = {1}'.format(t4ds.shape, test_dairds.shape)
```

[12]:
```
'shape(t4ds) = (138,), shape(test_dairds) = (1,)'
```

Réarrangement des intervalles

Les temps sont par défaut en *s* dans Tripoli-4 alors que les données que nous avons à disposition sont données par intervalles de temps en *ns*, on convertit donc les intervalles de Tripoli-4 en *ns*.

Par ailleurs, les intervalles en temps dans le jeu de données ont été décalés de 100 ns pour que la description de la source (gaussienne) soit correcte et complètement prise en compte dans le temps de la simulation (jeu de données tourné avec Tripoli-4, version 10.2, ce bug a été corrigé pour la version 11, mais c'est une jolie illustration des calculs possibles sur les `Dataset`).

[13]:
```
print('Bins en s et avant décalage:\n',t4ds.bins['t'] )
t4ds.bins['t'] = t4ds.bins['t'] * 1e9 - 100
print('Bins en ns et après décalage:\n', t4ds.bins['t'])
```

```
Bins en s et avant décalage:
 [0.00e+00 2.38e-07 2.40e-07 2.42e-07 2.44e-07 2.46e-07 2.48e-07 2.50e-07
 2.52e-07 2.54e-07 2.56e-07 2.58e-07 2.60e-07 2.62e-07 2.64e-07 2.66e-07
 2.68e-07 2.70e-07 2.72e-07 2.74e-07 2.76e-07 2.78e-07 2.80e-07 2.82e-07
 2.84e-07 2.86e-07 2.88e-07 2.90e-07 2.92e-07 2.94e-07 2.96e-07 2.98e-07
 3.00e-07 3.02e-07 3.04e-07 3.06e-07 3.08e-07 3.10e-07 3.12e-07 3.14e-07
 3.16e-07 3.18e-07 3.20e-07 3.22e-07 3.24e-07 3.26e-07 3.28e-07 3.30e-07
 3.32e-07 3.34e-07 3.36e-07 3.38e-07 3.40e-07 3.42e-07 3.44e-07 3.46e-07
 3.48e-07 3.50e-07 3.52e-07 3.54e-07 3.56e-07 3.58e-07 3.60e-07 3.62e-07
 3.64e-07 3.66e-07 3.68e-07 3.70e-07 3.72e-07 3.74e-07 3.76e-07 3.78e-07
 3.80e-07 3.82e-07 3.84e-07 3.86e-07 3.88e-07 3.90e-07 3.92e-07 3.94e-07
 3.96e-07 3.98e-07 4.00e-07 4.02e-07 4.04e-07 4.06e-07 4.08e-07 4.10e-07
 4.12e-07 4.14e-07 4.16e-07 4.18e-07 4.20e-07 4.22e-07 4.24e-07 4.26e-07
 4.28e-07 4.30e-07 4.32e-07 4.34e-07 4.36e-07 4.38e-07 4.40e-07 4.42e-07
 4.44e-07 4.46e-07 4.48e-07 4.50e-07 4.52e-07 4.54e-07 4.56e-07 4.58e-07
 4.60e-07 4.62e-07 4.64e-07 4.66e-07 4.68e-07 4.70e-07 4.72e-07 4.74e-07
 4.76e-07 4.78e-07 4.80e-07 4.82e-07 4.84e-07 4.86e-07 4.88e-07 4.90e-07
 4.92e-07 4.94e-07 4.96e-07 4.98e-07 5.00e-07 5.02e-07 5.04e-07 5.06e-07
 5.08e-07 5.10e-07 1.00e+01]
Bins en ns et après décalage:
 [-1.0000000e+02  1.3800000e+02  1.4000000e+02  1.4200000e+02
  1.4400000e+02  1.4600000e+02  1.4800000e+02  1.5000000e+02
  1.5200000e+02  1.5400000e+02  1.5600000e+02  1.5800000e+02
  1.6000000e+02  1.6200000e+02  1.6400000e+02  1.6600000e+02
  1.6800000e+02  1.7000000e+02  1.7200000e+02  1.7400000e+02
  1.7600000e+02  1.7800000e+02  1.8000000e+02  1.8200000e+02
  1.8400000e+02  1.8600000e+02  1.8800000e+02  1.9000000e+02
  1.9200000e+02  1.9400000e+02  1.9600000e+02  1.9800000e+02
  2.0000000e+02  2.0200000e+02  2.0400000e+02  2.0600000e+02
  2.0800000e+02  2.1000000e+02  2.1200000e+02  2.1400000e+02
  2.1600000e+02  2.1800000e+02  2.2000000e+02  2.2200000e+02
  2.2400000e+02  2.2600000e+02  2.2800000e+02  2.3000000e+02
  2.3200000e+02  2.3400000e+02  2.3600000e+02  2.3800000e+02
  2.4000000e+02  2.4200000e+02  2.4400000e+02  2.4600000e+02
  2.4800000e+02  2.5000000e+02  2.5200000e+02  2.5400000e+02
  2.5600000e+02  2.5800000e+02  2.6000000e+02  2.6200000e+02
  2.6400000e+02  2.6600000e+02  2.6800000e+02  2.7000000e+02
  2.7200000e+02  2.7400000e+02  2.7600000e+02  2.7800000e+02
  2.8000000e+02  2.8200000e+02  2.8400000e+02  2.8600000e+02
  2.8800000e+02  2.9000000e+02  2.9200000e+02  2.9400000e+02
  2.9600000e+02  2.9800000e+02  3.0000000e+02  3.0200000e+02
  3.0400000e+02  3.0600000e+02  3.0800000e+02  3.1000000e+02
```

(continues on next page)

(continued from previous page)

```
  3.1200000e+02  3.1400000e+02  3.1600000e+02  3.1800000e+02
  3.2000000e+02  3.2200000e+02  3.2400000e+02  3.2600000e+02
  3.2800000e+02  3.3000000e+02  3.3200000e+02  3.3400000e+02
  3.3600000e+02  3.3800000e+02  3.4000000e+02  3.4200000e+02
  3.4400000e+02  3.4600000e+02  3.4800000e+02  3.5000000e+02
  3.5200000e+02  3.5400000e+02  3.5600000e+02  3.5800000e+02
  3.6000000e+02  3.6200000e+02  3.6400000e+02  3.6600000e+02
  3.6800000e+02  3.7000000e+02  3.7200000e+02  3.7400000e+02
  3.7600000e+02  3.7800000e+02  3.8000000e+02  3.8200000e+02
  3.8400000e+02  3.8600000e+02  3.8800000e+02  3.9000000e+02
  3.9200000e+02  3.9400000e+02  3.9600000e+02  3.9800000e+02
  4.0000000e+02  4.0200000e+02  4.0400000e+02  4.0600000e+02
  4.0800000e+02  4.1000000e+02  9.9999999e+09]
```

De plus, le dataset actuel, `t4ds`, contient toujours les bins extrêmes, il faut donc les enlever.

Le *slicing* a été codé dans `Dataset` : il est effectué en même temps sur les valeurs, sur les erreurs et sur les intervalles.

[14]:
```
print('bins en temps (avant slicing):')
print(t4ds.bins['t'])
print("shape(t4ds) = {0}, bins(nom = {1}, shape = {2})"
      .format(t4ds.shape, list(t4ds.bins.keys()), t4ds.bins['t'].shape))
t4ds = t4ds[1:-1]
print('bins en temps (après slicing):')
print(t4ds.bins['t'])
print("shape(t4ds) = {0}, bins(nom = {1}, shape = {2})"
      .format(t4ds.shape, list(t4ds.bins.keys()), t4ds.bins['t'].shape))
```

```
bins en temps (avant slicing):
[-1.0000000e+02  1.3800000e+02  1.4000000e+02  1.4200000e+02
  1.4400000e+02  1.4600000e+02  1.4800000e+02  1.5000000e+02
  1.5200000e+02  1.5400000e+02  1.5600000e+02  1.5800000e+02
  1.6000000e+02  1.6200000e+02  1.6400000e+02  1.6600000e+02
  1.6800000e+02  1.7000000e+02  1.7200000e+02  1.7400000e+02
  1.7600000e+02  1.7800000e+02  1.8000000e+02  1.8200000e+02
  1.8400000e+02  1.8600000e+02  1.8800000e+02  1.9000000e+02
  1.9200000e+02  1.9400000e+02  1.9600000e+02  1.9800000e+02
  2.0000000e+02  2.0200000e+02  2.0400000e+02  2.0600000e+02
  2.0800000e+02  2.1000000e+02  2.1200000e+02  2.1400000e+02
  2.1600000e+02  2.1800000e+02  2.2000000e+02  2.2200000e+02
  2.2400000e+02  2.2600000e+02  2.2800000e+02  2.3000000e+02
  2.3200000e+02  2.3400000e+02  2.3600000e+02  2.3800000e+02
  2.4000000e+02  2.4200000e+02  2.4400000e+02  2.4600000e+02
  2.4800000e+02  2.5000000e+02  2.5200000e+02  2.5400000e+02
  2.5600000e+02  2.5800000e+02  2.6000000e+02  2.6200000e+02
  2.6400000e+02  2.6600000e+02  2.6800000e+02  2.7000000e+02
  2.7200000e+02  2.7400000e+02  2.7600000e+02  2.7800000e+02
  2.8000000e+02  2.8200000e+02  2.8400000e+02  2.8600000e+02
  2.8800000e+02  2.9000000e+02  2.9200000e+02  2.9400000e+02
  2.9600000e+02  2.9800000e+02  3.0000000e+02  3.0200000e+02
  3.0400000e+02  3.0600000e+02  3.0800000e+02  3.1000000e+02
  3.1200000e+02  3.1400000e+02  3.1600000e+02  3.1800000e+02
  3.2000000e+02  3.2200000e+02  3.2400000e+02  3.2600000e+02
  3.2800000e+02  3.3000000e+02  3.3200000e+02  3.3400000e+02
  3.3600000e+02  3.3800000e+02  3.4000000e+02  3.4200000e+02
  3.4400000e+02  3.4600000e+02  3.4800000e+02  3.5000000e+02
```

(continues on next page)

(continued from previous page)

```
  3.5200000e+02  3.5400000e+02  3.5600000e+02  3.5800000e+02
  3.6000000e+02  3.6200000e+02  3.6400000e+02  3.6600000e+02
  3.6800000e+02  3.7000000e+02  3.7200000e+02  3.7400000e+02
  3.7600000e+02  3.7800000e+02  3.8000000e+02  3.8200000e+02
  3.8400000e+02  3.8600000e+02  3.8800000e+02  3.9000000e+02
  3.9200000e+02  3.9400000e+02  3.9600000e+02  3.9800000e+02
  4.0000000e+02  4.0200000e+02  4.0400000e+02  4.0600000e+02
  4.0800000e+02  4.1000000e+02  9.9999999e+09]
shape(t4ds) = (138,), bins(nom = ['t'], shape = (139,))
bins en temps (après slicing):
[138. 140. 142. 144. 146. 148. 150. 152. 154. 156. 158. 160. 162. 164.
 166. 168. 170. 172. 174. 176. 178. 180. 182. 184. 186. 188. 190. 192.
 194. 196. 198. 200. 202. 204. 206. 208. 210. 212. 214. 216. 218. 220.
 222. 224. 226. 228. 230. 232. 234. 236. 238. 240. 242. 244. 246. 248.
 250. 252. 254. 256. 258. 260. 262. 264. 266. 268. 270. 272. 274. 276.
 278. 280. 282. 284. 286. 288. 290. 292. 294. 296. 298. 300. 302. 304.
 306. 308. 310. 312. 314. 316. 318. 320. 322. 324. 326. 328. 330. 332.
 334. 336. 338. 340. 342. 344. 346. 348. 350. 352. 354. 356. 358. 360.
 362. 364. 366. 368. 370. 372. 374. 376. 378. 380. 382. 384. 386. 388.
 390. 392. 394. 396. 398. 400. 402. 404. 406. 408. 410.]
shape(t4ds) = (136,), bins(nom = ['t'], shape = (137,))
```

Enfin, les données issues de la simulation sont données par intervalle (avec les extémités des intervalles) alors que les données expérimentales sont données au centre de l'intervalle. Ces dernières sont également des `int` et non des `float` comme les temps issus de Tripoli-4. L'étape finale est donc de supprimer la première (ou dernière) valeur dans les *bins* et de toutes les décaler d'1 ns (largeur d'intervalle toujours de 2 ns) et de les transformer en `int`.

[15]:
```
t4ds.bins['t'] = np.rint(t4ds.bins['t'][1:] - 1)
print(t4ds.bins['t'])
```

```
[139. 141. 143. 145. 147. 149. 151. 153. 155. 157. 159. 161. 163. 165.
 167. 169. 171. 173. 175. 177. 179. 181. 183. 185. 187. 189. 191. 193.
 195. 197. 199. 201. 203. 205. 207. 209. 211. 213. 215. 217. 219. 221.
 223. 225. 227. 229. 231. 233. 235. 237. 239. 241. 243. 245. 247. 249.
 251. 253. 255. 257. 259. 261. 263. 265. 267. 269. 271. 273. 275. 277.
 279. 281. 283. 285. 287. 289. 291. 293. 295. 297. 299. 301. 303. 305.
 307. 309. 311. 313. 315. 317. 319. 321. 323. 325. 327. 329. 331. 333.
 335. 337. 339. 341. 343. 345. 347. 349. 351. 353. 355. 357. 359. 361.
 363. 365. 367. 369. 371. 373. 375. 377. 379. 381. 383. 385. 387. 389.
 391. 393. 395. 397. 399. 401. 403. 405. 407. 409.]
```

Le `Dataset` est maintenant prêt pour la comparaison aux données. On met simplement à jour son nom et la variable qu'il représente (ordonnée) pour le représenter explicitement.

[16]:
```
t4ds.name = 'T4'
t4ds.what = 'Neutron count rate'
```

Résultats expérimentaux

Les résultats expérimentaux sont fournis dans un fichier ASCII, sous forme de tableaux de valeurs. Il faut donc les parser et les transformer en `Dataset`.

Cette étape est actuellement faite dans une petite classe externe : LivermoreExps

[17]:
```
from livermore_exps import LivermoreExps

exps = LivermoreExps('s10a11.res.mesure')
```

```
s10a11.res.mesure
['BERYLLIUM', '0.8', '30', 765.2]
['CARBON', '0.5', '30', 766.0]
['CARBON', '0.5', '120', 975.2]
['CARBON', '2.9', '30', 766.0]
['CARBON', '2.9', '120', 975.2]
['NITROGEN', '3.1', '30', 765.2]
['OXYGENE', '0.7', '30', 754.0]
['MAGNESIUM', '0.7', '30', 765.2]
['MAGNESIUM', '0.7', '120', 977.2]
['ALUMINIUM', '0.9', '30', 765.2]
['ALUMINIUM', '0.9', '120', 977.2]
['IRON', '0.9', '30', 766.0]
['IRON', '0.9', '120', 975.2]
['IRON', '4.8', '30', 766.0]
['IRON', '4.8', '120', 975.2]
['WATER', '1.1', '30', 754.0]
['H_WATER', '1.2', '30', 765.0]
['CONCRETE', '2.0', '120', 975.4]
ALL KEYS:
 [('BERYLLIUM', '0.8', '30'), ('CARBON', '0.5', '30'), ('CARBON', '0.5', '120'), (
↪'CARBON', '2.9', '30'), ('CARBON', '2.9', '120'), ('NITROGEN', '3.1', '30'), (
↪'OXYGENE', '0.7', '30'), ('MAGNESIUM', '0.7', '30'), ('MAGNESIUM', '0.7', '120'), (
↪'ALUMINIUM', '0.9', '30'), ('ALUMINIUM', '0.9', '120'), ('IRON', '0.9', '30'), (
↪'IRON', '0.9', '120'), ('IRON', '4.8', '30'), ('IRON', '4.8', '120'), ('WATER', '1.1
↪', '30'), ('H_WATER', '1.2', '30'), ('CONCRETE', '2.0', '120')]
```

Toutes les données expérimentales sont ainsi disponibles, il suffit de les charger une seule fois pour toutes les analyses des sphères de Livermore. Ici on ne considèrera qu'un seul cas : `('NITROGEN', '3.1', '30')`.

[18]:
```
exp_key = ('NITROGEN', '3.1', '30')
exp_data = exps.res[exp_key]
type(exp_data)
```

[18]:
```
valjean.eponine.dataset.Dataset
```

[19]:
```
print(exp_data)
```

```
shape: (136,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['t: [139. 141. 143. 145.␣
↪147. 149. 151. 153. 155. 157. 159. 161. 163. 165. 167. 169. 171. 173. 175. 177. 179.
↪ 181. 183. 185. 187. 189. 191. 193. 195. 197. 199. 201. 203. 205. 207. 209. 211.␣
↪213. 215. 217. 219. 221. 223. 225. 227. 229. 231. 233. 235. 237. 239. 241. 243. 245.
↪ 247. 249. 251. 253. 255. 257. 259. 261. 263. 265. 267. 269. 271. 273. 275. 277.␣
↪279. 281. 283. 285. 287. 289. 291. 293. 295. 297. 299. 301. 303. 305. 307. 309. 311.
↪ 313. 315. 317. 319. 321. 323. 325. 327. 329. 331. 333. 335. 337. 339. 341. 343.␣
↪345. 347. 349. 351. 353. 355. 357. 359. 361. 363. 365. 367. 369. 371. 373. 375. 377.
```

(continues on next page)

(continued from previous page)

```
↪ 379. 381. 383. 385. 387. 389. 391. 393. 395. 397. 399. 401. 403. 405. 407. 409.]'],
↪ name: data, what:
```

Les objets dans le dictionnaire de résultat (`exps.res`) sont des `Dataset`.

Comparaison entre données et expérience

Pour les graphiques on utilise le rapport des spectres simulé et expérimental. La classe `Dataset` fournit les outils pour faire ce type de calculs.

[20]:
```
print(np.array_equal(t4ds.bins['t'], exp_data.bins['t']))
```

```
True
```

Malgré le fait que les bins nous apparaissaient complètement équivalents, ils ne l'étaient pas : cela vient probablement de la conversion de strings (depuis les fichiers ASCII) en float alors que les nombres n'étaient pas écrits de la même manière (`int` pour les données expérimentales, `float` en notation exponentielle à 6 chiffres après la virgule pour Tripoli-4 ayant subis quelques calculs en plus).

[21]:
```
ratio = t4ds / exp_data
```

Comparaisons numériques

Il est possible de comparer les deux `Dataset` numériquement grâce aux fonctions disponibles dans `gavroche.test.py` qui agissent sur les datasets.

[22]:
```
from valjean.gavroche.test import TestApproxEqual
test_equality = TestApproxEqual(t4ds, exp_data, name='light criteria on approx eq',␣
↪rtol=0.1, atol=1e-2)
print(bool(test_equality.evaluate()))
```

```
True
```

Lors de la VV ce test est davantage fait sur l'intégrale du spectre.

[23]:
```
integ = sphere_b.select_by(score_name='neutron_response_integral_30deg')
intnumds = integ['results']['score'].squeeze()
intnumds.name='integ'
integds = intnumds / norm / TIME_BIN_WIDTH
integds = integds[1:-1]
print(integds)
```

```
shape: (1,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['t: [2.38e-07 5.10e-07]'],␣
↪name: integ, what: reaction
```

Petite vérification de routine rapide :

[24]:
```
np.allclose(np.sum(t4ds.value), integds.value)
```

[24]:
```
True
```

On supprime les bins ici pour simplifier la comparaison (il s'agit d'une intégrale).

[25]:
```
from collections import OrderedDict
integds.bins = OrderedDict()
print(integds)
```

```
shape: (1,), dim: 1, type: <class 'numpy.ndarray'>, bins: [], name: integ, what:␣
↪reaction
```

Intégrale des données, en supposant les intervalles indépendants :

[26]:
```
from valjean.eponine.dataset import Dataset
```

[27]:
```
quad_err = np.sqrt(np.sum(exp_data.error ** 2))
integ_data = Dataset(np.sum(exp_data.value), quad_err)
print(integ_data)
```

```
value: 2.015474e-01, error: 4.239589e-04, bins: OrderedDict(), type: <class 'numpy.
↪float64'>,name: , what:
```

[28]:
```
equ_integ = TestApproxEqual(integds, integ_data, name='approx eq integrales', rtol=0.
↪03, atol=1e-4)
print(bool(equ_integ))
```

```
True
```

Graphiques par défaut dans valjean

La majorité des tests peut être représentée sous forme de graphique.

[29]:
```
from valjean.javert.representation import FullRepresenter
from valjean.javert.rst import RstFormatter
from valjean.javert.mpl import MplPlot
from valjean.javert.verbosity import Verbosity

frepr = FullRepresenter()
rstformat = RstFormatter()
```

[30]:
```
teq_res = test_equality.evaluate()
eqrepr = frepr(teq_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une liste de␣
↪templates
eqrst = rstformat.template(eqrepr[1])
print(eqrst)
mpl = MplPlot(eqrepr[0]).draw()
```

```
.. role:: hl

.. table::
    :widths: auto

    ===  ===========  ===========  =============
     t       T4          data      approx equal?
    ===  ===========  ===========  =============
    139  5.55457e-05   0.0009519          True
    141   0.00118528      0.0077          True
    143   0.00893213   0.0204795          True
    145    0.0218566    0.026057          True
    147    0.0216699   0.0173575          True
```

(continues on next page)

(continued from previous page)

```
    149    0.0132692    0.0095605          True
    151   0.00748997      0.00616          True
    153   0.00502181    0.0047192          True
    155   0.00388959    0.0040897          True
    157    0.0034185   0.00406495          True
    159   0.00317997   0.00369335          True
    161   0.00297349   0.00326535          True
    163   0.00279022    0.0029624          True
    165   0.00258342   0.00269255          True
    167   0.00236749   0.00232045          True
    169   0.00226739    0.0020996          True
    171   0.00206574   0.00201375          True
    173   0.00190346    0.0017443          True
    175   0.00170911    0.0016056          True
    177    0.0016319   0.00157055          True
    179   0.00158248    0.0015384          True
    181   0.00155479    0.0014737          True
    183   0.00153633    0.0014077          True
    185   0.00151627   0.00138115          True
    187    0.0015399     0.001371          True
    189   0.00157329   0.00138875          True
    191   0.00155078    0.0012998          True
    193   0.00148645   0.00130525          True
    195   0.00144237    0.0012063          True
    197   0.00134622     0.001105          True
    199   0.00131588    0.0010722          True
    201   0.00124256    0.0010684          True
    203   0.00117172   0.00099225          True
    205   0.00116655    0.0009726          True
    207   0.00109664   0.00095265          True
    209   0.00108667    0.0009802          True
    211   0.00108588   0.00101455          True
    213   0.00111711   0.00099005          True
    215   0.00109222   0.00102495          True
    217    0.0010924    0.0009409          True
    219   0.00106172   0.00085005          True
    221   0.00103876   0.00089455          True
    223   0.00106566    0.0009124          True
    225   0.00104802   0.00087615          True
    227   0.00102713   0.00088065          True
    229   0.00100579    0.0008775          True
    231   0.00103697     0.000875          True
    233   0.00103529   0.00091045          True
    235   0.00100917   0.00088475          True
    237   0.00100023   0.00084455          True
    239   0.00100228    0.0008346          True
    241  0.000983864   0.00083185          True
    243  0.000997541   0.00085315          True
    245   0.00104778   0.00088735          True
    247   0.00100842    0.0008887          True
    249   0.00106032   0.00093345          True
    251   0.00106348   0.00095775          True
    253   0.00107137    0.0009621          True
    255   0.00110801    0.0009755          True
    257   0.00102729   0.00087365          True
    259  0.000973585    0.0008456          True
```

(continues on next page)

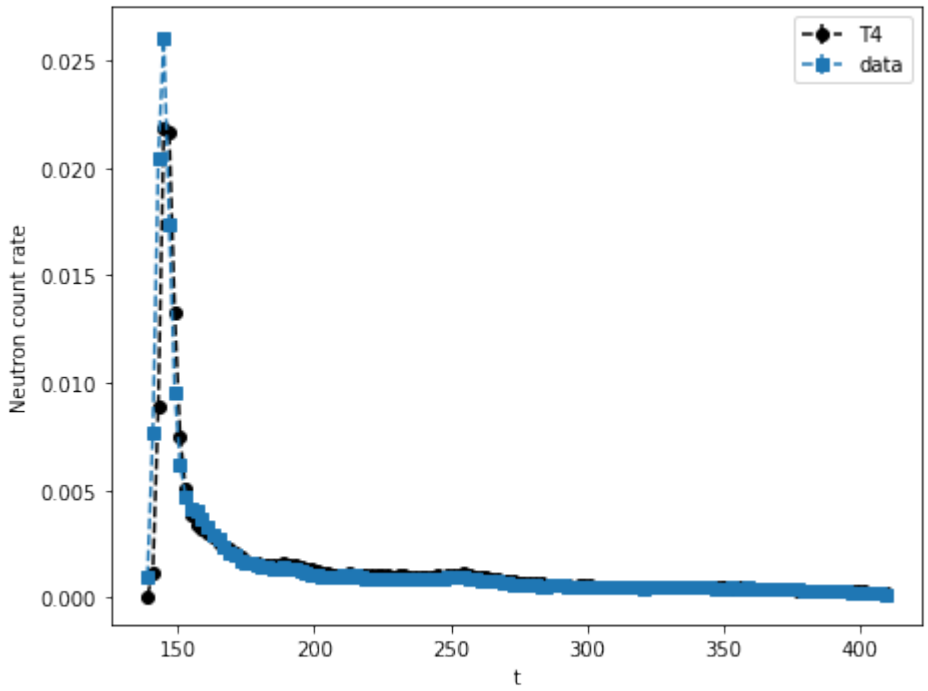
(continued from previous page)

```
261   0.00099714    0.0007888          True
263  0.000950634    0.0007969          True
265  0.000881015    0.0007551          True
267  0.000854355   0.00074015          True
269  0.000816657    0.0006776          True
271  0.000765727   0.00065885          True
273  0.000736658    0.0005968          True
275  0.000691288    0.0006269          True
277  0.000678993    0.0006212          True
279  0.000696424   0.00060335          True
281  0.000670687    0.0005857          True
283  0.000651523   0.00052105          True
285  0.000599761   0.00053375          True
287   0.00061133   0.00055205          True
289  0.000601374    0.0005533          True
291  0.000583103    0.0005488          True
293  0.000584056   0.00053585          True
295  0.000540916   0.00049291          True
297  0.000550412   0.00052315          True
299  0.000552923  0.000479485          True
301  0.000560512   0.00052695          True
303  0.000531155  0.000483245          True
305  0.000518599  0.000484415          True
307  0.000492058  0.000483095          True
309  0.000505487  0.000488225          True
311  0.000509219  0.000488575          True
313  0.000501507  0.000487045          True
315  0.000488588  0.000465225          True
317  0.000483927   0.00050375          True
319   0.00049458   0.00048882          True
321  0.000482865   0.00044683          True
323  0.000477005  0.000452225          True
325  0.000480317  0.000467225          True
327  0.000494829   0.00047503          True
329  0.000491724   0.00047855          True
331  0.000507182   0.00049157          True
333  0.000498105  0.000487365          True
335  0.000502847   0.00049807          True
337  0.000505027   0.00045714          True
339  0.000507307  0.000486445          True
341  0.000489606    0.0005155          True
343  0.000473202  0.000481145          True
345  0.000475292  0.000489915          True
347  0.000480632   0.00044432          True
349  0.000465248  0.000444205          True
351  0.000475173  0.000448605          True
353  0.000437251   0.00045204          True
355  0.000471841   0.00042395          True
357  0.000451007  0.000441215          True
359   0.00043615   0.00046094          True
361  0.000425573  0.000448195          True
363  0.000430618  0.000444745          True
365  0.000418642   0.00043726          True
367  0.000410464  0.000417455          True
369  0.000390776   0.00038885          True
371   0.00037246   0.00037076          True
```

(continues on next page)

(continued from previous page)

```
 373  0.000381532   0.00036254          True
 375  0.000359002  0.000398215          True
 377  0.000346426  0.000357555          True
 379  0.000354585  0.000324985          True
 381  0.000344207  0.000342015          True
 383    0.0003436   0.00031319          True
 385  0.000327421  0.000326635          True
 387  0.000309169  0.000303065          True
 389  0.000333483  0.000270825          True
 391  0.000301403  0.000296465          True
 393  0.000304476   0.00028084          True
 395  0.000291929    0.0002719          True
 397  0.000284313  0.000263645          True
 399  0.000269536   0.00025017          True
 401  0.000272009   0.00022047          True
 403   0.00025244   0.00020573          True
 405  0.000237807  0.000211075          True
 407  0.000227296   0.00019417          True
 409   0.00022099   0.00016967          True
 ===  ===========  ===========  =============
```



Avec un test de Holm-Bonferroni puisqu'il s'agit d'un spectre :

[31]:
```
from valjean.gavroche.stat_tests.student import TestStudent
from valjean.gavroche.stat_tests.bonferroni import TestHolmBonferroni
```

[32]:
```
sphere_b.globals
studt = TestStudent(t4ds, exp_data, name='Student test', ndf=sphere_b.globals[
↪'edition_batch_number'])
```

(continues on next page)

(continued from previous page)

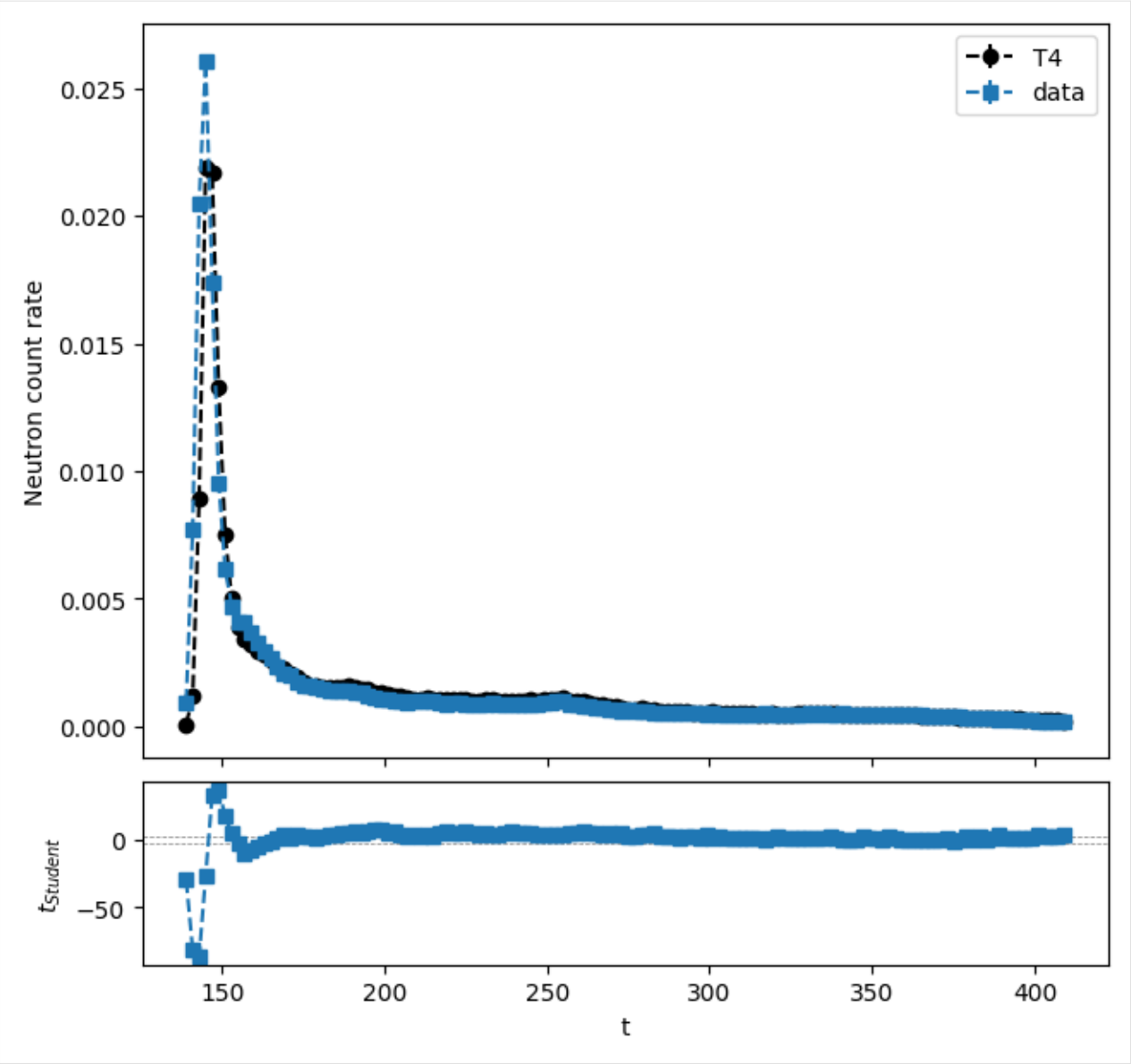
```
hb_res = TestHolmBonferroni(test=studt, name='Holm-Bonferroni test', description='').
↪evaluate()
```

[33]:
```
hbrepr = frepr(hb_res, verbosity=Verbosity.INTERMEDIATE)  # il s'agit d'une liste de␣
↪templates
hbrst = rstformat.template(hbrepr[1])
print(hbrst)
mpl = MplPlot(hbrepr[0]).draw()
```

```
.. role:: hl

.. table::
    :widths: auto

    ==========  ===  ==========  ============  ==========  ========== ␣
↪================
       test     ndf       α       min(p-value)    min(α)     N rejected  Holm-
↪Bonferroni?
    ==========  ===  ==========  ============  ==========  ========== ␣
↪================
    T4 vs data  136        0.005             0  3.67647e-05          39       :hl:
↪`False`
    ==========  ===  ==========  ============  ==========  ========== ␣
↪================
```



D'autres niveaux de verbosité sont disponibles. Il est également possible de changer un peu la représentation graphique des tests en utilisant une échelle logarithmique pour les données et la simulation par exemple.

[34]:
```
from valjean.javert import plot_repr as pltr
def log_post(templates, tres):
    pltr.post_treatment(templates, tres)
    for templ in templates:
        templ.subplots[0].attributes.logy = True
    return templates
```

[35]:
```
hbrepr = FullRepresenter(post=log_post)(hb_res, verbosity=Verbosity.FULL_DETAILS)  #␣
↪il s'agit d'une liste de templates
print(len(hbrepr[1:]))
hbrst = '\n'.join([str(rstformat.template(hbrepr[1])), str(rstformat.
↪template(hbrepr[2]))])
```

(continues on next page)

(continued from previous page)

```
print(hbrst)
mpl = MplPlot(hbrepr[0]).draw()
```

```
2
.. role:: hl

.. table::
    :widths: auto

    ==========  ===  ==========  ==========  ==========  ==========
       test     ndf       α       min(p-value)    min(α)     N rejected  Holm-Bonferroni?
    ==========  ===  ==========  ==========  ==========  ==========
    T4 vs data  136        0.005             0  3.67647e-05          39       :hl:`False`
    ==========  ===  ==========  ==========  ==========  ==========


.. role:: hl

.. table::
    :widths: auto

    ===  ==========  ==========  ==========  ==========  ==========  ==========
     t      v(T4)       σ(T4)       v(data)     σ(data)        t        Student?
    ===  ==========  ==========  ==========  ==========  ==========  ==========
    139  5.55457e-05  3.10611e-06   0.0009519  2.99445e-05    -29.7741  :hl:`False`
    141   0.00118528  1.42187e-05      0.0077   7.9415e-05    -80.7498  :hl:`False`
    143   0.00893213  4.06929e-05   0.0204795   0.00012866     -85.573  :hl:`False`
    145    0.0218566  6.54398e-05    0.026057     0.000145     -26.404  :hl:`False`
    147    0.0216699  6.09966e-05   0.0173575   0.00011853     32.3498  :hl:`False`
    149    0.0132692  5.06842e-05   0.0095605    8.831e-05     36.4235  :hl:`False`
    151   0.00748997  3.62418e-05     0.00616    7.122e-05     16.6432  :hl:`False`
    153   0.00502181  3.17534e-05   0.0047192   6.2585e-05     4.31188  :hl:`False`
    155   0.00388959  3.01783e-05   0.0040897   5.8415e-05    -3.04344  :hl:`False`
    157    0.0034185  2.60436e-05  0.00406495   5.8245e-05     -10.132  :hl:`False`
    159   0.00317997  2.54106e-05  0.00369335    5.563e-05    -8.39418  :hl:`False`
    161   0.00297349  2.39223e-05  0.00326535   5.2455e-05    -5.06248  :hl:`False`
    163   0.00279022   2.5883e-05   0.0029624   5.0095e-05     -3.0536  :hl:`False`
    165   0.00258342   2.3407e-05  0.00269255   4.7877e-05    -2.04779         True
    167   0.00236749  2.33064e-05  0.00232045   4.4653e-05    0.933804         True
    169   0.00226739  2.24576e-05   0.0020996   4.2624e-05     3.48261  :hl:`False`
    171   0.00206574  2.11621e-05  0.00201375   4.1809e-05     1.10938         True
    173   0.00190346  1.96915e-05   0.0017443  3.91435e-05     3.63225  :hl:`False`
    175   0.00170911  1.92469e-05   0.0016056   3.7693e-05     2.44584         True
    177    0.0016319  1.86048e-05  0.00157055  3.73155e-05     1.47144         True
    179   0.00158248  1.85253e-05   0.0015384  3.69715e-05     1.06596         True
    181   0.00155479  1.94426e-05   0.0014737   3.6263e-05     1.97088         True
    183   0.00153633  1.90821e-05   0.0014077   3.5526e-05     3.18979  :hl:`False`
    185   0.00151627  1.74553e-05  0.00138115   3.5225e-05       3.437  :hl:`False`
    187    0.0015399  2.00079e-05    0.001371   3.5109e-05     4.17969  :hl:`False`
    189   0.00157329  2.01629e-05  0.00138875  3.53115e-05     4.53821  :hl:`False`
    191   0.00155078  2.19893e-05   0.0012998   3.4287e-05     6.16179  :hl:`False`
    193   0.00148645  1.78986e-05  0.00130525  3.43505e-05       4.678  :hl:`False`
```

(continues on next page)

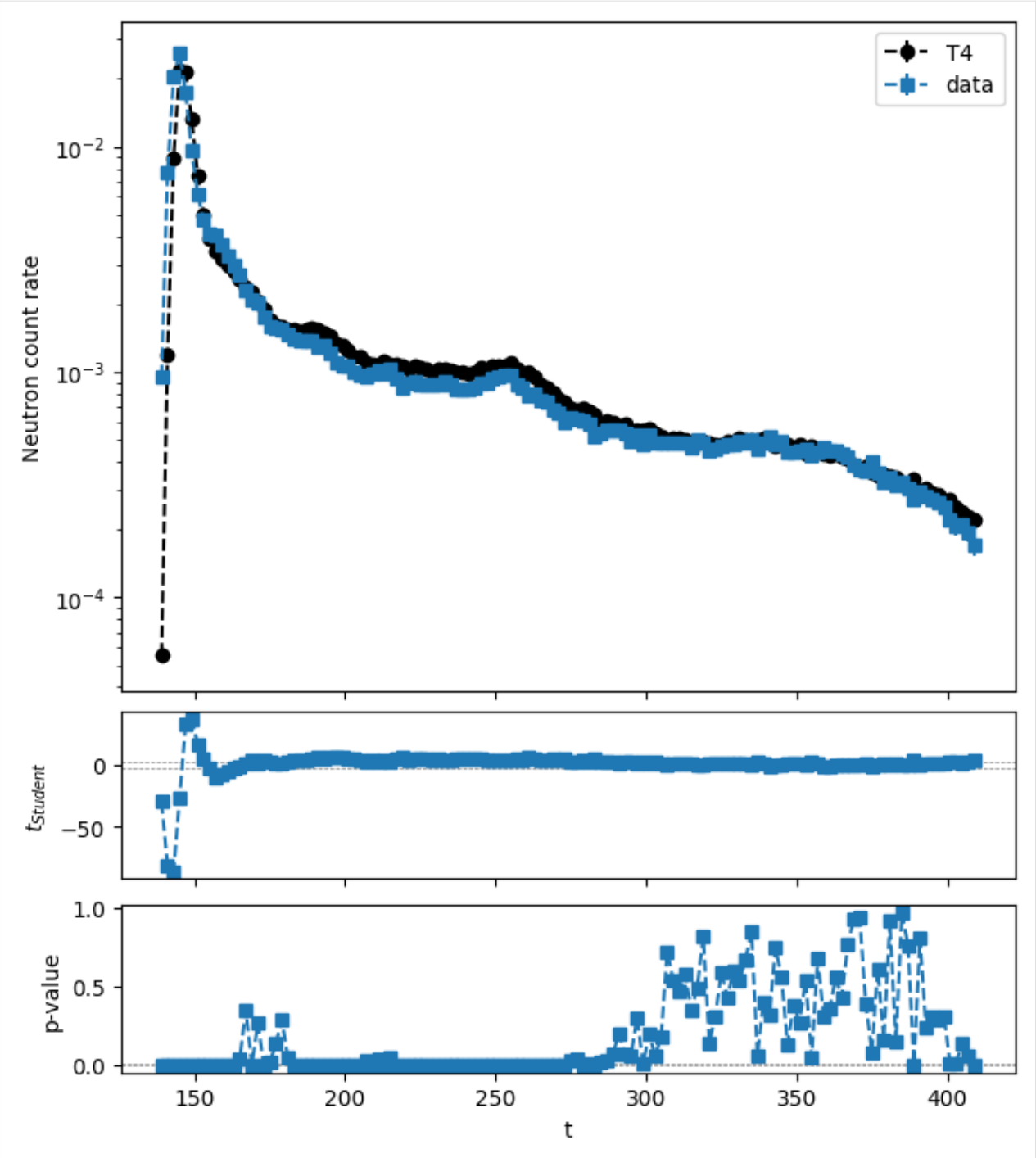
(continued from previous page)

195	0.00144237	1.88918e-05	0.0012063	3.31755e-05	6.18356	:hl:`False`
197	0.00134622	1.787e-05	0.001105	3.1928e-05	6.59263	:hl:`False`
199	0.00131588	1.94992e-05	0.0010722	3.1514e-05	6.57554	:hl:`False`
201	0.00124256	1.65324e-05	0.0010684	3.14655e-05	4.89969	:hl:`False`
203	0.00117172	1.65994e-05	0.00099225	3.04795e-05	5.17104	:hl:`False`
205	0.00116655	5.87326e-05	0.0009726	3.02205e-05	2.93635	:hl:`False`
207	0.00109664	5.846e-05	0.00095265	2.99545e-05	2.192	True
209	0.00108667	1.59775e-05	0.0009802	3.03205e-05	3.10663	:hl:`False`
211	0.00108588	1.5647e-05	0.00101455	3.07715e-05	2.06635	True
213	0.00111711	1.73304e-05	0.00099005	3.04505e-05	3.62654	:hl:`False`
215	0.00109222	1.61502e-05	0.00102495	3.09065e-05	1.92912	True
217	0.0010924	1.83855e-05	0.0009409	2.97965e-05	4.32694	:hl:`False`
219	0.00106172	1.68272e-05	0.00085005	2.8548e-05	6.38742	:hl:`False`
221	0.00103876	1.85236e-05	0.00089455	2.91665e-05	4.17379	:hl:`False`
223	0.00106566	1.63213e-05	0.0009124	2.9411e-05	4.55633	:hl:`False`
225	0.00104802	1.53696e-05	0.00087615	2.89125e-05	5.24889	:hl:`False`
227	0.00102713	1.59634e-05	0.00088065	2.8975e-05	4.42776	:hl:`False`
229	0.00100579	1.63727e-05	0.0008775	2.895e-05	3.85736	:hl:`False`
231	0.00103697	1.58147e-05	0.000875	2.88965e-05	4.91711	:hl:`False`
233	0.00103529	1.80806e-05	0.00091045	2.93845e-05	3.61824	:hl:`False`
235	0.00100917	1.656e-05	0.00088475	2.90315e-05	3.72257	:hl:`False`
237	0.00100023	2.05546e-05	0.00084455	2.8471e-05	4.4335	:hl:`False`
239	0.00100228	1.58244e-05	0.0008346	2.83305e-05	5.1674	:hl:`False`
241	0.000983864	1.50035e-05	0.00083185	2.82915e-05	4.74692	:hl:`False`
243	0.000997541	1.85275e-05	0.00085315	2.8592e-05	4.23805	:hl:`False`
245	0.00104778	2.1722e-05	0.00088735	2.90675e-05	4.42114	:hl:`False`
247	0.00100842	1.64345e-05	0.0008887	2.9086e-05	3.58364	:hl:`False`
249	0.00106032	1.56954e-05	0.00093345	2.9696e-05	3.77718	:hl:`False`
251	0.00106348	1.61786e-05	0.00095775	3.00225e-05	3.1002	:hl:`False`
253	0.00107137	2.16596e-05	0.0009621	3.008e-05	2.948	:hl:`False`
255	0.00110801	1.69586e-05	0.0009755	3.0259e-05	3.82009	:hl:`False`
257	0.00102729	1.65585e-05	0.00087365	2.8878e-05	4.61556	:hl:`False`
259	0.000973585	1.50525e-05	0.0008456	2.8486e-05	3.9724	:hl:`False`
261	0.00099714	1.87934e-05	0.0007888	2.76745e-05	6.22793	:hl:`False`
263	0.000950634	1.59219e-05	0.0007969	2.7792e-05	4.79972	:hl:`False`
265	0.000881015	1.43211e-05	0.0007551	2.71325e-05	4.10412	:hl:`False`
267	0.000854355	1.43856e-05	0.00074015	2.6961e-05	3.73723	:hl:`False`
269	0.000816657	1.52233e-05	0.0006776	2.6014e-05	4.61354	:hl:`False`
271	0.000765727	1.35224e-05	0.00065885	2.57235e-05	3.67766	:hl:`False`
273	0.000736658	1.50207e-05	0.0005968	2.4733e-05	4.83321	:hl:`False`
275	0.000691288	1.37425e-05	0.0006269	2.5221e-05	2.24177	True
277	0.000678993	1.28367e-05	0.0006212	2.513e-05	2.04803	True
279	0.000696424	1.6121e-05	0.00060335	2.4844e-05	3.14267	:hl:`False`
281	0.000670687	1.65141e-05	0.0005857	2.4557e-05	2.87185	:hl:`False`
283	0.000651523	1.276e-05	0.00052105	2.3475e-05	4.8832	:hl:`False`
285	0.000599761	1.26954e-05	0.00053375	2.3694e-05	2.45571	True
287	0.00061133	1.33979e-05	0.00055205	2.40015e-05	2.15659	True
289	0.000601374	1.20326e-05	0.0005533	2.40225e-05	1.78929	True
291	0.000583103	1.21542e-05	0.0005488	2.3947e-05	1.27733	True
293	0.000584056	1.20167e-05	0.00053585	2.37295e-05	1.81235	True
295	0.000540916	1.1966e-05	0.00049291	2.29925e-05	1.85209	True
297	0.000550412	1.21389e-05	0.00052315	2.3514e-05	1.03022	True
299	0.000552923	1.18554e-05	0.000479485	2.27575e-05	2.86191	:hl:`False`
301	0.000560512	1.15318e-05	0.00052695	2.35785e-05	1.27866	True
303	0.000531155	1.09718e-05	0.000483245	2.28235e-05	1.8919	True
305	0.000518599	1.11668e-05	0.000484415	2.2844e-05	1.34438	True

(continues on next page)

(continued from previous page)

307	0.000492058	1.07867e-05	0.000483095	2.2821e-05	0.355101	True
309	0.000505487	1.69899e-05	0.000488225	2.2911e-05	0.605184	True
311	0.000509219	1.7619e-05	0.000488575	2.2917e-05	0.714139	True
313	0.000501507	1.25527e-05	0.000487045	2.289e-05	0.553988	True
315	0.000488588	1.06051e-05	0.000465225	2.2505e-05	0.9391	True
317	0.000483927	1.75318e-05	0.00050375	2.3191e-05	-0.681862	True
319	0.00049458	1.08281e-05	0.00048882	2.29215e-05	0.22722	True
321	0.000482865	1.05604e-05	0.00044683	2.2175e-05	1.46716	True
323	0.000477005	1.00847e-05	0.000452225	2.2272e-05	1.01353	True
325	0.000480317	9.92964e-06	0.000467225	2.25405e-05	0.531542	True
327	0.000494829	1.07151e-05	0.00047503	2.2679e-05	0.789352	True
329	0.000491724	1.07552e-05	0.00047855	2.2741e-05	0.5237	True
331	0.000507182	1.0728e-05	0.00049157	2.29695e-05	0.615827	True
333	0.000498105	1.0642e-05	0.000487365	2.2896e-05	0.425391	True
335	0.000502847	1.08098e-05	0.00049807	2.30825e-05	0.187432	True
337	0.000505027	1.24756e-05	0.00045714	2.23605e-05	1.87018	True
339	0.000507307	1.00953e-05	0.000486445	2.288e-05	0.834194	True
341	0.000489606	1.18542e-05	0.0005155	2.3383e-05	-0.987711	True
343	0.000473202	1.02049e-05	0.000481145	2.2757e-05	-0.318474	True
345	0.000475292	1.00383e-05	0.000489915	2.29405e-05	-0.583981	True
347	0.000480632	9.63098e-06	0.00044432	2.21295e-05	1.50456	True
349	0.000465248	9.71176e-06	0.000444205	2.21275e-05	0.870816	True
351	0.000475173	9.77166e-06	0.000448605	2.2207e-05	1.09505	True
353	0.000437251	9.24249e-06	0.00045204	2.2269e-05	-0.613391	True
355	0.000471841	9.91968e-06	0.00042395	2.17575e-05	2.00281	True
357	0.000451007	9.16715e-06	0.000441215	2.2073e-05	0.409681	True
359	0.00043615	9.44434e-06	0.00046094	2.24285e-05	-1.01865	True
361	0.000425573	1.07187e-05	0.000448195	2.21995e-05	-0.917668	True
363	0.000430618	9.78274e-06	0.000444745	2.2137e-05	-0.583687	True
365	0.000418642	8.511e-06	0.00043726	2.2001e-05	-0.789255	True
367	0.000410464	9.2263e-06	0.000417455	2.16375e-05	-0.297205	True
369	0.000390776	7.92623e-06	0.00038885	2.11005e-05	0.0854315	True
371	0.00037246	8.0628e-06	0.00037076	2.0754e-05	0.0763566	True
373	0.000381532	8.19401e-06	0.00036254	2.0595e-05	0.856856	True
375	0.000359002	7.44099e-06	0.000398215	2.1278e-05	-1.7396	True
377	0.000346426	7.51633e-06	0.000357555	2.04975e-05	-0.509748	True
379	0.000354585	7.41695e-06	0.000324985	1.98505e-05	1.39682	True
381	0.000344207	7.06501e-06	0.000342015	2.01915e-05	0.102481	True
383	0.0003436	7.10479e-06	0.00031319	1.96105e-05	1.45796	True
385	0.000327421	6.915e-06	0.000326635	1.98835e-05	0.0373397	True
387	0.000309169	6.48171e-06	0.000303065	1.94025e-05	0.298385	True
389	0.000333483	8.11078e-06	0.000270825	1.87245e-05	3.07061	:hl:`False`
391	0.000301403	6.34478e-06	0.000296465	1.92655e-05	0.243443	True
393	0.000304476	6.32524e-06	0.00028084	1.8933e-05	1.18408	True
395	0.000291929	6.22648e-06	0.0002719	1.87475e-05	1.01391	True
397	0.000284313	6.04437e-06	0.000263645	1.85705e-05	1.05831	True
399	0.000269536	5.84774e-06	0.00025017	1.8277e-05	1.00918	True
401	0.000272009	6.21651e-06	0.00022047	1.76135e-05	2.7593	:hl:`False`
403	0.00025244	5.50901e-06	0.00020573	1.7275e-05	2.57609	True
405	0.000237807	5.47488e-06	0.000211075	1.73985e-05	1.46558	True
407	0.000227296	5.32532e-06	0.00019417	1.70045e-05	1.85907	True
409	0.00022099	5.11706e-06	0.00016967	1.6417e-05	2.98444	:hl:`False`



L'impression des résultats des `TestStudent` dans le cas `FULL_DETAILS` donne le tableau de Student en `INTERMEDIATE`, soit les bins où le test à échouer.

Remarque : le test de Holm-Bonferroni n'est pas très adapté ici, comme nous comparons les données à la simulation, nous n'avons pas de nombre de degrés de liberté similaire pour les deux échantillons.

Graphiques de comparaison entre les données expérimentales et Tripoli-4

L'analyse des données peut également être faite en dehors de **valjean** en fonction de ses nécessités.

Comme pour la lecture des résultats expérimentaux, une petite classe a été dérivée pour facilier et personnaliser les graphiques de comparaison.

La bibliothèque utilisée est `matplotlib`.

Dans notre cas, on souhaite aisément :

- ajouter une nouvelle courbe, avec ses erreurs
- visualiser le rapport entre les différentes courbes
- pouvoir personnaliser facilement la couleur et l'aspect des courbes (aisé grâce à `matplotlib`, les arguments sont juste transmis ici)

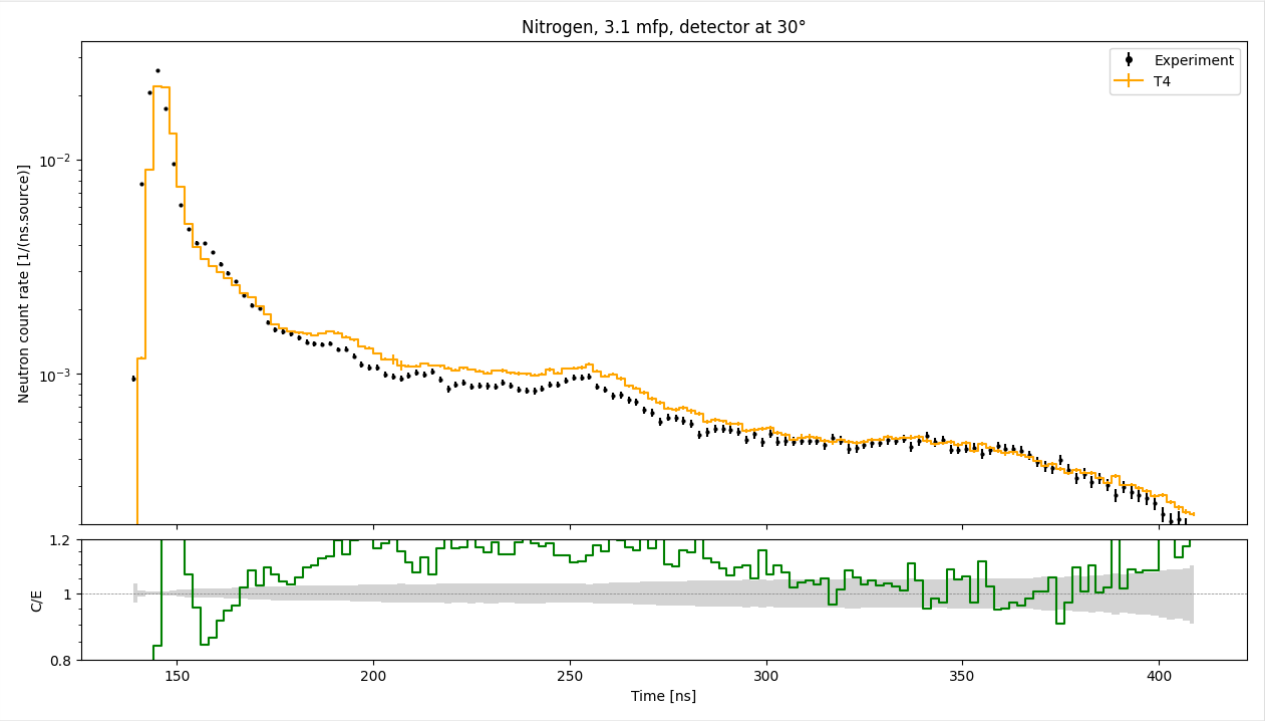
Cette petite classe, CompPlot, est aussi disponible dans le notebook.

Le nom de « l'analyse » correspond à la clef pour les données expérimentales. Cela permet de générer par exemple le titre.

[36]:
```
import matplotlib.pyplot as plt
from comp_plots import CompPlot

cplot = CompPlot(exp_key)
cplot.add_errorbar_plot(exp_data.bins['t'], exp_data.value, exp_data.error,
                        fmt='o', c='black', ms=2, ecolor='black', label='Experiment')
cplot.add_errorbar_plot(t4ds.bins['t'], t4ds.value, t4ds.error,
                        label='T4', drawstyle='steps-mid', fmt='-', c='orange')

cplot.add_errorbar_ratio(ratio.bins['t'], ratio.value, 0,
                         drawstyle='steps-mid', fmt='-', c='green')
cplot.splt[1].fill_between(
    ratio.bins['t'],
    np.ones(ratio.bins['t'].size) - exp_data.error/exp_data.value,
    np.ones(ratio.bins['t'].size) + exp_data.error/exp_data.value,
    facecolor='lightgrey', step='mid')
cplot.customize_plot()
plt.show()
```



3.1.6 Benchmark REPLICA : exemples de parsing

Principe du parsing des sorties Tripoli-4 standard

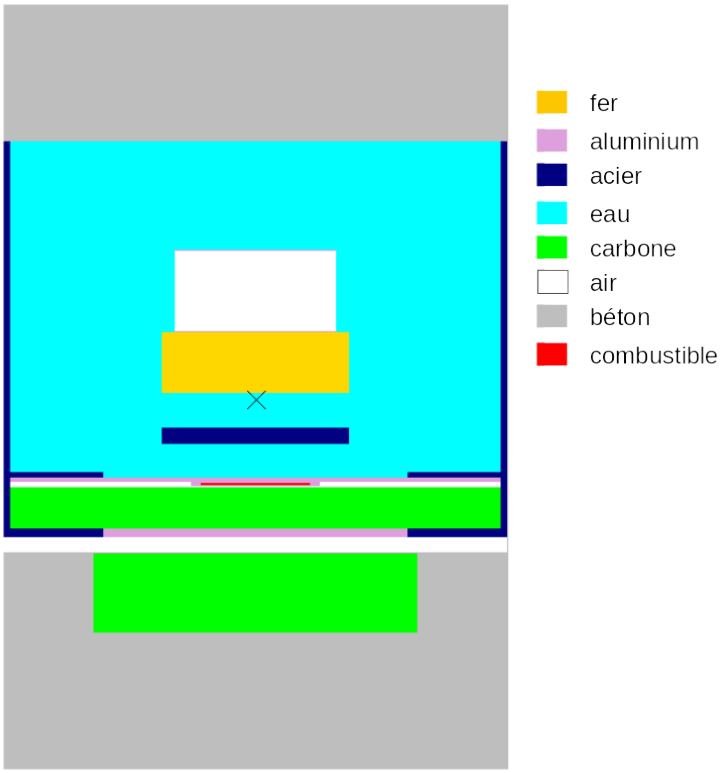
Le parsing est effectué en deux temps :

- un premier parcours rapide du fichier répérant un certain nombre de balises (nombre de bacthes requis, temps d'initialisation, de simulation, `NORMAL COMPLETION`, etc) → module `scan.py`. Ce parcours permet également de repérer les résultats dans la sortie, débutant par `RESULTS ARE GIVEN` et terminant le plus souvent par `'simulation time:'`.
- le réel parsing des résultats, piloté par le module `parse.py` qui appelle la grammaire (utilisant `pyparsing`) et transformant le résultat en objets python standards (listes, dictionnaires, tableaux numpy).

Exemple de parsing, jeu de données `fast_neutron.t4`

Présentation du jeu de données

Le jeu de données est issu du benchmark REPLICA du programme SINBAD, dont la géométrie Tripoli-4 est représentée ci-dessous. Il s'agit ici de la simulation des neutrons rapides.



2 types de réponses sont attendus :

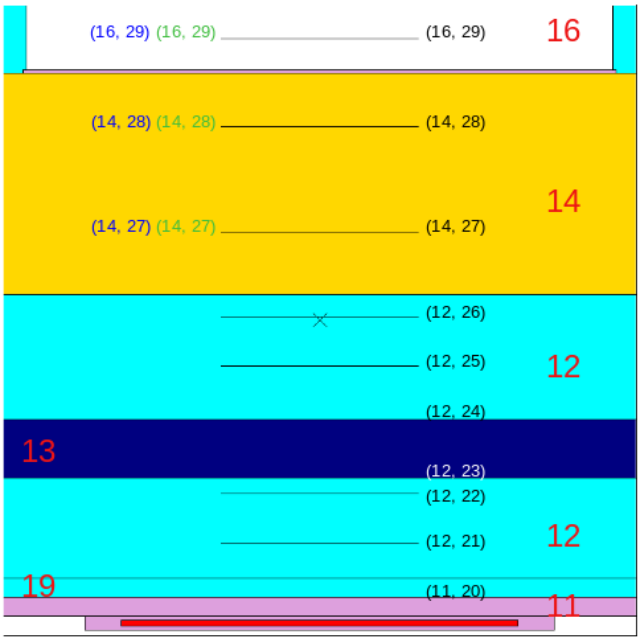
- réactions dans des détecteurs
- flux surfacique

Dans le cas des neutrons rapides on dispose de 3 détecteurs : ^{103}Rh (`RH103_IRDF85`), ^{115}In (`IN115_IRDF85`) et ^{32}S (`S32_IRDF85`).

On fait ces mesures à différents emplacements. Le résultat final est constitué de :

- 10 réactions sur le ^{103}Rh (10 frontières de volume différentes)
- 3 réactions sur le ^{115}In
- 3 réactions sur le ^{32}S
- 2 flux surfaciques

Les emplacements des mesures sont représentés ci-dessous.



N : numéro de volume
(I, J) : détecteur (frontière de volumes)
- noir : Rh103
- bleu : In115
- vert : S32
Flux surfaciques sur (14, 27) et (16, 29)

Étape 1 : scan puis parsing du résultat

On charge le module `Parser`. Le jeu de données est scanné automatiquement, tous les résultats de batches sont stockés dans la variable `scan_res`. On peut ensuite parser ceux qui nous intéressent.

Par défaut le résultat parsé sera le dernier batch, mais on pourrait en parser un autre.

Dernier batch → -1

Scan du jeu de données

[1]:
```
from valjean.eponine.tripoli4.parse import Parser
```

[2]:
```
t4vv_replica_fast = 'fast_neutron.res'
t4p = Parser(t4vv_replica_fast)
```

```
* Parsing fast_neutron.res
* Edition batch (-1) different from current batch (50)
```

(continues on next page)

(continued from previous page)

```
* If no Edition batch keep current batch, else keep edition batch
* Edition batch (-1) different from current batch (100)
* If no Edition batch keep current batch, else keep edition batch
* Successful scan in 0.012149 s
```

Parmi les membres de `t4p` il y a le résultat du scan. Certains paramètres accessibles à partir de ce résultat sont montrés ici :

- parallèle ou mono-processeur (booléen)
- les temps

[3]:
```
t4p.scan_res.normalend
```

[3]:
```
True
```

[4]:
```
t4p.scan_res.para
```

[4]:
```
False
```

[5]:
```
type(t4p.scan_res)
```

[5]:
```
valjean.eponine.tripoli4.scan.Scanner
```

[6]:
```
t4p.scan_res.times
```

[6]:
```
OrderedDict([('initialization_time', 12),
             ('simulation_time', {50: 4, 100: 9})])
```

Parsing du jeu de données

Le jeu de données peut être parser soit par numéro du batch quand on le connaît, grâce à la méthode `parse_from_number`, soit par son index dans la liste des batches scannés grâce à la méthode `parse_from_index`.

Le plus souvent c'est le dernier batch qui nous intéresse, le défaut de `parse_from_index` est donc `-1`.

Ces méthodes renvoient un `ParseResult` qui contient le batch parsé ainsi que les variables globales du jeu de données récupérées lors du scan (objet `res`).

Le `ParseResult` peut alors être transformé en `Browser` pour faciliter l'accès aux différents résultats grâce à la méthode `to_browser`.

[7]:
```
t4pres = t4p.parse_from_index()
```

```
* Successful parsing in 0.058571 s
```

[8]:
```
len(t4pres.res), type(t4pres.res), list(t4pres.res.keys())
```

[8]:
```
(3, dict, ['list_responses', 'batch_data', 'run_data'])
```

[9]:
```
print('run_data:', t4pres.res['run_data'])
print('batch_data:', t4pres.res['batch_data'])
```

```
run_data: {'warnings': 1, 'errors': 0, 'number_of_tasks': 1, 'normal_end': True,
↪'required_batches': 100, 'partial': False, 't4_file': 'fast_neutron.res',
↪'initialization_time': 12}
batch_data: {'source_intensity': 81114.52, 'mean_weight_leak': {'score': 359.8768,
↪'sigma': 143.1067, 'sigma%': 39.76546}, 'simulation_time': 9, 'batch_number': 100,
↪'name': ''}
```

Le résultat stocké dans `ParseResult` est un dictionnaire.

- `'batch_data'` correspond aux données spécifiques du batch (temps de simulation cumulé à la fin de celui-ci, numéro d'édition, intensité de la source, etc.)
- `'run_data'` correspond aux données globales du jeu de données (temps d'initialisation, les nombres d'erreurs et de warnings, le nom du jeu de données, etc.
- `'list_responses'` est la liste des résultats (liste de dictionnaires)

[10]:
```
print(type(t4pres.res['list_responses']))
print(type(t4pres.res['list_responses'][-1]))
```

```
<class 'list'>
<class 'dict'>
```

Les différents résultats sont stockés dans la liste dans l'ordre où ils apparaissent dans la sortie de TRIPOLI4, sous forme de dictionnaires.

Exemple court :

[11]:
```
print(t4pres.res['list_responses'][0])
```

```
{'response_function': 'REACTION', 'response_name': 'reaction_Rh103', 'score_name':
↪'reaction_Rh103_l10surf', 'energy_split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON
↪', 'response_type': 'score', 'reaction_on_nucleus': ('RH103_IRDF85',), 'composition
↪': ('none',), 'concentration': (1.0,), 'reaction': ('tabulated data',), 'response_
↪index': 0, 'scoring_mode': 'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_
↪zone_id': (11, 20), 'score_index': 0, 'results': {'discarded_batches': class:
↪<class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 16594.9,
        error: 367.99804761300004,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 585.8918,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
```

(continues on next page)

(continued from previous page)

```
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 16594.9,
       error: 367.99804761300004,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}}
```

Pour manipuler plus aisément les résultats un `Browser` est disponible :

[12]:
```
t4b = t4pres.to_browser()
```

[13]:
```
print(t4b)
```

```
Browser object -> Number of content items: 18, data key: 'results', available␣
↪metadata keys: ['composition', 'concentration', 'energy_split_name', 'index',
↪'particle', 'reaction', 'reaction_on_nucleus', 'response_function', 'response_index
↪', 'response_name', 'response_type', 'score_index', 'score_name', 'scoring_mode',
↪'scoring_zone_id', 'scoring_zone_type']
              -> Number of globals: 5
```

Il permet notamment de faire la sélection des résultats à partir des métadonnées du cas considéré. Leur nom est donné dans le `print` du `Browser`.

Les résultats de Tripoli-4 apparaissent sous la forme d'une liste de réponses (ou d'une liste de scores) dans le listing de sortie. Le `Browser` permet de les sélectionner de manière plus efficace.

Les résultats sont encapsulés dans un `Dataset`.

Étape 2 : sélection des données grâce au `Browser`

On différencie dans chaque "réponse" les données, soit les résultats du calcul effectué, et les métadonnées qui permettent de l'identifier. Chaque réponse est un dictionnaire dont la clef `'results'` correspond aux données. Le `Browser` est construit à partir de cela.

Le `Browser` contient notamment :

- la liste des résultats et de leurs métadonnées (`content`)
- un index sur ces résultats (`index`), basé sur les métadonnées
- les variables globales du batch (`globals`)

À noter : il est possible d'utiliser une autre clef que `'results'` pour les données, à condition de le spécifier à la créaction du `Browser` grâce à l'argument `data_key`.

Explorer le Browser

2 niveaux d'impression sont également disponibles pour le Browser.

Méthodes d'aide pour découvrir le Browser :

- keys: pour obtenir toutes les clefs de l'index, soit le nom / identifiant des métadonnées du listing concerné
- available_values: pour obtenir les valeurs correspondant à ces métadonnées

Dans ces deux cas il s'agit de *générateurs*. Pour afficher correctement les résultats il faut les encapsuler dans une liste par exemple.

[14]:
```
print('{!s}'.format(t4b))  # équivalent à print(t4rb)
```

```
Browser object -> Number of content items: 18, data key: 'results', available␣
↪metadata keys: ['composition', 'concentration', 'energy_split_name', 'index',
↪'particle', 'reaction', 'reaction_on_nucleus', 'response_function', 'response_index
↪', 'response_name', 'response_type', 'score_index', 'score_name', 'scoring_mode',
↪'scoring_zone_id', 'scoring_zone_type']
               -> Number of globals: 5
```

[15]:
```
print('{!r}'.format(t4b))
```

```
<class 'valjean.eponine.browser.Browser'>, (Content items: [{'response_function':
↪'REACTION', 'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf
↪', 'energy_split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type':
↪'score', 'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',),
↪'concentration': (1.0,), 'reaction': ('tabulated data',), 'response_index': 0,
↪'scoring_mode': 'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id':␣
↪(11, 20), 'score_index': 0, 'results': {'discarded_batches': class: <class 'valjean.
↪eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 16594.9,
        error: 367.99804761300004,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 585.8918,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
```

(continues on next page)

(continued from previous page)

```
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 16594.9,
       error: 367.99804761300004,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}, 'index': 0}, {'response_function': 'REACTION',
↪'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 0, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (12, 21), 'score_
↪index': 1, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 3191.064,
       error: 90.06005902512001,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 112.6622,
       error: nan,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 3191.064,
       error: 90.06005902512001,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}, 'index': 1}, {'response_function': 'REACTION',
↪'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',), 'concentration':
```

(continues on next page)

(continued from previous page)

```
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 0, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (12, 22), 'score_
↪index': 2, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 1203.617,
        error: 35.162358712469995,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 42.49436,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 1203.617,
        error: 35.162358712469995,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction'}, 'index': 2}, {'response_function': 'REACTION',
↪'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 0, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (12, 23), 'score_
↪index': 3, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
```

(continues on next page)

(continued from previous page)

```
       value: 1047.175,
       error: 30.108333712999997,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 36.97106,
       error: nan,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 1047.175,
       error: 30.108333712999997,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}, 'index': 3}, {'response_function': 'REACTION',
↪'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 0, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (12, 24), 'score_
↪index': 4, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 397.2467,
       error: 15.013601366805002,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 14.02501,
       error: nan,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
```

(continues on next page)

(continued from previous page)

```
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 397.2467,
       error: 15.013601366805002,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}, 'index': 4}, {'response_function': 'REACTION',
↪'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 0, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (12, 25), 'score_
↪index': 5, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 98.96879,
       error: 4.0786265571996,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 3.494146,
       error: nan,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 98.96879,
       error: 4.0786265571996,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
```

(continues on next page)

(continued from previous page)

```
        name: '', what: 'reaction'}, 'index': 5}, {'response_function': 'REACTION',
↪'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 0, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (12, 26), 'score_
↪index': 6, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 40.50244,
        error: 1.9534910047136,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 1.429961,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 40.50244,
        error: 1.9534910047136,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction'}, 'index': 6}, {'response_function': 'REACTION',
↪'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 0, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (14, 27), 'score_
↪index': 7, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
```

(continues on next page)

(continued from previous page)

```
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 18.79975,
       error: 1.1705560938875,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 0.6637354,
       error: nan,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 18.79975,
       error: 1.1705560938875,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}, 'index': 7}, {'response_function': 'REACTION',
↪'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 0, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (14, 28), 'score_
↪index': 8, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 5.299455,
       error: 0.3073416807468,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
```

(continues on next page)

(continued from previous page)

```
        value: 0.1871001,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 5.299455,
        error: 0.3073416807468,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction'}, 'index': 8}, {'response_function': 'REACTION',
↪'response_name': 'reaction_Rh103', 'score_name': 'reaction_Rh103_l10surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('RH103_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 0, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (16, 29), 'score_
↪index': 9, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 1.656025,
        error: 0.08065141490524999,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.05846686,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 1.656025,
        error: 0.08065141490524999,
```

(continues on next page)

(continued from previous page)

```
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction'}, 'index': 9}, {'response_function': 'REACTION',
↪'response_name': 'reaction_In115', 'score_name': 'reaction_In115_l3surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('IN115_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 1, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (14, 27), 'score_
↪index': 0, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 3.279737,
        error: 0.16236711908518,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.1157929,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 3.279737,
        error: 0.16236711908518,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction'}, 'index': 10}, {'response_function': 'REACTION',
↪'response_name': 'reaction_In115', 'score_name': 'reaction_In115_l3surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('IN115_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 1, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (14, 28), 'score_
↪index': 1, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
```

(continues on next page)

(continued from previous page)

```
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.7353725,
        error: 0.050511670734875,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.02596272,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.7353725,
        error: 0.050511670734875,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction'}, 'index': 11}, {'response_function': 'REACTION',
↪'response_name': 'reaction_In115', 'score_name': 'reaction_In115_l3surf', 'energy_
↪split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score',
↪'reaction_on_nucleus': ('IN115_IRDF85',), 'composition': ('none',), 'concentration':
↪ (1.0,), 'reaction': ('tabulated data',), 'response_index': 1, 'scoring_mode':
↪'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id': (16, 29), 'score_
↪index': 2, 'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.2086517,
        error: 0.010065887983318,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
```

(continues on next page)

(continued from previous page)

```
↪dtype=float64))]),
       name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 0.007366562,
       error: nan,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 0.2086517,
       error: 0.010065887983318,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}, 'index': 12}, {'response_function': 'REACTION',
↪'response_name': 'reaction_S32', 'score_name': 'reaction_S32_l3surf', 'energy_split_
↪name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score', 'reaction_
↪on_nucleus': ('S32_IRDF85',), 'composition': ('none',), 'concentration': (1.0,),
↪'reaction': ('tabulated data',), 'response_index': 2, 'scoring_mode': 'SCORE_SURF',
↪'scoring_zone_type': 'Frontier', 'scoring_zone_id': (14, 27), 'score_index': 0,
↪'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.Dataset'>,␣
↪data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 1.019999,
       error: 0.07280398922347002,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 0.03601161,
       error: nan,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
```

(continues on next page)

(continued from previous page)

```
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 1.019999,
       error: 0.07280398922347002,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}, 'index': 13}, {'response_function': 'REACTION',
↪'response_name': 'reaction_S32', 'score_name': 'reaction_S32_l3surf', 'energy_split_
↪name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score', 'reaction_
↪on_nucleus': ('S32_IRDF85',), 'composition': ('none',), 'concentration': (1.0,),
↪'reaction': ('tabulated data',), 'response_index': 2, 'scoring_mode': 'SCORE_SURF',
↪'scoring_zone_type': 'Frontier', 'scoring_zone_id': (14, 28), 'score_index': 1,
↪'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.Dataset'>,␣
↪data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 0.1516089,
       error: 0.01610849110767,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 0.005352634,
       error: nan,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 0.1516089,
       error: 0.01610849110767,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}, 'index': 14}, {'response_function': 'REACTION',
↪'response_name': 'reaction_S32', 'score_name': 'reaction_S32_l3surf', 'energy_split_
↪name': 'DEC_INTEGRAL', 'particle': 'NEUTRON', 'response_type': 'score', 'reaction_
↪on_nucleus': ('S32_IRDF85',), 'composition': ('none',), 'concentration': (1.0,),
↪'reaction': ('tabulated data',), 'response_index': 2, 'scoring_mode': 'SCORE_SURF',
```

(continues on next page)

(continued from previous page)

```
↪'scoring_zone_type': 'Frontier', 'scoring_zone_id': (16, 29), 'score_index': 2,
↪'results': {'discarded_batches': class: <class 'valjean.eponine.dataset.Dataset'>,␣
↪data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.03666873,
        error: 0.0029075714077661996,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.001294609,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.03666873,
        error: 0.0029075714077661996,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction'}, 'index': 15}, {'response_function': 'FLUX',
↪'response_name': 'flux', 'score_name': 'flux_l2surf', 'energy_split_name': 'DEC_
↪SPECTRE', 'particle': 'NEUTRON', 'response_type': 'score', 'response_index': 3,
↪'scoring_mode': 'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id':␣
↪(14, 27), 'score_index': 0, 'results': {'discarded_batches': class: <class 'valjean.
↪eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 44, 1, 1, 1)
        value: [0.1528255  0.1498575  0.4623927  0.4774755  0.5853392  0.4022689
 0.5847542  0.5541887  1.329951   0.4184096  1.270775   0.8474643
 2.227431   0.7243235  0.7575581  1.973943   4.411218   1.044785
```

(continues on next page)

(continued from previous page)

```
1.759198   3.106783   2.932957   3.357995   2.410037   1.634667
2.14932    2.402607   1.557616   1.298836   1.619081   1.204458
0.9177346  0.7668855  0.8060953  0.7114108  0.2920371  0.4349422
0.380331   0.3279657  0.3124642  0.3140535  0.1289041  0.1411806
0.08703017 0.08508481],
       error: [0.05632055 0.05018791 0.12369583 0.11418201 0.19987718 0.11761779
0.1277354  0.1470686  0.2948367  0.09983287 0.26088197 0.42792337
1.05125321 0.25577992 0.22138613 0.75998345 1.58046045 0.22374113
0.27835948 1.17676365 0.49418419 0.85799627 0.57635312 0.3012451
0.43288272 0.55326417 0.21334152 0.17371282 0.19080271 0.13713802
0.12052911 0.11073857 0.10772327 0.09604771 0.05286599 0.08091682
0.05768724 0.05819502 0.0658838  0.08481524 0.03489702 0.05760408
0.03272591 0.04466015],
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([ 0.0463,  0.0525,  0.
↪0595,  0.0674,  0.0764,  0.0865,  0.097 ,
       0.111 ,  0.126 ,  0.143 ,  0.162 ,  0.183 ,  0.207 ,  0.235 ,
       0.266 ,  0.302 ,  0.342 ,  0.388 ,  0.439 ,  0.498 ,  0.564 ,
       0.639 ,  0.724 ,  0.821 ,  0.93  ,  1.054 ,  1.194 ,  1.353 ,
       1.534 ,  1.738 ,  1.969 ,  2.231 ,  2.528 ,  2.865 ,  3.246 ,
       3.679 ,  4.169 ,  4.724 ,  5.353 ,  6.065 ,  6.873 ,  7.788 ,
       8.825 , 10.    , 20.    ])), ('t', array([], dtype=float64)), ('mu', array([],
↪ dtype=float64)), ('phi', array([], dtype=float64))]),
       name: '', what: 'flux', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 44, 1, 1, 1)
       value: [ 1.216075   1.197297   3.708972   3.809515   4.714329   3.511222
 4.337321   4.372238  10.50823    3.35394   10.42561    6.876946
17.55725    5.845523   5.968283  15.86978   34.95567    8.460204
13.95075   24.96327   23.49181   26.88829   19.16809   13.11284
17.17214   19.26454   12.45939   10.34481   12.96755    9.651824
 7.346338   6.136133   6.441557   5.69789    2.33224    3.478555
 3.043143   2.623699   2.502168   2.511099   1.03137    1.129404
 0.6962608  0.1227514],
       error: [nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan␣
↪nan
nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
nan nan nan nan nan nan nan nan],
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([ 0.0463,  0.0525,  0.
↪0595,  0.0674,  0.0764,  0.0865,  0.097 ,
       0.111 ,  0.126 ,  0.143 ,  0.162 ,  0.183 ,  0.207 ,  0.235 ,
       0.266 ,  0.302 ,  0.342 ,  0.388 ,  0.439 ,  0.498 ,  0.564 ,
       0.639 ,  0.724 ,  0.821 ,  0.93  ,  1.054 ,  1.194 ,  1.353 ,
       1.534 ,  1.738 ,  1.969 ,  2.231 ,  2.528 ,  2.865 ,  3.246 ,
       3.679 ,  4.169 ,  4.724 ,  5.353 ,  6.065 ,  6.873 ,  7.788 ,
       8.825 , 10.    , 20.    ])), ('t', array([], dtype=float64)), ('mu', array([],
↪ dtype=float64)), ('phi', array([], dtype=float64))]),
       name: '', what: 'flux/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 49.51463,
       error: 3.5046068899886,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
```

(continues on next page)

(continued from previous page)

```
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([ 0.0463, 20.    ])),␣
↪('t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],
↪ dtype=float64))]),
       name: '', what: 'flux'}, 'index': 16}, {'response_function': 'FLUX',
↪'response_name': 'flux', 'score_name': 'flux_l2surf', 'energy_split_name': 'DEC_
↪SPECTRE', 'particle': 'NEUTRON', 'response_type': 'score', 'response_index': 3,
↪'scoring_mode': 'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_id':␣
↪(16, 29), 'score_index': 1, 'results': {'discarded_batches': class: <class 'valjean.
↪eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
       shape: (1, 1, 1, 44, 1, 1, 1)
       value: [0.03050079 0.06832957 0.07695755 0.09925667 0.1350992  0.02155418
 0.08208997 0.1378661  0.3047329  0.07209889 0.2593902  0.1593264
 0.2394927  0.1843582  0.4052755  0.3252563  0.4268424  0.1484201
 0.2554633  0.2897066  0.4703281  0.498303   0.2471051  0.2301476
 0.2617021  0.1997403  0.1476602  0.121049   0.09662921 0.09268166
 0.05543372 0.04428578 0.03727038 0.02329219 0.01471019 0.01064267
 0.01057807 0.010661   0.00989727 0.00583138 0.00626957 0.00349015
 0.00309229 0.00675048],
       error: [0.01210843 0.03119505 0.01720868 0.01918197 0.02985159 0.00884732
 0.01680104 0.02190428 0.03902812 0.01818942 0.03283141 0.03104628
 0.09498242 0.06235245 0.09121256 0.059099   0.08275433 0.0348335
 0.04874998 0.05476054 0.07168115 0.07391498 0.0579394  0.04094811
 0.06295158 0.02666253 0.02132944 0.01697825 0.01354986 0.01089185
 0.00827571 0.00729076 0.00627467 0.00498183 0.00346623 0.00270574
 0.00260966 0.0023411  0.00240927 0.00189545 0.00196971 0.00159711
 0.00149497 0.0047856 ],
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([ 0.0463,  0.0525,  0.
↪0595,  0.0674,  0.0764,  0.0865,  0.097 ,
       0.111 ,  0.126 ,  0.143 ,  0.162 ,  0.183 ,  0.207 ,  0.235 ,
       0.266 ,  0.302 ,  0.342 ,  0.388 ,  0.439 ,  0.498 ,  0.564 ,
       0.639 ,  0.724 ,  0.821 ,  0.93  ,  1.054 ,  1.194 ,  1.353 ,
       1.534 ,  1.738 ,  1.969 ,  2.231 ,  2.528 ,  2.865 ,  3.246 ,
       3.679 ,  4.169 ,  4.724 ,  5.353 ,  6.065 ,  6.873 ,  7.788 ,
       8.825 , 10.    , 20.    ])), ('t', array([], dtype=float64)), ('mu', array([],
↪ dtype=float64)), ('phi', array([], dtype=float64))]),
       name: '', what: 'flux', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 44, 1, 1, 1)
       value: [0.2427032  0.5459238  0.6172965  0.7919144  1.088091   0.1881366
 0.6088893  1.087686   2.407761   0.5779393  2.128071   1.292891
 1.88775    1.48783    3.192889   2.614941   3.382414   1.20184
 2.025869   2.327817   3.76714    3.990034   1.965336   1.84618
 2.090888   1.601554   1.181136   0.9641162  0.7739233  0.7426966
 0.4437392  0.3543468  0.2978298  0.1865537  0.1174771  0.08511736
 0.08463831 0.08528712 0.07925593 0.04662634 0.05016322 0.02792023
 0.024739   0.00973889],
       error: [nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan␣
```

(continues on next page)

(continued from previous page)

```
↪nan
 nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan nan
 nan nan nan nan nan nan nan nan],
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([ 0.0463,  0.0525,  0.
↪0595,  0.0674,  0.0764,  0.0865,  0.097 ,
       0.111 ,  0.126 ,  0.143 ,  0.162 ,  0.183 ,  0.207 ,  0.235 ,
       0.266 ,  0.302 ,  0.342 ,  0.388 ,  0.439 ,  0.498 ,  0.564 ,
       0.639 ,  0.724 ,  0.821 ,  0.93  ,  1.054 ,  1.194 ,  1.353 ,
       1.534 ,  1.738 ,  1.969 ,  2.231 ,  2.528 ,  2.865 ,  3.246 ,
       3.679 ,  4.169 ,  4.724 ,  5.353 ,  6.065 ,  6.873 ,  7.788 ,
       8.825 , 10.    , 20.    ])), ('t', array([], dtype=float64)), ('mu', array([],
↪ dtype=float64)), ('phi', array([], dtype=float64))]),
       name: '', what: 'flux/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
       shape: (1, 1, 1, 1, 1, 1, 1)
       value: 6.329569,
       error: 0.32567252874664,
       bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([ 0.0463, 20.    ])),␣
↪('t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],
↪ dtype=float64))]),
       name: '', what: 'flux'}, 'index': 17}], Index: defaultdict(<function _make_
↪defaultdict_set at 0x7fc7b4c10940>, {'response_function': defaultdict(<class 'set'>,
↪ {'REACTION': {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}, 'FLUX': {16,␣
↪17}}), 'response_name': defaultdict(<class 'set'>, {'reaction_Rh103': {0, 1, 2, 3,␣
↪4, 5, 6, 7, 8, 9}, 'reaction_In115': {10, 11, 12}, 'reaction_S32': {13, 14, 15},
↪'flux': {16, 17}}), 'score_name': defaultdict(<class 'set'>, {'reaction_Rh103_
↪l10surf': {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, 'reaction_In115_l3surf': {10, 11, 12},
↪'reaction_S32_l3surf': {13, 14, 15}, 'flux_l2surf': {16, 17}}), 'energy_split_name':
↪ defaultdict(<class 'set'>, {'DEC_INTEGRAL': {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,␣
↪12, 13, 14, 15}, 'DEC_SPECTRE': {16, 17}}), 'particle': defaultdict(<class 'set'>, {
↪'NEUTRON': {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}}),
↪'response_type': defaultdict(<class 'set'>, {'score': {0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
↪ 10, 11, 12, 13, 14, 15, 16, 17}}), 'reaction_on_nucleus': defaultdict(<class 'set'>
↪, {('RH103_IRDF85',): {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, ('IN115_IRDF85',): {10, 11,␣
↪12}, ('S32_IRDF85',): {13, 14, 15}}), 'composition': defaultdict(<class 'set'>, {(
↪'none',): {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}}), 'concentration':
↪ defaultdict(<class 'set'>, {(1.0,): {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13,␣
↪14, 15}}), 'reaction': defaultdict(<class 'set'>, {('tabulated data',): {0, 1, 2, 3,
↪ 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15}}), 'response_index': defaultdict(<class
↪'set'>, {0: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, 1: {10, 11, 12}, 2: {13, 14, 15}, 3:
↪{16, 17}}), 'scoring_mode': defaultdict(<class 'set'>, {'SCORE_SURF': {0, 1, 2, 3,␣
↪4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17}}), 'scoring_zone_type':␣
↪defaultdict(<class 'set'>, {'Frontier': {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12,␣
↪13, 14, 15, 16, 17}}), 'scoring_zone_id': defaultdict(<class 'set'>, {(11, 20): {0},
↪ (12, 21): {1}, (12, 22): {2}, (12, 23): {3}, (12, 24): {4}, (12, 25): {5}, (12,␣
↪26): {6}, (14, 27): {16, 10, 13, 7}, (14, 28): {8, 11, 14}, (16, 29): {9, 12, 17,␣
↪15}}), 'score_index': defaultdict(<class 'set'>, {0: {0, 16, 10, 13}, 1: {1, 11, 14,
↪ 17}, 2: {2, 12, 15}, 3: {3}, 4: {4}, 5: {5}, 6: {6}, 7: {7}, 8: {8}, 9: {9}}),
↪'index': defaultdict(<class 'set'>, {0: {0}, 1: {1}, 2: {2}, 3: {3}, 4: {4}, 5: {5},
↪ 6: {6}, 7: {7}, 8: {8}, 9: {9}, 10: {10}, 11: {11}, 12: {12}, 13: {13}, 14: {14},␣
↪15: {15}, 16: {16}, 17: {17}})}))
```

[16]:
```
list(t4b.keys())
```

[16]:
```
['response_function',
 'response_name',
 'score_name',
 'energy_split_name',
 'particle',
 'response_type',
 'reaction_on_nucleus',
 'composition',
 'concentration',
 'reaction',
 'response_index',
 'scoring_mode',
 'scoring_zone_type',
 'scoring_zone_id',
 'score_index',
 'index']
```

[17]:
```
print('values for score_name:', list(t4b.available_values('score_name')))
print('values for response_function:', list(t4b.available_values('response_function
↪')))
print('values for scoring_zone_id:', list(t4b.available_values('scoring_zone_id')))
print('values for particle:', list(t4b.available_values('particle')))
print('values for reaction_on_nucleus:', list(t4b.available_values('reaction_on_
↪nucleus')))
```

```
values for score_name: ['reaction_Rh103_l10surf', 'reaction_In115_l3surf', 'reaction_
↪S32_l3surf', 'flux_l2surf']
values for response_function: ['REACTION', 'FLUX']
values for scoring_zone_id: [(11, 20), (12, 21), (12, 22), (12, 23), (12, 24), (12,␣
↪25), (12, 26), (14, 27), (14, 28), (16, 29)]
values for particle: ['NEUTRON']
values for reaction_on_nucleus: [('RH103_IRDF85',), ('IN115_IRDF85',), ('S32_IRDF85',
↪)]
```

Sélection des réponses / résultats grâce au `Browser`

Il est possible de sélectionner les réponses à partir des métadonnées. La méthode à choisir dépend de l'objet dont on a besoin en sortie.

- `filter_by` pour récupérer un sous-`Browser`
- `select_by` pour récupérer la réponse si elle est unique

La sélection se fait

- par mot-clef/valeur, soit métadonnée/valeur de cette métadonnée (*keyword arguments* correspondant aux clefs disponibles)
- `include=tuple(key)` pour sélectionner toutes les réponses contenant cette métadonnée sans distinction de la valeur
- `exclude=tuple(key)` pour exclure les réponses contenant une métadonnée

`include` et `exclude` ne fonctionnent que sur le nom de la métadonnée, pas sur sa valeur.

Ces 3 possibilités sont bien sûr combinables.

[18]:
```
b_flux = t4b.filter_by(response_function='FLUX')
print(b_flux)
```

```
Browser object -> Number of content items: 2, data key: 'results', available metadata␣
↪keys: ['energy_split_name', 'index', 'particle', 'response_function', 'response_
↪index', 'response_name', 'response_type', 'score_index', 'score_name', 'scoring_mode
↪', 'scoring_zone_id', 'scoring_zone_type']
               -> Number of globals: 5
```

Le nombre de réponses dans le `Browser` n'est maintenant plus que de 2 au lieu de 18, il ne reste plus que celles correspondant à un flux. Récupérer les réponses peut se faire directement ou avec l'autre méthode.

Le sous-`Browser` remet à zéro les index (en crée un nouveau en réalité).

À noter : les variables globales du `Browser` sont transmises au sous-`Browser`.

[19]:
```
b_flux.globals == t4b.globals
```

[19]:
```
True
```

[20]:
```
b_excl_compo = t4b.filter_by(exclude=('composition',))  # attention à bien utiliser␣
↪un tuple
print(b_excl_compo)
```

```
Browser object -> Number of content items: 2, data key: 'results', available metadata␣
↪keys: ['energy_split_name', 'index', 'particle', 'response_function', 'response_
↪index', 'response_name', 'response_type', 'score_index', 'score_name', 'scoring_mode
↪', 'scoring_zone_id', 'scoring_zone_type']
               -> Number of globals: 5
```

Les réponses ont été « aplaties » (ou *flattened*). Dans le jeu de données d'origine 3 réponses sont demandées sur une liste de scoring zones à chaque fois.

```
SCORE
    4
    NAME reaction_Rh103_l10surf
    reaction_Rh103
    SURF DECOUPAGE DEC_INTEGRAL FRONTIER LIST 10
        11 20   12 21   12 22   12 23   12 24
        12 25   12 26   14 27   14 28   16 29
    NAME reaction_In115_l3surf
    reaction_In115
    SURF DECOUPAGE DEC_INTEGRAL FRONTIER LIST 3
        14 27   14 28   16 29
    NAME reaction_S32_l3surf
    reaction_S32
    SURF DECOUPAGE DEC_INTEGRAL FRONTIER LIST 3
        14 27   14 28   16 29
    NAME flux_l2surf
    flux
    SURF DECOUPAGE DEC_SPECTRE FRONTIER LIST 2
        14 27   16 29
FIN_SCORE
```

Pour récupérer une réponse donnée dans le cas présent il faut au moins deux critères, dont `scoring_zone_id`.

[21]:
```
b_reacIn_16_29 = t4b.filter_by(score_name='reaction_In115_l3surf', scoring_zone_
↪id=(16, 29))
print(b_reacIn_16_29)
```

```
Browser object -> Number of content items: 1, data key: 'results', available metadata␣
↪keys: ['composition', 'concentration', 'energy_split_name', 'index', 'particle',
↪'reaction', 'reaction_on_nucleus', 'response_function', 'response_index', 'response_
↪name', 'response_type', 'score_index', 'score_name', 'scoring_mode', 'scoring_zone_
↪id', 'scoring_zone_type']
               -> Number of globals: 5
```

Sélection d'une réponse donnée : `select_by`

[22]:
```
reacIn_16_29 = t4b.select_by(score_name='reaction_In115_l3surf', scoring_zone_id=(16,␣
↪29))
print(type(reacIn_16_29))
print(reacIn_16_29)
```

```
<class 'dict'>
{'response_function': 'REACTION', 'response_name': 'reaction_In115', 'score_name':
↪'reaction_In115_l3surf', 'energy_split_name': 'DEC_INTEGRAL', 'particle': 'NEUTRON',
↪ 'response_type': 'score', 'reaction_on_nucleus': ('IN115_IRDF85',), 'composition':␣
↪('none',), 'concentration': (1.0,), 'reaction': ('tabulated data',), 'response_index
↪': 1, 'scoring_mode': 'SCORE_SURF', 'scoring_zone_type': 'Frontier', 'scoring_zone_
↪id': (16, 29), 'score_index': 2, 'results': {'discarded_batches': class: <class
↪'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 0.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.000000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'score': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.
↪ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.2086517,
        error: 0.010065887983318,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction', 'score/lethargy': class: <class 'valjean.eponine.
↪dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.007366562,
        error: nan,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
        name: '', what: 'reaction/lethargy', 'units': {'u': 'cm', 'v': 'unknown', 'w':
↪ 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi': 'rad', 'score': 'unknown', 'sigma
↪': '%'}, 'score_integrated': class: <class 'valjean.eponine.dataset.Dataset'>, data␣
↪type: <class 'numpy.ndarray'>
        shape: (1, 1, 1, 1, 1, 1, 1)
        value: 0.2086517,
        error: 0.010065887983318,
        bins: OrderedDict([('u', array([], dtype=float64)), ('v', array([],␣
```

(continues on next page)

(continued from previous page)

```
↪dtype=float64)), ('w', array([], dtype=float64)), ('e', array([1.e-11, 2.e+01])), (
↪'t', array([], dtype=float64)), ('mu', array([], dtype=float64)), ('phi', array([],␣
↪dtype=float64))]),
       name: '', what: 'reaction'}, 'index': 12}
```

[23]:
```
flux_14_27 = t4b.select_by(response_function='FLUX', scoring_zone_id=(14, 27))
print(type(flux_14_27))
print(list(flux_14_27['results'].keys()))
```

```
<class 'dict'>
['discarded_batches', 'used_batches', 'score', 'score/lethargy', 'units', 'score_
↪integrated']
```

Une fois le résultat voulu sélectionné on peut accéder aux données elles-mêmes.

Dans le cas de `flux_14_27` (flux surfacique entre les surfaces 14 et 27) il existe 6 types de données :

- les nombres de batches utilisés et mis de côté pour le calcul du flux
- le spectre utilisant tout le découpage en énergie (score demandé) (`score`)
- l'intégrale du spectre sur le découpage en énergie (`score_integrated`)
- le spectre score/lethargy, soit renormalisé à la largeur du bin, sur tout le découpage en énergie (`score/lethargy`)
- les unités

Tous ces résultats sont encapsulés dans un `Dataset`, à l'exception des unités qui sont dans un dictionnaire.

Étape 3 : utilisation d'un `Dataset`

Le `Dataset` permet de comparer les données à d'autres grâce aux tests ou de les représenter par exemple.

Le `Dataset` permet de stocker les données et de faire des opérations dessus (ajout de datasets, sélection de dimension, multiplication par un dataset ou une constante, etc.). Les tests statistiques proposés par `valjean` attendent également des `Dataset`. Leur argument `name` est mis par défaut à celui du fichier parsé. L'argument `what` est utilisé pour donner à un nom au type de données qu'ils contiennent. Il sera utilisé en axe des ordonnées d'un histogramme 1D par exemple. Ces deux arguments sont modifiables à tout moment.

[24]:
```
fds_flux_14_27 = flux_14_27['results']['score']
print(fds_flux_14_27)
fds_flux_14_27.name = 'flux(14, 27)'
fds_flux_14_27.what = 'Flux'
```

```
shape: (1, 1, 1, 44, 1, 1, 1), dim: 7, type: <class 'numpy.ndarray'>, bins: ['u: []',
↪'v: []', 'w: []', 'e: [ 0.0463  0.0525  0.0595  0.0674  0.0764  0.0865  0.097   0.
↪111   0.126  0.143   0.162   0.183   0.207   0.235   0.266   0.302   0.342   0.388 ␣
↪0.439   0.498   0.564   0.639   0.724   0.821   0.93    1.054   1.194  1.353   1.
↪534   1.738   1.969   2.231   2.528   2.865   3.246   3.679  4.169   4.724   5.353 ␣
↪ 6.065   6.873   7.788   8.825  10.     20.    ]', 't: []', 'mu: []', 'phi: []'],␣
↪name: , what: flux
```

Les spectres (comme les maillages) sont par défault en 7 dimensions, dont les noms sont stockés dans les bins. Le dictionnaire de bins est un `OrderedDict` qui respecte l'ordre des dimensions de l'array Numpy.

Petit commentaire sur les noms de ces bins :

- `'u'`, `'v'`, `'w'` correspondent aux 3 variables d'espace (surtout pour les maillages), ce qui peut donc être par exemple (x, y, z), (r, θ, z) ou (r, θ, ϕ)
- `'e'` l'énergie, `'t'` le temps
- `'mu'` et `'phi'` la direction de la particule

Des unités sont données par défaut (celle de Tripoli par défaut si non précisé, `unknown` pour les variables d'espace).

[25]:
```
print(flux_14_27['results']['units'])
```

```
{'u': 'cm', 'v': 'unknown', 'w': 'unknown', 'e': 'MeV', 't': 's', 'mu': '', 'phi':
↪'rad', 'score': 'unknown', 'sigma': '%'}
```

La plupart du temps beaucoup de ces dimensions ne sont pas précisées dans Tripoli-4, n'ont donc pas de découpage (ou de bins) et sont ainsi à 1 dans la shape. Pour les réduire, pour l'affichage ou l'utilisation, il est possible de *squeezer* le `Dataset`. Cette suppression des dimensions non utilisées en conseillée pour les tests.

Dans le cas de `flux_14_27` seule la dimension correspondant à l'énergie est pertinente, le *squeeze* permet de ne conserver qu'elle.

[26]:
```
ds_flux_14_27 = fds_flux_14_27.squeeze()
print(ds_flux_14_27)
```

```
shape: (44,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['e: [ 0.0463  0.0525  0.
↪0595  0.0674  0.0764  0.0865  0.097   0.111   0.126  0.143   0.162   0.183   0.207 ␣
↪ 0.235   0.266   0.302   0.342   0.388  0.439   0.498   0.564   0.639   0.724   0.
↪821   0.93    1.054   1.194  1.353   1.534   1.738   1.969   2.231   2.528   2.865 ␣
↪ 3.246   3.679  4.169   4.724   5.353   6.065   6.873   7.788   8.825  10.     20. ␣
↪  ]'], name: flux(14, 27), what: Flux
```

Le `Dataset` correspondant au résultat intégré du spectre donne quant à lui :

[27]:
```
ds_flux_14_27_int = flux_14_27['results']['score_integrated']
print(ds_flux_14_27_int)
ds_flux_14_27_int.name='flux_14_27'
ds_flux_14_27_int.what='Flux'
print(ds_flux_14_27_int.squeeze())
```

```
shape: (1, 1, 1, 1, 1, 1, 1), dim: 7, type: <class 'numpy.ndarray'>, bins: ['u: []',
↪'v: []', 'w: []', 'e: [ 0.0463 20.    ]', 't: []', 'mu: []', 'phi: []'], name: ,␣
↪what: flux
shape: (), dim: 0, type: <class 'numpy.ndarray'>, bins: [], name: flux_14_27, what:␣
↪Flux
```

Le *squeeze* supprime les bins dans deux conditions :

- il n'y a pas de bins
- il n'y a qu'un bin, donc la dimension est triviale

Remarque : pour une dimension donnée, si on a N valeurs, les bins sont

- soit donnés par leurs limites, il y a donc N+1 valeurs de bins,

- soit donnés par leurs centres, il y a donc N valeurs

Pour construire un `Dataset` on doit cependant connaître les clefs disponibles sous `results`.

Dans le cas des taux de réaction on a :

[28]:
```
print(list(reacIn_16_29['results'].keys()))
```

```
['discarded_batches', 'used_batches', 'score', 'score/lethargy', 'units', 'score_
↪integrated']
```

Dans ce cas précis, comme le découpage ne comprend qu'un seul bin, le `Dataset` issu de `'integrated'` et celui issu de `'spectrum'` devraient être les mêmes.

[29]:
```
ds_reacIn_16_29_spec = reacIn_16_29['results']['score']
ds_reacIn_16_29_spec.name='spectrum'
ds_reacIn_16_29_spec.what='Flux'
print(ds_reacIn_16_29_spec)
print(ds_reacIn_16_29_spec.value)
ds_reacIn_16_29_int = reacIn_16_29['results']['score_integrated']
ds_reacIn_16_29_int.name='integrated'
ds_reacIn_16_29_int.what='Flux'
print(ds_reacIn_16_29_int)
print(ds_reacIn_16_29_int.value)
```

```
shape: (1, 1, 1, 1, 1, 1, 1), dim: 7, type: <class 'numpy.ndarray'>, bins: ['u: []',
↪'v: []', 'w: []', 'e: [1.e-11 2.e+01]', 't: []', 'mu: []', 'phi: []'], name:␣
↪spectrum, what: Flux
[[[[[[[0.2086517]]]]]]]
shape: (1, 1, 1, 1, 1, 1, 1), dim: 7, type: <class 'numpy.ndarray'>, bins: ['u: []',
↪'v: []', 'w: []', 'e: [1.e-11 2.e+01]', 't: []', 'mu: []', 'phi: []'], name:␣
↪integrated, what: Flux
[[[[[[[0.2086517]]]]]]]
```

Étape 4 : faire un test et le représenter, exemple du test de Student

Pour avoir une description du test de Student, voir la documentation de *valjean*.

[30]:
```
from valjean.gavroche.stat_tests.student import TestStudent
```

La représentation d'un test peut se faire sous forme de tableau ou de graphique.

[31]:
```
# includes et initialisations pour les tableaux
from valjean.javert.representation import TableRepresenter
from valjean.javert.rst import RstFormatter

tabrepr = TableRepresenter()
rstformat = RstFormatter()
```

[32]:
```
# include et initialisation pour les graphiques
from valjean.javert.representation import PlotRepresenter
from valjean.javert.mpl import MplPlot

plotrepr = PlotRepresenter()
```

Comparaison de l'intégrale du spectre et du spectre dans le cas de la réaction sur In115 entre les surfaces 16 et 29

[33]:
```
stud_reacIn = TestStudent(ds_reacIn_16_29_spec, ds_reacIn_16_29_int, name='spectrum␣
↪vs intregral',
                          description="Comparaison du spectre sur 1 bin à l'intégrale
↪").evaluate()
print(type(stud_reacIn))
print(bool(stud_reacIn))
```

```
<class 'valjean.gavroche.stat_tests.student.TestResultStudent'>
True
```

[34]:
```
montab = tabrepr(stud_reacIn)  # il s'agit d'une liste de TableTemplate
monrst = rstformat.template(montab[0])
print(monrst)
```

```
Student test: OK
```

[35]:
```
monplot = plotrepr(stud_reacIn)
print(monplot)
```

```
[]
```

Les graphiques ne sont pas disponibles pour les quantités sans réels bins.

Comparaison des flux entre les surfaces 14 et 27 et les surfaces 16 et 29

Le découpage en énergie de ces deux flux étant les mêmes leur comparaison est possible.

[36]:
```
# construction du Dataset pour le flux entre les surfaces 16 et 29
flux_16_29 = t4b.select_by(response_function='FLUX', scoring_zone_id=(16, 29))
ds_flux_16_29 = flux_16_29['results']['score'].squeeze()
ds_flux_16_29.name='flux(16, 29)'
ds_flux_16_29.what='Flux'
```

[37]:
```
tstud_flux = TestStudent(ds_flux_14_27, ds_flux_16_29, name="flux_14_17_vs_16_29",
                         description="Comparison of the flux between surfaces 14 and␣
↪27 and surfaces 16 and 29")
stud_flux = tstud_flux.evaluate()
print(bool(stud_flux))
```

```
False
```

[38]:
```
montab = tabrepr(stud_flux)  # encore une liste
monrst = rstformat.template(montab[0])
print(monrst)
```

```
.. role:: hl

.. table::
    :widths: auto

    ===============  ===============  ===============  =============== ␣
```

(continues on next page)

(continued from previous page)

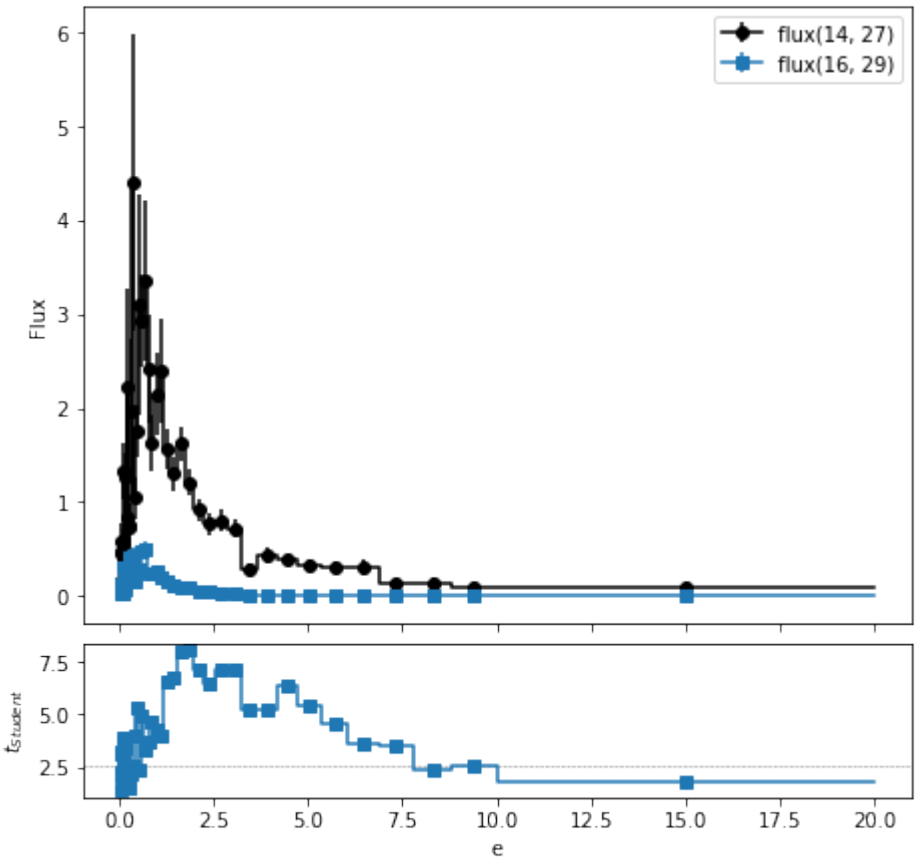
```
↪===============  ===========  ===========
          e         v(flux(14, 27))  σ(flux(14, 27))  v(flux(16, 29))  σ(flux(16,␣
↪29))       t        Student?
   ===============  ===============  ===============  =============== ␣
↪===============  ===========  ===========
   0.0595 - 0.0674         0.462393         0.123696        0.0769575        0.
↪0172087      3.08627  :hl:`False`
   0.0674 - 0.0764         0.477475         0.114182        0.0992567         0.
↪019182      3.26665  :hl:`False`
    0.0865 - 0.097         0.402269         0.117618        0.0215542       0.
↪00884732      3.22776  :hl:`False`
     0.097 - 0.111         0.584754         0.127735          0.08209         0.
↪016801      3.90159  :hl:`False`
     0.111 - 0.126         0.554189         0.147069         0.137866        0.
↪0219043      2.79992  :hl:`False`
     0.126 - 0.143          1.32995         0.294837         0.304733        0.
↪0390281      3.44717  :hl:`False`
     0.143 - 0.162          0.41841        0.0998329        0.0720989        0.
↪0181894      3.41272  :hl:`False`
     0.162 - 0.183          1.27077         0.260882          0.25939        0.
↪0328314      3.84645  :hl:`False`
     0.388 - 0.439          1.04479         0.223741          0.14842        0.
↪0348335      3.95857  :hl:`False`
     0.439 - 0.498           1.7592         0.278359         0.255463          0.
↪04875      5.32114  :hl:`False`
     0.564 - 0.639          2.93296         0.494184         0.470328        0.
↪0716812      4.93161  :hl:`False`
     0.639 - 0.724          3.35799         0.857996         0.498303         0.
↪073915      3.32069  :hl:`False`
     0.724 - 0.821          2.41004         0.576353         0.247105        0.
↪0579394      3.73397  :hl:`False`
      0.821 - 0.93          1.63467         0.301245         0.230148        0.
↪0409481       4.6199  :hl:`False`
      0.93 - 1.054          2.14932         0.432883         0.261702        0.
↪0629516      4.31519  :hl:`False`
     1.054 - 1.194          2.40261         0.553264          0.19974        0.
↪0266625      3.97697  :hl:`False`
     1.194 - 1.353          1.55762         0.213342          0.14766        0.
↪0213294      6.57613  :hl:`False`
     1.353 - 1.534          1.29884         0.173713         0.121049        0.
↪0169782      6.74793  :hl:`False`
     1.534 - 1.738          1.61908         0.190803        0.0966292        0.
↪0135499      7.95915  :hl:`False`
     1.738 - 1.969          1.20446         0.137138        0.0926817        0.
↪0108918      8.08154  :hl:`False`
     1.969 - 2.231         0.917735         0.120529        0.0554337       0.
↪00827571      7.13749  :hl:`False`
     2.231 - 2.528         0.766885         0.110739        0.0442858       0.
↪00729076      6.51118  :hl:`False`
     2.528 - 2.865         0.806095         0.107723        0.0372704       0.
↪00627467      7.12496  :hl:`False`
     2.865 - 3.246         0.711411        0.0960477        0.0232922       0.
↪00498183      7.15472  :hl:`False`
     3.246 - 3.679         0.292037         0.052866        0.0147102       0.
↪00346623      5.23461  :hl:`False`
     3.679 - 4.169         0.434942        0.0809168        0.0106427       0.
```

(continues on next page)

(continued from previous page)

```
↪00270574      5.24072  :hl:`False`
     4.169 - 4.724         0.380331        0.0576872        0.0105781        0.
↪00260966      6.40307  :hl:`False`
     4.724 - 5.353         0.327966         0.058195         0.010661         0.
↪0023411       5.44803  :hl:`False`
     5.353 - 6.065         0.312464        0.0658838        0.00989727       0.
↪00240927      4.58937  :hl:`False`
     6.065 - 6.873         0.314053        0.0848152        0.00583138       0.
↪00189545      3.63313  :hl:`False`
     6.873 - 7.788         0.128904         0.034897        0.00626957       0.
↪00196971       3.5086  :hl:`False`
   ================  ===============  ===============  ===============
↪===============  ===========  ===========
```

[39]:
```
monplot = plotrepr(stud_flux)
mpl = MplPlot(monplot[0]).draw()
```



Dans le cas d'un spectre un test de Student est effectué par bin. Dans le cas présent les spectres ne sont manifestement pas en accord, ce qui est attendu. Cependant, dans le cas d'une comparaison avec des données réellement comparables on peut s'attendre à un certain nombre de bins pour lesquels la comparaison échoue mais pas tous. Ce nombre dépend du nombre de bins.

Pour évaluer cela un autre test statistique est possible, le test de Holm-Bonferroni (voir la

documentation pour plus de précisions).

3.1.7 Parsing des sensibilités

Le jeu de données utilisé ici est sensitivity_godiva.

Comme dans le cas précédent les résultats sont stockés dans la liste des réponses. Le `Browser` simplifie l'accès à des données grâce à la possibilité de sélection sur les métadonnées.

[1]:
```
from valjean.eponine.tripoli4.parse import Parser

t4vv_sg = 'sensitivity_godiva.d.res.ceav5'
# scan du jeu de données
t4p = Parser(t4vv_sg)
# parsing du dernier batch
t4pres = t4p.parse_from_index()
# clefs disponibles dans le dictionnaire de résultats
list(t4pres.res.keys())
```

```
* Parsing sensitivity_godiva.d.res.ceav5
* Successful scan in 0.109931 s
* Successful parsing in 0.045208 s
```

[1]:
```
['list_responses', 'keff_auto', 'batch_data', 'run_data']
```

[2]:
```
lresp = t4pres.res['list_responses']
len(lresp)
```

[2]:
```
18
```

Le nombre peut être plus grand qu'attendu par la lecture du jeu de données car chaque résultat consistue une entrée dans le dictionnaire, soit chaque résultat dont la valeur d'une métadonnée varie.

[3]:
```
for i, resp in enumerate(lresp):
    print('Response {0}: clefs = {1}'.format(i, sorted(resp.keys())))
```

```
Response 0: clefs = ['keff_estimator', 'response_function', 'response_index',
↪'response_type', 'results']
Response 1: clefs = ['keff_estimator', 'response_function', 'response_index',
↪'response_type', 'results']
Response 2: clefs = ['keff_estimator', 'response_function', 'response_index',
↪'response_type', 'results']
Response 3: clefs = ['keff_estimator', 'response_function', 'response_index',
↪'response_type', 'results']
Response 4: clefs = ['keff_estimator', 'response_function', 'response_index',
↪'response_type', 'results']
Response 5: clefs = ['keff_estimator', 'response_function', 'response_index',
↪'response_type', 'results']
Response 6: clefs = ['keff_estimator', 'response_function', 'response_index',
↪'response_type', 'results']
Response 7: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 8: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 9: clefs = ['response_function', 'response_index', 'response_type', 'results
```

(continues on next page)

(continued from previous page)

```
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 10: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 11: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 12: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 13: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 14: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 15: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 16: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
Response 17: clefs = ['response_function', 'response_index', 'response_type', 'results
↪', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction', 'sensitivity_
↪type']
```

On construit donc un `Browser` pour nous faciliter la tâche.

[4]:
```
t4b = t4pres.to_browser()
print(t4b)
```

```
Browser object -> Number of content items: 22, data key: 'results', available␣
↪metadata keys: ['index', 'keff_estimator', 'response_function', 'response_index',
↪'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_reaction',
↪ 'sensitivity_type']
               -> Number of globals: 6
```

[5]:
```
for k in list(t4b.keys()):
    print("{0} -> {1}".format(k, list(t4b.available_values(k))))
```

```
response_function -> ['KEFFS', 'IFP ADJOINT WEIGHTED KEFF SENSITIVITIES']
response_type -> ['keff', 'sensitivity', 'keff_auto']
response_index -> [0, 1]
keff_estimator -> ['KSTEP', 'KCOLL', 'KTRACK', 'KSTEP-KCOLL', 'KSTEP-KTRACK', 'KCOLL-
↪KTRACK', 'full combination', 'MACRO KCOLL']
index -> [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,␣
↪21]
sensitivity_index -> [1, 2, 3]
sensitivity_nucleus -> ['U235', 'U238']
sensitivity_reaction -> ['SECTION CODE 104', 'SECTION CODE 33', 'SECTION CODE 52',
↪'PROMPT FISSION_NU', 'TOTAL FISSION_NU', 'DELAYED FISSION_NU', 'PROMPT FISSION_CHI',
↪ 'TOTAL FISSION_CHI', 'TOTAL FISSION_CHI (CONSTRAINED)', 'SCATTERING LAW 21',
↪'SCATTERING LAW 21 (CONSTRAINED)']
sensitivity_type -> ['CROSS SECTION', 'FISSION NU', 'FISSION CHI', 'SCATTERING␣
↪TRANSFER FUNCTION']
```

Exemples de sélection

Pour la « démo », mais cela reflète probablement une future démarche de développement de test, on va récupérer des `Browser` et non les réponses directement. Cela permet notamment de sélectionner la bonne réponse pas à pas.

Sélection 1 : réponses correspondant à l'U238

[6]:
```
b_u238 = t4b.filter_by(sensitivity_nucleus='U238')
["{0} -> {1}".format(k, list(b_u238.available_values(k))) for k in list(b_u238.
↪keys())]
```

[6]:
```
["response_function -> ['IFP ADJOINT WEIGHTED KEFF SENSITIVITIES']",
 "response_type -> ['sensitivity']",
 'response_index -> [1]',
 'sensitivity_index -> [3, 1, 2]',
 "sensitivity_nucleus -> ['U238']",
 "sensitivity_reaction -> ['SECTION CODE 52', 'SCATTERING LAW 21', 'SCATTERING LAW 21␣
↪(CONSTRAINED)']",
 "sensitivity_type -> ['CROSS SECTION', 'SCATTERING TRANSFER FUNCTION']",
 'index -> [0, 1, 2]']
```

Sélection 2 : réponses correspondant à des sections efficaces

[7]:
```
b_cs = t4b.filter_by(sensitivity_type='CROSS SECTION')
["{0} -> {1}".format(k, list(b_cs.available_values(k))) for k in list(b_cs.keys())]
```

[7]:
```
["response_function -> ['IFP ADJOINT WEIGHTED KEFF SENSITIVITIES']",
 "response_type -> ['sensitivity']",
 'response_index -> [1]',
 'sensitivity_index -> [1, 2, 3]',
 "sensitivity_nucleus -> ['U235', 'U238']",
 "sensitivity_reaction -> ['SECTION CODE 104', 'SECTION CODE 33', 'SECTION CODE 52']",
 "sensitivity_type -> ['CROSS SECTION']",
 'index -> [0, 1, 2]']
```

Sélection 3 : réponses correspondant au code de section efficace 52 ($n \to \gamma$ absorption)

[8]:
```
b_s42 = t4b.filter_by(sensitivity_reaction='SECTION CODE 42')
```

```
  WARNING      browser: SECTION CODE 42 is not a valid sensitivity_reaction
```

[9]:
```
b_s52 = t4b.filter_by(sensitivity_reaction='SECTION CODE 52')
["{0} -> {1}".format(k, list(b_s52.available_values(k))) for k in list(b_s52.keys())]
```

[9]:
```
["response_function -> ['IFP ADJOINT WEIGHTED KEFF SENSITIVITIES']",
 "response_type -> ['sensitivity']",
 'response_index -> [1]',
 'sensitivity_index -> [3]',
 "sensitivity_nucleus -> ['U238']",
 "sensitivity_reaction -> ['SECTION CODE 52']",
```

(continues on next page)

(continued from previous page)

```
 "sensitivity_type -> ['CROSS SECTION']",
 'index -> [0]']
```

Dans ce cas on peut récupérer directement la réponse et l'utiliser grâce à la méthode `select_by`. Elle ne fonctionne que s'il n'y a qu'une seule réponse satisfaisant la sélection.

[10]:
```
r_s52 = t4b.select_by(sensitivity_reaction='SECTION CODE 52')
list(r_s52.keys())
```

[10]:
```
['response_function',
 'response_type',
 'response_index',
 'sensitivity_index',
 'sensitivity_nucleus',
 'sensitivity_reaction',
 'sensitivity_type',
 'results',
 'index']
```

[11]:
```
list(r_s52['results'].keys())
```

[11]:
```
['score', 'units', 'integrated', 'used_batches']
```

Dans ce cas quatre résultats sont disponibles et disponibles sous forme de `Dataset`.

Nombre de batches utilisés

[12]:
```
print('nombre de batches utilisés :', r_s52['results']['used_batches'])
ubres_s52 = r_s52['results']['used_batches']
print(ubres_s52)
```

```
nombre de batches utilisés : value: 9.100000e+01, error: 0.000000e+00, bins:␣
↪OrderedDict(), type: <class 'numpy.int64'>,name: , what: sensitivity
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict(), type: <class 'numpy.
↪int64'>,name: , what: sensitivity
```

Remarque : il n'y a pas d'erreur sur le nombre de batches utilisés, le choix a été fait de l'initialiser à `np.nan`, ce qui ne bloque pas les tests. Il en est de même pour tous les résultats non affectés d'une erreur.

Spectre

[13]:
```
dssres_s52 = r_s52['results']['score']
```

[14]:
```
dssres_s52.ndim
```

[14]:
```
3
```

[15]:
```
dssres_s52.shape
```

[15]:
```
(1, 33, 1)
```

[16]:
```
dssres_s52.bins
```

[16]:
```
OrderedDict([('einc', array([], dtype=float64)),
             ('e',
              array([1.000010e-11, 1.000000e-07, 5.400000e-07, 4.000000e-06,
                     8.315287e-06, 1.370959e-05, 2.260329e-05, 4.016900e-05,
                     6.790405e-05, 9.166088e-05, 1.486254e-04, 3.043248e-04,
                     4.539993e-04, 7.485183e-04, 1.234098e-03, 2.034684e-03,
                     3.354626e-03, 5.530844e-03, 9.118820e-03, 1.503439e-02,
                     2.478752e-02, 4.086771e-02, 6.737947e-02, 1.110900e-01,
                     1.831564e-01, 3.019738e-01, 4.978707e-01, 8.208500e-01,
                     1.353353e+00, 2.231302e+00, 3.678794e+00, 6.065307e+00,
                     1.000000e+01, 1.964033e+01])),
             ('mu', array([], dtype=float64))])
```

Comme pour les spectres ou les maillages les bins des sensibilités sont stockés dans un `OrderedDict`, comme dans le cas d'un spectre habituel. Seules les coordonnées sont changées, précisées par les bins.

[17]:
```
dssres_s52.what
```

[17]:
```
'sensitivity'
```

Le `what` par défaut est `'sensitivity'`.

[18]:
```
dssres_s52.name='section 52'
print(dssres_s52)
```

```
shape: (1, 33, 1), dim: 3, type: <class 'numpy.ndarray'>, bins: ['einc: []', 'e: [1.
↪000010e-11 1.000000e-07 5.400000e-07 4.000000e-06 8.315287e-06 1.370959e-05 2.
↪260329e-05 4.016900e-05 6.790405e-05 9.166088e-05 1.486254e-04 3.043248e-04 4.
↪539993e-04 7.485183e-04 1.234098e-03 2.034684e-03 3.354626e-03 5.530844e-03 9.
↪118820e-03 1.503439e-02 2.478752e-02 4.086771e-02 6.737947e-02 1.110900e-01 1.
↪831564e-01 3.019738e-01 4.978707e-01 8.208500e-01 1.353353e+00 2.231302e+00 3.
↪678794e+00 6.065307e+00 1.000000e+01 1.964033e+01]', 'mu: []'], name: section 52,␣
↪what: sensitivity
```

Comme dans l'exemple précédent, il est possible de réduire le spectre aux seuls bins utilisés.

[19]:
```
print(dssres_s52.squeeze())
```

```
shape: (33,), dim: 1, type: <class 'numpy.ndarray'>, bins: ['e: [1.000010e-11 1.
↪000000e-07 5.400000e-07 4.000000e-06 8.315287e-06 1.370959e-05 2.260329e-05 4.
↪016900e-05 6.790405e-05 9.166088e-05 1.486254e-04 3.043248e-04 4.539993e-04 7.
↪485183e-04 1.234098e-03 2.034684e-03 3.354626e-03 5.530844e-03 9.118820e-03 1.
↪503439e-02 2.478752e-02 4.086771e-02 6.737947e-02 1.110900e-01 1.831564e-01 3.
↪019738e-01 4.978707e-01 8.208500e-01 1.353353e+00 2.231302e+00 3.678794e+00 6.
↪065307e+00 1.000000e+01 1.964033e+01]'], name: section 52, what: sensitivity
```

Résultat intégré

[20]:
```
ires_s52 = r_s52['results']['integrated']
ires_s52.name='section 52'
ires_s52.what='sensitivity'
print(ires_s52)
```

```
shape: (1, 1, 1), dim: 3, type: <class 'numpy.ndarray'>, bins: ['einc: []', 'e: [1.
↪000010e-11 1.964033e+01]', 'mu: []'], name: section 52, what: sensitivity
```

Il est également possible ici de réduire les dimensions, ce qui reviendra à ne plus avoir de bins, vu qu'il n'y en a qu'un en énergie :

[21]:
```
print(ires_s52.squeeze())
```

```
shape: (), dim: 0, type: <class 'numpy.ndarray'>, bins: [], name: section 52, what:␣
↪sensitivity
```

Sélection 4 : première réponse (0) : les k_{eff}

Deux types de résultats de k_{eff} sont disponibles : * les k_{eff} qui apparaissent comme les réponses standard dans le listing de sortie de Tripoli-4, qui comportent normalement trois évaluations : KSTEP, KCOLL et KTRACK ainsi que leurs corrélations et le résultat de leur combinaison, appelés ici k_{eff} "génériques" * les k_{eff} apparaissant le plus souvent en toute fin de listing, donc le *discard* est calculé automatiquement, de manière à en donner la meilleure estimation, appelés ici k_{eff} "automatiques"

k_{eff} "génériques"

[22]:
```
keffs = t4b.filter_by(response_index=0)
print(keffs.content)
```

```
[{'response_function': 'KEFFS', 'response_type': 'keff', 'response_index': 0, 'keff_
↪estimator': 'KSTEP', 'results': {'keff': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.float64'>
value: 9.954907e-01, error: 8.759186e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 1.000000e+00, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 0}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KCOLL', 'results': {'keff': class: <class 'valjean.
↪eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.959480e-01, error: 6.532095e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 1.000000e+00, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
```

(continues on next page)

(continued from previous page)

```
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 1}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KTRACK', 'results': {'keff': class: <class 'valjean.
↪eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.963401e-01, error: 5.638612e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 1.000000e+00, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 2}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KSTEP-KCOLL', 'results': {'keff': class: <class
↪'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.959777e-01, error: 6.522359e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 7.799494e-01, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 3}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KSTEP-KTRACK', 'results': {'keff': class: <class
↪'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.962853e-01, error: 5.619582e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 5.764322e-01, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 4}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KCOLL-KTRACK', 'results': {'keff': class: <class
↪'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.962508e-01, error: 5.546109e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 7.398244e-01, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
```

(continues on next page)

(continued from previous page)

```
}, 'index': 5}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'full combination', 'results': {'keff': class: <class
↪'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.962563e-01, error: 5.542979e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 1.000000e+00, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 6}]
```

D'autres sélections sont également possibles, sans nécessité de connaître l'index de la réponses :

[23]:
```
keffs = t4b.filter_by(response_function='KEFFS')
print(len(keffs))
print(keffs.content)
```

```
7
[{'response_function': 'KEFFS', 'response_type': 'keff', 'response_index': 0, 'keff_
↪estimator': 'KSTEP', 'results': {'keff': class: <class 'valjean.eponine.dataset.
↪Dataset'>, data type: <class 'numpy.float64'>
value: 9.954907e-01, error: 8.759186e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 1.000000e+00, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 0}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KCOLL', 'results': {'keff': class: <class 'valjean.
↪eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.959480e-01, error: 6.532095e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 1.000000e+00, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 1}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KTRACK', 'results': {'keff': class: <class 'valjean.
↪eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.963401e-01, error: 5.638612e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 1.000000e+00, error:    nan, bins: OrderedDict()
```

(continues on next page)

(continued from previous page)

```
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 2}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KSTEP-KCOLL', 'results': {'keff': class: <class
↪'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.959777e-01, error: 6.522359e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 7.799494e-01, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 3}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KSTEP-KTRACK', 'results': {'keff': class: <class
↪'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.962853e-01, error: 5.619582e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 5.764322e-01, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 4}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'KCOLL-KTRACK', 'results': {'keff': class: <class
↪'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.962508e-01, error: 5.546109e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 7.398244e-01, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
}, 'index': 5}, {'response_function': 'KEFFS', 'response_type': 'keff', 'response_
↪index': 0, 'keff_estimator': 'full combination', 'results': {'keff': class: <class
↪'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.962563e-01, error: 5.542979e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'correlation_keff': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.float64'>
value: 1.000000e+00, error:    nan, bins: OrderedDict()
name: '', what: 'correlation'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
```

(continues on next page)

(continued from previous page)

```
name: '', what: 'used_batches'
}, 'index': 6}]
```

Il y a 7 k_{eff} disponibles comme attendu (les 3 valeurs, les 3 combinaisons deux à deux et les corrélations associées et la combinaison des trois. Il est également possible de construire des datasets à partir de chacun de ces k_{eff}. Il faut cependant isoler les résultats un par un :

[24]:
```
kstep = t4b.select_by(response_function='KEFFS', keff_estimator='KSTEP')
print(list(kstep['results'].keys()))
ds_kstep = kstep['results']['keff']
ds_kstep.name='kstep'
print(ds_kstep)
```

```
['keff', 'correlation_keff', 'used_batches']
value: 9.954907e-01, error: 8.759186e-04, bins: OrderedDict(), type: <class 'numpy.
↪float64'>,name: kstep, what: keff
```

[25]:
```
kstep_kcoll = t4b.select_by(response_function='KEFFS', keff_estimator='KSTEP-KCOLL')
print(list(kstep_kcoll['results'].keys()))
print(kstep_kcoll['results']['keff'])
```

```
['keff', 'correlation_keff', 'used_batches']
value: 9.959777e-01, error: 6.522359e-04, bins: OrderedDict(), type: <class 'numpy.
↪float64'>,name: , what: keff
```

Dans ce cas la corrélation est atteignable par :

[26]:
```
print(kstep_kcoll['results']['correlation_keff'])
```

```
value: 7.799494e-01, error:    nan, bins: OrderedDict(), type: <class 'numpy.float64'>
↪,name: , what: correlation
```

Remarque : elle n'a pas d'erreur.

k_{eff} "automatiques"

[27]:
```
keffs = t4b.filter_by(response_type='keff_auto')
print(len(keffs))
print(keffs)
```

```
4
Browser object -> Number of content items: 4, data key: 'results', available metadata␣
↪keys: ['index', 'keff_estimator', 'response_type']
               -> Number of globals: 6
```

Dans ce cas il y a un autre estimateur en plus : MACRO KCOLL. À noter également la présence du nombre de batches *discarded* puisqu'il est calculé par Tripoli-4.

Sélection 5 : boucle sur toutes les réponses

Il est toujours possible de faire une boucle sur toutes les réponses, dans l'ordre dans lequel elles apparaissent dans le jeu de données (par exemple pour les besoins de la non-régression).

Premier choix : boucle directe sur la liste

[28]:
```
for resp in lresp:
    print("Response function: {0}, response type: {1}, r_index = {2}, s_index = {3}\n␣
↪results keys: {4}"
          .format(resp['response_function'], resp['response_type'], resp['response_
↪index'],
                  resp.get('sensitivity_index', None), list(resp['results'].keys())))
```

```
Response function: KEFFS, response type: keff, r_index = 0, s_index = None
 results keys: ['keff', 'correlation_keff', 'used_batches']
Response function: KEFFS, response type: keff, r_index = 0, s_index = None
 results keys: ['keff', 'correlation_keff', 'used_batches']
Response function: KEFFS, response type: keff, r_index = 0, s_index = None
 results keys: ['keff', 'correlation_keff', 'used_batches']
Response function: KEFFS, response type: keff, r_index = 0, s_index = None
 results keys: ['keff', 'correlation_keff', 'used_batches']
Response function: KEFFS, response type: keff, r_index = 0, s_index = None
 results keys: ['keff', 'correlation_keff', 'used_batches']
Response function: KEFFS, response type: keff, r_index = 0, s_index = None
 results keys: ['keff', 'correlation_keff', 'used_batches']
Response function: KEFFS, response type: keff, r_index = 0, s_index = None
 results keys: ['keff', 'correlation_keff', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 1
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 2
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 3
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 1
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 2
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 3
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 1
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 2
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 3
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
```

(continues on next page)

(continued from previous page)

```
↪sensitivity, r_index = 1, s_index = 1
 results keys: ['score', 'units', 'integrated', 'used_batches']
Response function: IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type:␣
↪sensitivity, r_index = 1, s_index = 2
 results keys: ['score', 'units', 'integrated', 'used_batches']
```

Dans ce cas les réponses sont déjà « aplaties »... mais le gros avantage est que les métadonnées sont accessibles pour chaque réponse (même les communes), il n'y a donc pas de perte d'information.

Deuxième choix : boucle grâce au `Browser` (boucle sur le 'response_index'`)

[29]:
```
val_rindex = list(t4b.available_values('response_index'))
val_rindex
```

[29]:
```
[0, 1]
```

[30]:
```
for ind in val_rindex:
    b_ind = t4b.filter_by(response_index=ind)
    print("Nombre de 'scores' par réponse =", b_ind)
    for resp in b_ind.content:
        print("Response function = {0}, sensitivity type = {1}, sensitivity index =
↪{2}"
              .format(resp['response_function'], resp.get('sensitivity_type', None),
                      resp.get('sensitivity_index', None)))
```

```
Nombre de 'scores' par réponse = Browser object -> Number of content items: 7, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_
↪function', 'response_index', 'response_type']
               -> Number of globals: 6
Response function = KEFFS, sensitivity type = None, sensitivity index = None
Response function = KEFFS, sensitivity type = None, sensitivity index = None
Response function = KEFFS, sensitivity type = None, sensitivity index = None
Response function = KEFFS, sensitivity type = None, sensitivity index = None
Response function = KEFFS, sensitivity type = None, sensitivity index = None
Response function = KEFFS, sensitivity type = None, sensitivity index = None
Response function = KEFFS, sensitivity type = None, sensitivity index = None
Nombre de 'scores' par réponse = Browser object -> Number of content items: 11, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type = CROSS␣
↪SECTION, sensitivity index = 1
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type = CROSS␣
↪SECTION, sensitivity index = 2
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type = CROSS␣
↪SECTION, sensitivity index = 3
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type =␣
↪FISSION NU, sensitivity index = 1
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type =␣
↪FISSION NU, sensitivity index = 2
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type =␣
↪FISSION NU, sensitivity index = 3
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type =␣
```

(continues on next page)

(continued from previous page)

```
↪FISSION CHI, sensitivity index = 1
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type =␣
↪FISSION CHI, sensitivity index = 2
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type =␣
↪FISSION CHI, sensitivity index = 3
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type =␣
↪SCATTERING TRANSFER FUNCTION, sensitivity index = 1
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, sensitivity type =␣
↪SCATTERING TRANSFER FUNCTION, sensitivity index = 2
```

Dans le cas présent on obtient tous les résultats formatés de la même manière dans le listing de sortie de Tripoli-4, les k_{eff} "automatiques" n'y sont donc pas. Ils peuvent cependant être également obtenus dans la boucle, à condition de boucler sur `'index'` au lieu de `'response_index'` :

[31]:
```
val_index = list(t4b.available_values('index'))
print('index :', val_index)
for ind in val_index:
    b_ind = t4b.filter_by(index=ind)
    print("Nombre de 'scores' par réponse =", b_ind)
    for resp in b_ind.content:
        print("Response function = {0}, response type = {1}, sensitivity type = {2},␣
↪sensitivity index = {3}"
              .format(resp.get('response_function', None), resp['response_type'],
                      resp.get('sensitivity_type', None), resp.get('sensitivity_index
↪', None)))
        if resp.get('response_function') is None:
            print(resp)
```

```
index : [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21]
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_
↪function', 'response_index', 'response_type']
               -> Number of globals: 6
Response function = KEFFS, response type = keff, sensitivity type = None, sensitivity␣
↪index = None
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_
↪function', 'response_index', 'response_type']
               -> Number of globals: 6
Response function = KEFFS, response type = keff, sensitivity type = None, sensitivity␣
↪index = None
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_
↪function', 'response_index', 'response_type']
               -> Number of globals: 6
Response function = KEFFS, response type = keff, sensitivity type = None, sensitivity␣
↪index = None
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_
↪function', 'response_index', 'response_type']
               -> Number of globals: 6
Response function = KEFFS, response type = keff, sensitivity type = None, sensitivity␣
↪index = None
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_
↪function', 'response_index', 'response_type']
```

(continues on next page)

(continued from previous page)

```
               -> Number of globals: 6
Response function = KEFFS, response type = keff, sensitivity type = None, sensitivity␣
↪index = None
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_
↪function', 'response_index', 'response_type']
               -> Number of globals: 6
Response function = KEFFS, response type = keff, sensitivity type = None, sensitivity␣
↪index = None
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_
↪function', 'response_index', 'response_type']
               -> Number of globals: 6
Response function = KEFFS, response type = keff, sensitivity type = None, sensitivity␣
↪index = None
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = CROSS SECTION, sensitivity index = 1
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = CROSS SECTION, sensitivity index = 2
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = CROSS SECTION, sensitivity index = 3
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = FISSION NU, sensitivity index = 1
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = FISSION NU, sensitivity index = 2
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
```

(continues on next page)

(continued from previous page)

```
↪sensitivity, sensitivity type = FISSION NU, sensitivity index = 3
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = FISSION CHI, sensitivity index = 1
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = FISSION CHI, sensitivity index = 2
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = FISSION CHI, sensitivity index = 3
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = SCATTERING TRANSFER FUNCTION, sensitivity index = 1
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'response_function', 'response_
↪index', 'response_type', 'sensitivity_index', 'sensitivity_nucleus', 'sensitivity_
↪reaction', 'sensitivity_type']
               -> Number of globals: 6
Response function = IFP ADJOINT WEIGHTED KEFF SENSITIVITIES, response type =␣
↪sensitivity, sensitivity type = SCATTERING TRANSFER FUNCTION, sensitivity index = 2
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_type
↪']
               -> Number of globals: 6
Response function = None, response type = keff_auto, sensitivity type = None,␣
↪sensitivity index = None
{'response_type': 'keff_auto', 'keff_estimator': 'KSTEP', 'results': {'keff': class:
↪<class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.956131e-01, error: 8.121986e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.150000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'discarded_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.int64'>
value: 5.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
}, 'index': 0}
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
```

(continues on next page)

(continued from previous page)

```
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_type
↪']
               -> Number of globals: 6
Response function = None, response type = keff_auto, sensitivity type = None,␣
↪sensitivity index = None
{'response_type': 'keff_auto', 'keff_estimator': 'KCOLL', 'results': {'keff': class:
↪<class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.957839e-01, error: 6.040021e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.140000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'discarded_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.int64'>
value: 6.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
}, 'index': 0}
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_type
↪']
               -> Number of globals: 6
Response function = None, response type = keff_auto, sensitivity type = None,␣
↪sensitivity index = None
{'response_type': 'keff_auto', 'keff_estimator': 'KTRACK', 'results': {'keff': class:
↪<class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.961751e-01, error: 4.986042e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.150000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'discarded_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.int64'>
value: 5.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
}, 'index': 0}
Nombre de 'scores' par réponse = Browser object -> Number of content items: 1, data␣
↪key: 'results', available metadata keys: ['index', 'keff_estimator', 'response_type
↪']
               -> Number of globals: 6
Response function = None, response type = keff_auto, sensitivity type = None,␣
↪sensitivity index = None
{'response_type': 'keff_auto', 'keff_estimator': 'MACRO KCOLL', 'results': {'keff':␣
↪class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.957877e-01, error: 5.886514e-04, bins: OrderedDict()
name: '', what: 'keff'
, 'used_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class
↪'numpy.int64'>
value: 1.150000e+02, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'used_batches'
, 'discarded_batches': class: <class 'valjean.eponine.dataset.Dataset'>, data type:
↪<class 'numpy.int64'>
value: 5.000000e+00, error: 0.000000e+00, bins: OrderedDict()
name: '', what: 'discarded_batches'
}, 'index': 0}
```

Exemples de comparaison de k_{eff}

On va faire différents tests pour comparer les valeurs de k_{eff} :

- `TestEqual` qui vérifie que les datasets sont égaux (ce test est plutôt prévu pour des valeurs entières comme les nombres de batches)
- `TestApproxEqual` qui vérifie que les datasets sont approximativement égaux (pertinent pour les `float` pour lesquels on n'a pas d'erreur associée, les corrélations de k_{eff} par exemple
- `TestStudent` dans le cas où l'on veut prendre en compte les erreurs sur les valeurs

Pour tous ces tests il faut définir une référence, qui sera en fait le premier dataset donné.

Pour les exemples ci-dessous on choisit les k_{eff} des réponses (`response_function='KEFFS'`). On prendra comme référence le `KSTEP` pour plus de facilités car c'est le premier résultat donné.

[32]:
```
sb = t4b.filter_by(response_function='KEFFS')
print('estimators values:', list(sb.available_values('keff_estimator')))
dsets = []
for keff in sb.content:
    dsets.append(keff['results']['keff'])
    dsets[-1].name=keff['keff_estimator']
print(dsets)
```

```
estimators values: ['KSTEP', 'KCOLL', 'KTRACK', 'KSTEP-KCOLL', 'KSTEP-KTRACK', 'KCOLL-
↪KTRACK', 'full combination']
[class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.954907e-01, error: 8.759186e-04, bins: OrderedDict()
name: 'KSTEP', what: 'keff'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.959480e-01, error: 6.532095e-04, bins: OrderedDict()
name: 'KCOLL', what: 'keff'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.963401e-01, error: 5.638612e-04, bins: OrderedDict()
name: 'KTRACK', what: 'keff'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.959777e-01, error: 6.522359e-04, bins: OrderedDict()
name: 'KSTEP-KCOLL', what: 'keff'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.962853e-01, error: 5.619582e-04, bins: OrderedDict()
name: 'KSTEP-KTRACK', what: 'keff'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.962508e-01, error: 5.546109e-04, bins: OrderedDict()
name: 'KCOLL-KTRACK', what: 'keff'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.float64'>
value: 9.962563e-01, error: 5.542979e-04, bins: OrderedDict()
name: 'full combination', what: 'keff'
]
```

On importe les tests et la possibilité d'en faire des représentations.

[33]:
```
from valjean.gavroche.test import TestEqual, TestApproxEqual
from valjean.gavroche.stat_tests.student import TestStudent
from valjean.javert.representation import FullRepresenter
from valjean.javert.rst import RstFormatter
from valjean.javert.mpl import MplPlot
```

(continues on next page)

(continued from previous page)

```
from valjean.javert.verbosity import Verbosity

frepr = FullRepresenter()
rstformat = RstFormatter()
```

[34]:
```
teq_res = TestEqual(*dsets, name='TestEqual', description='Test le TestEqual sur les␣
↪keff').evaluate()
print(bool(teq_res))  # expected: False
```

```
False
```

[35]:
```
eqrepr = frepr(teq_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une liste de␣
↪templates
print(eqrepr, len(eqrepr))
eqrst = rstformat.template(eqrepr[0])
print(eqrst)
```

```
[class: <class 'valjean.javert.templates.TableTemplate'>
headers: ['KSTEP', 'KCOLL', 'equal(KCOLL)?', 'KTRACK', 'equal(KTRACK)?', 'KSTEP-KCOLL
↪', 'equal(KSTEP-KCOLL)?', 'KSTEP-KTRACK', 'equal(KSTEP-KTRACK)?', 'KCOLL-KTRACK',
↪'equal(KCOLL-KTRACK)?', 'full combination', 'equal(full combination)?']
KSTEP: 0.9954907
KCOLL: 0.995948
equal(KCOLL)?: False
KTRACK: 0.9963401
equal(KTRACK)?: False
KSTEP-KCOLL: 0.9959777
equal(KSTEP-KCOLL)?: False
KSTEP-KTRACK: 0.9962853
equal(KSTEP-KTRACK)?: False
KCOLL-KTRACK: 0.9962508
equal(KCOLL-KTRACK)?: False
full combination: 0.9962563
equal(full combination)?: False
highlights: [array(False), array(False), True, array(False), True, array(False), True,
↪ array(False), True, array(False), True, array(False), True]] 1
.. role:: hl

.. table::
    :widths: auto

    ===========  ===========  =============  ===========  ==============  ===========␣
↪ ===================  ============  ====================  ============ ␣
↪====================  ================  ========================
       KSTEP        KCOLL     equal(KCOLL)?    KTRACK     equal(KTRACK)?  KSTEP-KCOLL␣
↪ equal(KSTEP-KCOLL)?  KSTEP-KTRACK  equal(KSTEP-KTRACK)?  KCOLL-KTRACK  equal(KCOLL-
↪KTRACK)?  full combination  equal(full combination)?
    ===========  ===========  =============  ===========  ==============  ===========␣
↪ ===================  ============  ====================  ============ ␣
↪====================  ================  ========================
       0.995491     0.995948    :hl:`False`      0.99634     :hl:`False`     0.995978␣
↪          :hl:`False`      0.996285           :hl:`False`      0.996251           :
↪hl:`False`          0.996256               :hl:`False`
    ===========  ===========  =============  ===========  ==============  ===========␣
↪ ===================  ============  ====================  ============ ␣
↪====================  ================  ========================
```

(continues on next page)

(continued from previous page)

[36]:
```
tstud_res = TestStudent(*dsets, name='TestStudent', description='Test le TestStudent␣
↪sur les keff').evaluate()
print(bool(tstud_res))
```

```
True
```

[37]:
```
studrepr = frepr(tstud_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une␣
↪liste de templates
print(len(studrepr), [type(t) for t in studrepr])
studrst = rstformat.template(studrepr[0])
print(studrst)
```

```
1 [<class 'valjean.javert.templates.TableTemplate'>]
.. role:: hl

.. table::
    :widths: auto

    ===========  ===========  ===========  ===========  ===========  ===============  ␣
↪===========  ===========  ===========  ===========  ================  ==============  ␣
↪==============  ==============  ======================  ===============  ␣
↪===============  ===============  ======================  ===============  ␣
↪===============  ===============  ======================  ====================  ␣
↪====================  ====================  ============================
     v(KSTEP)     σ(KSTEP)     v(KCOLL)     σ(KCOLL)     t(KCOLL)    Student(KCOLL)?  ␣
↪ v(KTRACK)    σ(KTRACK)    t(KTRACK)   Student(KTRACK)?  v(KSTEP-KCOLL)  σ(KSTEP-
↪KCOLL)  t(KSTEP-KCOLL)  Student(KSTEP-KCOLL)?  v(KSTEP-KTRACK)  σ(KSTEP-KTRACK)  ␣
↪t(KSTEP-KTRACK)  Student(KSTEP-KTRACK)?  v(KCOLL-KTRACK)  σ(KCOLL-KTRACK)  t(KCOLL-
↪KTRACK)  Student(KCOLL-KTRACK)?  v(full combination)  σ(full combination)  t(full␣
↪combination)  Student(full combination)?
    ===========  ===========  ===========  ===========  ===========  ===============  ␣
↪===========  ===========  ===========  ===========  ================  ==============  ␣
↪==============  ==============  ======================  ===============  ␣
↪===============  ===============  ======================  ===============  ␣
↪===============  ===============  ======================  ====================  ␣
↪====================  ====================  ============================
       0.995491  0.000875919     0.995948   0.00065321    -0.418518             True  ␣
↪    0.99634  0.000563861    -0.815385              True        0.995978     0.
↪000652236       -0.445937                   True         0.996285      0.000561958  ␣
↪      -0.763534                    True         0.996251      0.000554611        -0.
↪733165                   True             0.996256          0.000554298           ␣
↪-0.738589                         True
    ===========  ===========  ===========  ===========  ===========  ===============  ␣
↪===========  ===========  ===========  ===========  ================  ==============  ␣
↪==============  ==============  ======================  ===============  ␣
↪===============  ===============  ======================  ===============  ␣
↪===============  ===============  ======================  ====================  ␣
↪====================  ====================  ============================
```

Il est possible d'avoir une représentation de ce test plus lisible en créant deux `Dataset` : l'un contenant tous ces résultats, l'autre, qui servira de référence, le KSTEP autant de fois que d'éléments dans le premier. Dans ce cas la représentation permettra également d'avoir une représentation graphique.

Pour les bins, le plus simple est de donner les noms des estimateurs, actuellement stocké comme `'name'`.

Cette forme permet également de comparer ces mêmes valeurs pour différentes versions de Tripoli-4 par exemple.

[38]:
```
from collections import OrderedDict
import numpy as np
from valjean.eponine.dataset import Dataset
```

[39]:
```
dset_test = Dataset(value=np.array([k.value for k in dsets]),
                    error=np.array([k.error for k in dsets]),
                    bins=OrderedDict([('estimator', np.array([k.name for k in␣
↪dsets]))]),
                    name='Test', what='keff')
print(dset_test)
```

```
shape: (7,), dim: 1, type: <class 'numpy.ndarray'>, bins: ["estimator: ['KSTEP' 'KCOLL
↪' 'KTRACK' 'KSTEP-KCOLL' 'KSTEP-KTRACK' 'KCOLL-KTRACK' 'full combination']"], name:␣
↪Test, what: keff
```

On fait maintenant le `Dataset` de référence.

ATTENTTION : pour pouvoir comparer des ``Dataset`` il faut qu'ils aient les mêmes bins. La référence (qui ne contient que KSTEP) doit donc avoir tous les estimateurs dans les bins (les valeurs étant bien sûr celles du KSTEP).

[40]:
```
dset_ref = Dataset(value=np.array([ds_kstep.value for _ in dsets]),
                   error=np.array([ds_kstep.error for _ in dsets]),
                   bins=OrderedDict([('estimator', np.array([k.name for k in␣
↪dsets]))]),
                   name='Ref (KSTEP)', what='keff')
print(dset_ref)
```

```
shape: (7,), dim: 1, type: <class 'numpy.ndarray'>, bins: ["estimator: ['KSTEP' 'KCOLL
↪' 'KTRACK' 'KSTEP-KCOLL' 'KSTEP-KTRACK' 'KCOLL-KTRACK' 'full combination']"], name:␣
↪Ref (KSTEP), what: keff
```

Représentation avec le `TestStudent`

[41]:
```
tstudRT_res = TestStudent(dset_ref, dset_test, name='TestStudent', description='Test␣
↪le TestStudent sur les keff').evaluate()
print(bool(tstudRT_res))
```

```
True
```

[42]:
```
studRTtemp = frepr(tstudRT_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une␣
↪liste de templates
print(len(studRTtemp), [type(t) for t in studRTtemp])
studRTrst = rstformat.template(studRTtemp[1])
print(studRTrst)
mpl = MplPlot(studRTtemp[0]).draw()
```

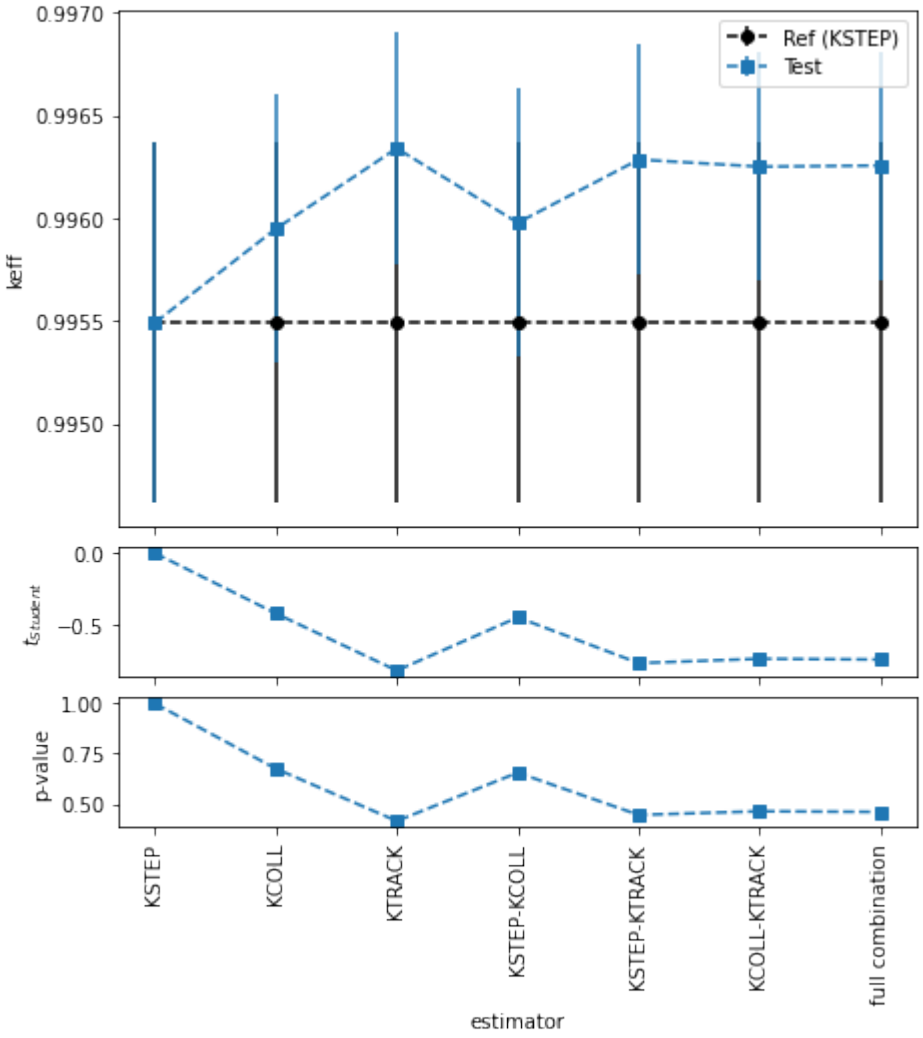
```
2 [<class 'valjean.javert.templates.PlotTemplate'>, <class 'valjean.javert.templates.
↪TableTemplate'>]
.. role:: hl
```

(continues on next page)

(continued from previous page)

```
.. table::
    :widths: auto

    ================  ==============  ==============  ==========  ========== ␣
↪==========  ========
      estimator      v(Ref (KSTEP))  σ(Ref (KSTEP))    v(Test)      σ(Test)        ␣
↪t       Student?
    ================  ==============  ==============  ==========  ========== ␣
↪==========  ========
              KSTEP        0.995491     0.000875919     0.995491  0.000875919        ␣
↪    0      True
              KCOLL        0.995491     0.000875919     0.995948   0.00065321    -0.
↪418518      True
              KTRACK       0.995491     0.000875919      0.99634  0.000563861    -0.
↪815385      True
        KSTEP-KCOLL        0.995491     0.000875919     0.995978  0.000652236    -0.
↪445937      True
       KSTEP-KTRACK        0.995491     0.000875919     0.996285  0.000561958    -0.
↪763534      True
       KCOLL-KTRACK        0.995491     0.000875919     0.996251  0.000554611    -0.
↪733165      True
    full combination       0.995491     0.000875919     0.996256  0.000554298    -0.
↪738589      True
    ================  ==============  ==============  ==========  ========== ␣
↪==========  ========
```



Représentation avec le `TestApproxEqual`

[43]:
```
dscorr = []
for keff in sb.content:
    dscorr.append(keff['results']['correlation_keff'])
    dscorr[-1].name=keff['keff_estimator']
dscorr_test = Dataset(value=np.array([k.value for k in dscorr]),
                      error=np.array([k.error for k in dscorr]),
                      bins=OrderedDict([('estimator', np.array([k.name for k in␣
↪dscorr]))]),
                      name='Test', what='correlation')
dscorr_kstep = kstep['results']['correlation_keff']
dscorr_kstep.name=kstep['keff_estimator']
dscorr_ref = Dataset(value=np.array([dscorr_kstep.value for _ in dscorr]),
                     error=np.array([dscorr_kstep.error for _ in dscorr]),
                     bins=OrderedDict([('estimator', np.array([k.name for k in␣
↪dscorr]))]),
                     name='Ref (KSTEP)', what='correlation')
```

(continues on next page)

(continued from previous page)

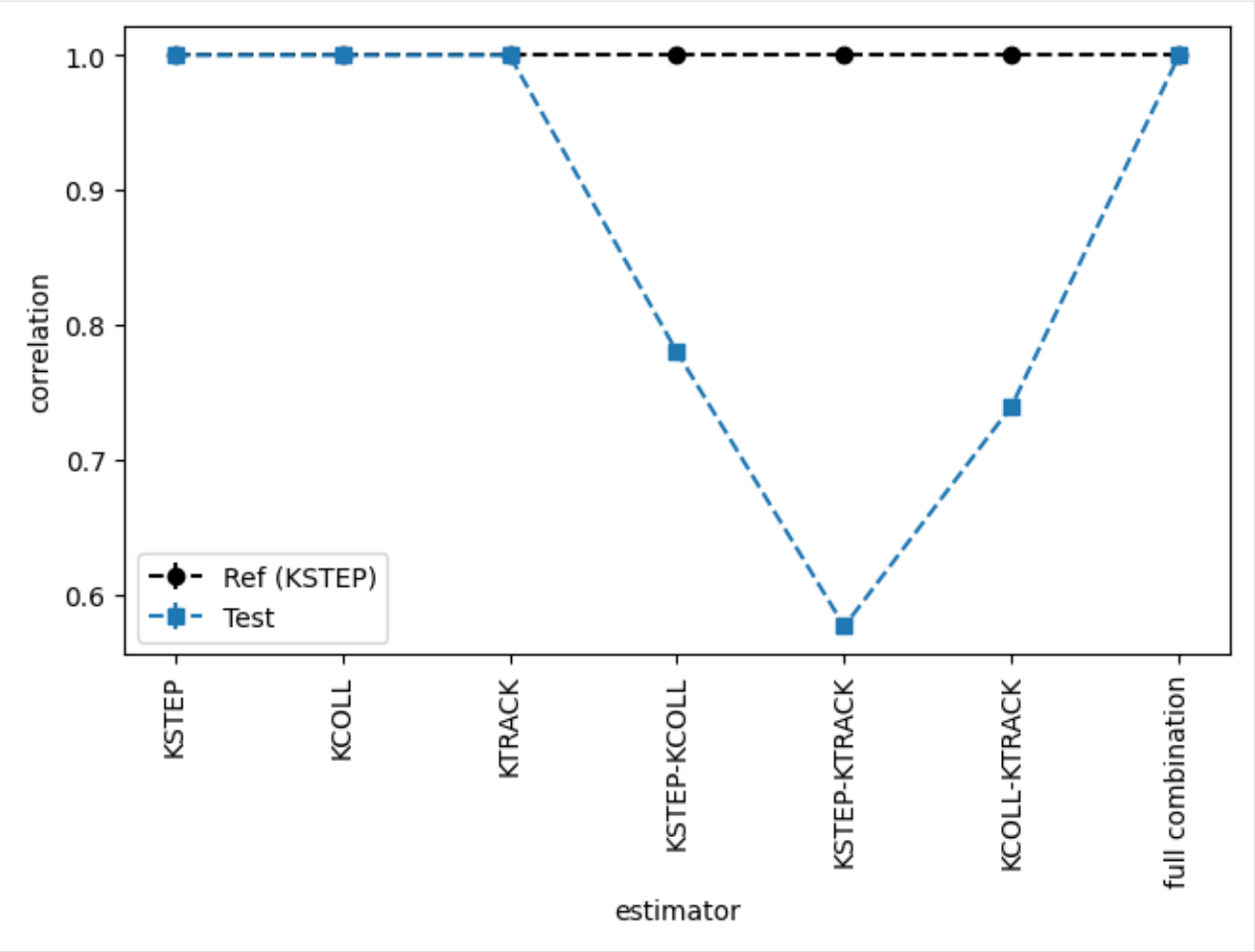
```
taeqRT_res = TestApproxEqual(
    dscorr_ref, dscorr_test, name='TestApproxEqual',
    description='Test le TestApproxEqual sur les correlations entre estimations de␣
↪keff').evaluate()
print(bool(taeqRT_res))
aeqRTtemp = frepr(taeqRT_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une␣
↪liste de templates
print(len(aeqRTtemp), [type(t) for t in aeqRTtemp])
aeqRTrst = rstformat.template(aeqRTtemp[1])
print(aeqRTrst)
mpl = MplPlot(aeqRTtemp[0]).draw()
```

```
False
2 [<class 'valjean.javert.templates.PlotTemplate'>, <class 'valjean.javert.templates.
↪TableTemplate'>]
.. role:: hl

.. table::
    :widths: auto

    ================  ===========  ===========  =============
       estimator      Ref (KSTEP)     Test      approx equal?
    ================  ===========  ===========  =============
               KSTEP            1            1           True
               KCOLL            1            1           True
              KTRACK            1            1           True
         KSTEP-KCOLL            1     0.779949    :hl:`False`
        KSTEP-KTRACK            1     0.576432    :hl:`False`
        KCOLL-KTRACK            1     0.739824    :hl:`False`
    full combination            1            1           True
    ================  ===========  ===========  =============
```

```
/home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/python3.8/
↪site-packages/matplotlib/axes/_base.py:2532: UserWarning: Warning: converting a␣
↪masked element to nan.
  xys = np.asarray(xys)
/home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/python3.8/
↪site-packages/matplotlib/axes/_base.py:2532: UserWarning: Warning: converting a␣
↪masked element to nan.
  xys = np.asarray(xys)
```



Représentation avec TestEqual

[44]:
```
dsub = []
for keff in sb.content:
    dsub.append(keff['results']['used_batches'])
    dsub[-1].name=keff['keff_estimator']  #, what='used batches'))
dsub_test = Dataset(value=np.array([k.value for k in dsub]),
                    error=np.array([k.error for k in dsub]),
                    bins=OrderedDict([('estimator', np.array([k.name for k in␣
↪dsub]))]),
                    name='Test', what='used batches')
dsub_kstep = kstep['results']['used_batches']
dsub_kstep.name=kstep['keff_estimator']  # , what='used batches')
dsub_ref = Dataset(value=np.array([dsub_kstep.value for _ in dsub]),
                   error=np.array([dsub_kstep.error for _ in dsub]),
                   bins=OrderedDict([('estimator', np.array([k.name for k in␣
↪dsub]))]),
                   name='Ref (KSTEP)', what='used batches')
print(dsub)
teqRT_res = TestEqual(
    dsub_ref, dsub_test, name='TestEqual',
    description='Test le TestEqual sur les batches utilisés pour les estimations de␣
```

(continues on next page)

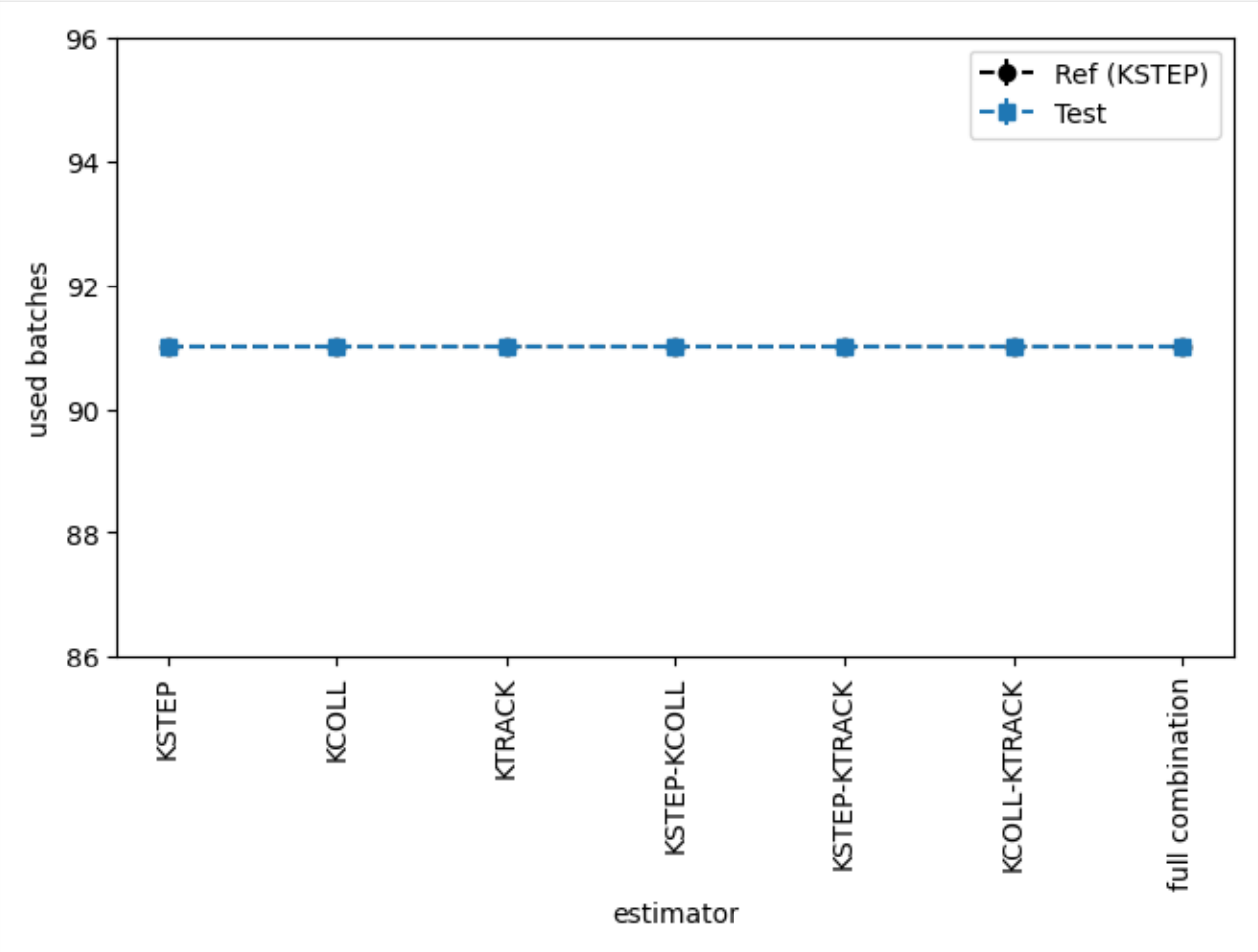
(continued from previous page)

```
↪keff').evaluate()
print(bool(teqRT_res))
eqRTtemp = frepr(teqRT_res, verbosity=Verbosity.FULL_DETAILS)  # il s'agit d'une␣
↪liste de templates
print(len(eqRTtemp), [type(t) for t in eqRTtemp])
eqRTrst = rstformat.template(eqRTtemp[1])
print(eqRTrst)
mpl = MplPlot(eqRTtemp[0]).draw()
```

```
[class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: 'KSTEP', what: 'used_batches'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: 'KCOLL', what: 'used_batches'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: 'KTRACK', what: 'used_batches'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: 'KSTEP-KCOLL', what: 'used_batches'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: 'KSTEP-KTRACK', what: 'used_batches'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: 'KCOLL-KTRACK', what: 'used_batches'
, class: <class 'valjean.eponine.dataset.Dataset'>, data type: <class 'numpy.int64'>
value: 9.100000e+01, error: 0.000000e+00, bins: OrderedDict()
name: 'full combination', what: 'used_batches'
]
True
2 [<class 'valjean.javert.templates.PlotTemplate'>, <class 'valjean.javert.templates.
↪TableTemplate'>]
.. role:: hl

.. table::
    :widths: auto

    ================  ===========  ====  ======
       estimator      Ref (KSTEP)  Test  equal?
    ================  ===========  ====  ======
               KSTEP           91    91    True
               KCOLL           91    91    True
              KTRACK           91    91    True
         KSTEP-KCOLL           91    91    True
        KSTEP-KTRACK           91    91    True
        KCOLL-KTRACK           91    91    True
    full combination           91    91    True
    ================  ===========  ====  ======
```



Si on veut comparer les résultats à d'autres obtenus à partir d'une autre version de Tripoli-4 par exemple, on peut faire un dataset unique et le mettre dans un test.

3.2 Jobs

3.2.1 Vaches sphériques

Cet exemple présente une implémentation simplifiée d'un job `valjean` pour la comparaison systématique de TRIPOLI-4® et MCNP sur des systèmes très simples : des sphères contenant un seul isotope et une source de neutrons monoénergétiques placée au centre. Nous calculons le flux neutron intégré sur la sphère et découpé à 616 groupes.

Note : cet exemple est censé être exécuté par l'exécutable `valjean`.

Fichier `job.py`

Imports

```
import os
from pathlib import Path

from valjean.cosette.run import RunTaskFactory
from valjean.cosette.task import TaskStatus
from valjean.cosette.pythontask import PythonTask
from valjean.eponine.tripoli4.parse import Parser
from valjean.eponine.tripoli4.data_convertor import convert_data
from valjean.gavroche.stat_tests.student import TestStudent
from valjean.gavroche.stat_tests.bonferroni import TestHolmBonferroni
from valjean.javert.test_report import TestReport
from valjean.javert.representation import Representation, FullRepresenter
from valjean.javert.rst import RstTestReportTask
import mcnp  # module auxiliaire de lecture de fichiers MCTAL, pas inclus dans valjean
```

Définitions des chemins et des paramètres

```
JOB_PATH = Path(__file__)

# paramètres pour TRIPOLI-4
T4_EXE = '/path/to/bin/tripoli4'
T4_PATH = '/path/to/t4path'

# paramètres pour MCNP
MCNP_EXE = '/path/to/bin/mcnp6'
MCNP_DATAPATH = '/path/to/MCNP_DATA'
MCNP_XSDIR = '/path/to/MCNP/xsdir'
os.environ['DATAPATH'] = MCNP_DATAPATH

TABPROB = True        # utiliser les tables de probabilités ?
N_HISTORIES = 100000  # nombre d'histoires à simuler
```

Systèmes à tester

On définit ici une liste de nucléides à étudier.

```
#              ┌──────────────────────────────────────── noyau
#              │          ┌──────────────────────────── nom noyau TRIPOLI-4
#              │          │       ┌──────────────────── code noyau MCNP (ZZZAAA)
#              │          │       │     ┌────────────── énergie (MeV)
#              │          │       │     │    ┌──────── température (K)
#              │          │       │     │    │      ┌─ concentration en 1/(b cm²)
#              v          v       v     v    v      v
SYSTEMS = [('O16',     'O16',  8016, 14.0, 294, 1.527e-1),
           ('U235',   'U235', 92235, 14.0, 294, 4.549e-3),
           ('Pu239', 'PU239', 94239, 14.0, 294, 4.590e-3)]
```

Définition de la fonction `job()`

Cette fonction définit le travail à exécuter par `valjean`. On va déclarer les tâches et les dépendances entre elles, et `valjean` va s'occuper du lancement. Il existe des types de tâches différentes.

```
def job():
    # On crée une classe utilitaire pour générer des tâches qui
    # exécutent TRIPOLI-4. Les options de la ligne de commande T4
    # sont paramétrées par des mot-clés.
    t4_args = ['-s', '{fmt}', '-d', '{input_file}', '-c', '{t4path}',
               '-l', '{language}', '-a', '-u']
    t4_fac = RunTaskFactory.from_executable(T4_EXE, name='t4',
                                            default_args=t4_args,
                                            t4path=T4_PATH,
                                            language='english')

    # On fait la même chose pour MCNP.
    mcnp_args = ['i={input_file}', 'xs={xsdir}', 'o={output_file}']
    mcnp_fac = RunTaskFactory.from_executable(MCNP_EXE,
                                              name='mcnp',
                                              default_args=mcnp_args,
                                              xsdir=MCNP_XSDIR)

    # Voici la liste des tâches de comparaison TRIPOLI-4 vs. MCNP
    # à exécuter. La fonction make_comparison_task() apparaît
    # ci-dessous.
    tasks = [make_comparison_task(system, t4_fac, mcnp_fac)
             for system in SYSTEMS]

    # Les tâches de comparaison vont générer des résultat de tests statistiques.
    # On va les transformer en rapport HTML, via un format intermédiaire
    # (reStructuredText, une sorte de Markdown).
    my_repr = Representation(FullRepresenter())
    # La construction du rapport de test est aussi une tâche valjean ! La seule
    # ligne mystérieuse est...
    report_task = RstTestReportTask.from_tasks(name='report',
                                               make_report=make_report, # celle-ci
                                               eval_tasks=tasks,
                                               representation=my_repr,
                                               author='Davide Mancusi',
                                               version='0.0.1',
                                               kwargs={'title': 'Spherical cows'})
    # Ici make_report est une fonction qui décrit comment organiser les résultats
    # de test en sections. Dans notre cas, elle va être très simple.

    # On renvoie uniquement la tâche de construction du rapport. Elle dépend
    # implicitement des tâches de comparaison, qui dependent des tâches d'exécution
    # de TRIPOLI-4 et MCNP...
    return [report_task]
```

Comparaison TRIPOLI-4/MCNP pour un système donné

Cette fonction prépare les jeux de données TRIPOLI-4 et MCNP en remplissant les cases vides de fichiers templates (`TRIPOLI-4`, `MCNP`).

```
def make_comparison_task(system, t4_fac, mcnp_fac):

    # On déballe le tuple qui décrit le système
    nucleus, t4_nucleus, zaid, energy, temperature, conc = system
    nuc_id = str(zaid)

    input_path = JOB_PATH.with_name('input') / f'{nucleus}_{energy}'
    input_path.mkdir(parents=True, exist_ok=True)

    # On crée le jeu de données pour TRIPOLI-4 à partir des paramètres du
    # système et d'un jeu de données "template".
    t4_input_path = input_path / 't4'
    format_t4_input(t4_input_path, nucleus=t4_nucleus, concentration=conc,
                    n_histories=N_HISTORIES, temperature=temperature,
                    energy=energy)

    # On fait la même chose pour MCNP
    mcnp_input_path = input_path / 'mcnp'
    mcnp_output_path = 'mcnp.out'
    format_mcnp_input(mcnp_input_path, nucleus=nucleus, concentration=conc,
                      nuc_id=nuc_id, tabprob=TABPROB,
                      n_histories=N_HISTORIES, temperature=temperature,
                      energy=energy)

    # Les fonctions format_t4_input() et format_mcnp_input() apparaîssent
    # plus bas.

    # Voici les tâches qui vont exécuter TRIPOLI-4 et MCNP
    t4_fmt = 'TABPROB' if TABPROB else 'NJOY'
    t4_task = t4_fac.make(input_file=str(t4_input_path), fmt=t4_fmt,
                          name=f'{nucleus}@{energy}')
    mcnp_task = mcnp_fac.make(input_file=str(mcnp_input_path),
                              output_file=str(mcnp_output_path),
                              name=f'{nucleus}@{energy}')

    # Et voici la tâche de comparaison entre le résultat de TRIPOLI-4 et
    # celui de MCNP. La tâche consiste à appeler une fonction appelée
    # compare(), qui apparaitra dans la suite
    task_name = f'comparison {nucleus}_{energy}'
    compare_task = PythonTask(task_name,
                              compare,  # voici la fonction à appeler
                              env_kwarg='env',
                              # voici les arguments à passer à compare()
                              kwargs={'nucleus': nucleus,
                                      'energy': float(energy),
                                      't4_name': t4_task.name,
                                      'task_name': task_name,
                                      'mcnp_name': mcnp_task.name},
                              # on déclare que cette tâche dépend de t4_task et mcnp_
↪task
                              deps=[t4_task, mcnp_task])
    return compare_task
```

Que fait la fonction de comparaison? Elle lit le listing de sortie TRIPOLI-4 et celui de MCNP, extrait les flux neutron multigroupe et fait un test statistique de compatibilité entre les deux.

```
def compare(*, t4_name, mcnp_name, task_name, nucleus, energy, env):
    mcnp_mctal = Path(env[mcnp_name]['result']).with_name('mctal')
    mcnp_ds = mcnp.MCTALResult(str(mcnp_mctal)).result(4, 1)
    mcnp_ds.name = mcnp_name

    t4_out = Path(env[t4_name]['result'])
    t4_br = Parser(t4_out).parse_from_index().to_browser()
    t4_res = t4_br.select_by(score_name='flux_score')['results']
    t4_ds = (convert_data(t4_res, 'spectrum', name=t4_name, what='flux')
             .squeeze())

    test = TestHolmBonferroni(test=TestStudent(t4_ds, mcnp_ds, name=nucleus),
                              name=f'Holm-Bonferroni test, {nucleus} '
                                   f'@ {energy} MeV',
                              labels={'nucleus': nucleus,
                                      'energy': energy})
    test_result = test.evaluate()
    return {task_name: {'result': [test_result]}}, TaskStatus.DONE
```

Fonctions auxiliaires

Le reste, c'est de la bureaucratie. La fonction qui génère le jeu de données TRIPOLI-4 ne fait que lire le template et remplir les cases vides.

```
def format_t4_input(output_path, *, n_batch=200, temperature, n_histories,
                    **kwargs):
    template_path = JOB_PATH.with_name('templates') / 't4'
    template = template_path.read_text()
    batch_size = max(1, n_histories // n_batch)
    edition = max(1, n_batch // 10)
    content = template.format(batch_size=batch_size, n_batch=n_batch,
                              temperature=int(temperature), edition=edition,
                              **kwargs)
    Path(output_path).write_text(content)
```

La fonction MCNP est très similaire.

```
def format_mcnp_input(output_path, *, tabprob, temperature, **kwargs):
    boltzmann = 8.617341e-11  # MeV/K
    temperature_MeV = temperature*boltzmann
    template_path = JOB_PATH.with_name('templates') / 'mcnp'
    template = template_path.read_text()
    tabprob_int = 0 if tabprob else 1
    content = template.format(tabprob=tabprob_int, temperature=temperature_MeV,
                              **kwargs)
    Path(output_path).write_text(content)
```

Voici enfin la fonction `make_report()`, qui organise les résultats de test en sections.

```
def make_report(all_test_results, *, title):
    # Ici all_test_results est un dictionnaire qui associe les nom des tâches
    # à des listes de résultats de tests.
    sections = []
```

(continues on next page)

(continued from previous page)

```
    for task_name, test_results in sorted(all_test_results.items()):
        # On récupère le nom du noyau et l'énergie du test pour construire
        # le titre de la page.
        test = test_results[0].test
        nucleus = test.labels['nucleus']
        energy = test.labels['energy']
        section = TestReport(title=f'{nucleus}, {energy} MeV',
                             content=test_results)
        sections.append(section)

    report = TestReport(title=title,
                        content=[TestReport(title='Test results',
                                            text='Here is the good stuff.',
                                            content=sections)])
    return report
```

Exécution

Voici ce qui se passe quand on exécute le job avec `valjean` dans un terminal :

```
$ valjean run job.py
* graphs built in 0.6359915770590305 seconds
* hard_graph contains 11 tasks
* soft_graph contains 11 tasks
* will schedule up to 4 tasks in parallel
* deserializing pickle environment from 'valjean.env' files in spherical_cows/output
* 0 environment files found and deserialized
* graph completed in 1.5154480934143066e-05 seconds
* graph completed in 1.0513700544834137e-05 seconds
* hard graph copied in 0.00017551984637975693 seconds
* hard graph flattened in 1.2297183275222778e-05 seconds
* graph completed in 9.783543646335602e-06 seconds
* full graph computed in 0.00023496989160776138 seconds
* full graph flattened in 9.047798812389374e-06 seconds
* scheduling tasks
* full graph sorted in 6.301887333393097e-05 seconds
* master: 11 tasks left
* master: 5 tasks left
* task 'U235@14.0.t4' starts
* task 'U235@14.0.mcnp' starts
* task 'O16@14.0.mcnp' starts
* task 'O16@14.0.t4' starts
* task 'U235@14.0.mcnp' completed with status TaskStatus.DONE
* task 'Pu239@14.0.t4' starts
* master: 5 tasks left
* task 'O16@14.0.mcnp' completed with status TaskStatus.DONE
* task 'Pu239@14.0.mcnp' starts
* master: 5 tasks left
* task 'U235@14.0.t4' completed with status TaskStatus.DONE
* master: 5 tasks left
* master: 4 tasks left
* task 'comparison U235_14.0' starts
* Parsing spherical_cows/output/U235@14.0.t4/stdout
* Successful scan in 0.017888 s
* Successful parsing in 0.130465 s
```

(continues on next page)

(continued from previous page)

```
* task 'Pu239@14.0.mcnp' completed with status TaskStatus.DONE
* master: 4 tasks left
* task 'comparison U235_14.0' completed with status TaskStatus.DONE
* master: 4 tasks left
* task 'Pu239@14.0.t4' completed with status TaskStatus.DONE
* master: 4 tasks left
* master: 3 tasks left
* task 'comparison Pu239_14.0' starts
* Parsing spherical_cows/output/Pu239@14.0.t4/stdout
* Successful scan in 0.029409 s
* Successful parsing in 0.129308 s
* task 'comparison Pu239_14.0' completed with status TaskStatus.DONE
* master: 3 tasks left
* task 'O16@14.0.t4' completed with status TaskStatus.DONE
* master: 3 tasks left
* master: 2 tasks left
* task 'comparison O16_14.0' starts
* Parsing spherical_cows/output/O16@14.0.t4/stdout
* Successful scan in 0.027760 s
* Successful parsing in 0.093801 s
* task 'comparison O16_14.0' completed with status TaskStatus.DONE
* master: 2 tasks left
* master: 1 tasks left
* task 'report-report' starts
* task 'report-report' completed with status TaskStatus.DONE
* master: 1 tasks left
* task 'report' starts
* writing tree_path: spherical_cows/report/report/index
* writing tree_path: spherical_cows/report/report/Test results
* writing tree_path: spherical_cows/report/report/Test results/O16, 14.0 MeV
* writing tree_path: spherical_cows/report/report/Test results/Pu239, 14.0 MeV
* writing tree_path: spherical_cows/report/report/Test results/U235, 14.0 MeV
* writing 3 plots using 4 subprocesses
valjean/valjean/gavroche/stat_tests/student.py:478: RuntimeWarning: invalid value␣
↪encountered in true_divide
  studentt = diff.value / diff.error
* drawing figure spherical_cows/report/report/figures/plot_
↪cc240cbb3a9ae844616da16989931f4a21578e75aa8b9b3d895d36b2247ac7fd.png
* drawing figure spherical_cows/report/report/figures/plot_
↪9589d583b0b1af6b1ff2ec2aefcc88deda3ee5cec98d1a25c487e247c5aeaf52.png
* drawing figure spherical_cows/report/report/figures/plot_
↪1f8ae35814f479a1b25ba26f0e810126875b2a1cfb5ce3d2bb23bcd4f2de6f9a.png
* task 'report' completed with status TaskStatus.DONE
* final environment statistics:
     DONE: 11/11 (100.0%)
* serializing pickle environment to 'valjean.env' files
* 7 environment files written
```

Un rapport au format RST a été généré dans `report/report`. Je le convertis en HTML avec `sphinx` :

```
$ sphinx-build -j30 -a -bhtml report/report report/html
```

4 For developers

4.1 Installation and setup

valjean uses `pyproject.toml` file for packaging and dependency management as recommended in **PEP 517**, **PEP 518** and **PEP 621**.

If you plan to develop *valjean* and you have checked out the source repository, you need to install it in development mode:

```
$ python3 -m venv ~/venv-valjean-dev
$ source ~/venv-valjean-dev/bin/activate
$ pip install -U pip
(venv-valjean-dev) $ cd /path/to/valjean-X.Y.Z
(venv-valjean-dev) $ pip install -e .[dev]
```

This will install *valjean* and all of its dependencies in your current virtual environment. The development dependencies (e.g. *sphinx*, *pytest*, etc.) will also be installed; if you want to skip them, omit the `[dev]` string from the `pip install` command:

```
$ pip install -e .
```

This procedure should install just a link to your source folder `/path/to/valjean-X.Y.Z`, instead of copying the source files to the installation directory. This way, you will not need to reinstall the package every time you modify the sources.

You can also install *valjean* with poetry, version > 1.3 and `python > 3.7`:

```
$ python3 -m venv ~/venv-valjean-dev
$ source ~/venv-valjean-dev/bin/activate
$ pip install -U pip
$ pip install poetry
(venv-valjean-dev) $ cd /path/to/valjean-X.Y.Z
(venv-valjean-dev) $ poetry install -E dev
```

4.1.1 Dependency management

Developers should think about updating the package constraints in `pyproject.toml` when needed using poetry or directly.

See the poetry documentation for more information about adding or upgrading dependencies.

4.2 Testing

4.2.1 Unit tests

valjean uses pytest as a unit-test framework; some of the unit tests rely on the hypothesis package for property-based testing. The tests are defined in the `tests` folder and are automatically discovered by pytest.

You can run the unit-test suite with:

```
$ pytest
```

This will run the tests and produce a summary report. Additionally, pytest will calculate code coverage; a text report will be printed to stdout, and a nice HTML report will be written to `tests/htmlcov`.

The default *pytest* options are defined in the `pytest.ini` file, and *coverage* options are defined in `.coveragerc`, using pytest-doc. Extra options to *pytest* can be passed on the command line:

```
$ pytest tests/cosette/test_rlist.py  # run all tests in the named file
$ pytest tests/cosette/test_rlist.py::test_insert  # run this test only
$ pytest -k depgraph        # only run tests whose name matches "depgraph"
$ pytest -v                 # increase verbosity level by number of v
$ pytest --verbosity=N      # set verbosity test output
$ pytest -x                 # stop on the first test failure
$ pytest --ff               # run previously failed tests first
```

The `--verbosity` option sets all the *valjean* loggers to the verbosity level N. `N=0` is equivalent to `WARNING`, `N=1` to `INFO` (default) and `N>1` to `DEBUG`. The verbosity level can also be increased by invoking the `-v` option, possibly multiple times. It is useful when debugging a failing test.

4.2.2 Property-based testing and the hypothesis package

In traditional unit testing, one verifies that the code gives the expected answer for a given, fixed set of input values. For instance, one may test a sort algorithm by verifying that it produces `[1,2,3,4]` when fed `[4,2,3,1]`. However, this does not check that the algorithm behaves sensibly on any of the following arguments:

- empty lists;
- long lists;
- lists that are already sorted;
- lists of objects other than integers…

You see the point. One could write additional, specific unit tests to address each of these limitations. However, it is impossible to make sure that we have not forgotten some important edge case; also, we will need to write essentially the same tests over and over, and nobody likes doing the same thing over and over. But wait a minute, isn't "doing the same thing over and over" the kind of thing that computers are actually good at?

Enter *property-based testing*. The idea of this approach is that the developer should only check the expected properties of the code to be tested. For instance, in the case of the sort algorithm, the developer could check the following invariants about the list returned by the algorithm:

- it is sorted;
- it contains the same elements as the input list.

The testing library will generate a number of random inputs, call the sorting algorithm on each of them and check that the properties specified by the developer hold. If a counterexample is found, it will be shown to the user.

In Python, the hypothesis library offers a flexible and well-documented API for property-based testing. Additionally, hypothesis is well integrated with *pytest*. *valjean* unit tests leverage hypothesis whenever possible.

If you want some examples within *valjean*, a good place to start is the `tests.cosette.test_rlist` test module, which tests the invariants of the *RList* class.

4.2.3 Testing example docstrings with pytest

Sometimes the docstrings contain example code such as the following:

```
>>> print(1+2)
3
```

These examples are also automatically tested with pytest.

4.2.4 `tox` integration

There are specific `tox` test environments to run the unit tests. Check the page about *using tox for continuous integration*.

4.3 Linting

The **PEP 8** document makes a number of style recommendations for writing Python code. *valjean* tries to be PEP 8-compliant; you can check many of the PEP 8 recommendations with linters such as **`flake8`** or **`pylint`**, which are automatically pulled and installed when *valjean* is installed with the `[dev]` extra feature. Just run one of the following from the source folder:

```
$ flake8
$ pylint valjean tests
```

As a rule, try hard to keep *valjean* lint-free by running these commands often. If the linter emits a warning, you can do one of the following:

1. Fix the warning;
2. No, really, fix the warning;
3. If there's a warning you really can't fix, you can shut up the linter about that particular code line by adding special annotations (see below). But really, you should fix the warning instead.

4.3.1 Suppressing linter warnings

Sometimes, linters emit warning, but you **know** that the warning is actually benign. You can tell **flake8** or **pylint** to shut up about a specific warning by adding a comment with a special annotation in the source code. For **flake8**, suppression annotations look like this:

```
<code raising a warning>  # noqa: E402
```

Note the `E402` error code, which must matches **flake8**'s output message. For **pylint**, annotations look like this:

```
<code raising a warning>  # pylint: disable=trailing-whitespace
```

Again, the string after `disable=` must match the name of the warning in the **pylint** output.

See the **flake8** and **pylint** documentation for more details.

4.3.2 tox integration

There is a specific `tox` test environment to run the linters. Check the page about *using tox for continuous integration*.

4.4 Building the documentation

valjean uses the sphinx package for its own documentation. Before building the documentation, you will need to install *valjean*, as explained in *the installation section*.

The HTML documentation can be built with:

```
$ cd doc && make html
$ sphinx-build -a src build/html  # equivalently
```

The documentation will appear in `doc/build/html` and can be browsed starting with the `index.html` file.

sphinx is also able to build a LaTeX version of the documentation with:

```
$ cd doc && make latex
$ sphinx-build -a -b latex src build/latex  # equivalently
```

This will dump the LaTeX sources in `doc/build/latex`, where you can compile them to PDF with **make**.

The automatic documentation for the tests can be built by adding the `-t tests` option to `sphinx-build`:

```
$ sphinx-build -a -t tests src build/html
```

The documentation also contains a few examples in IPython notebook format. Conversion of the notebooks to HTML requires `pandoc`, which needs to be separately installed. Conversion of the notebooks is enabled by default. To deactivate it, pass the `-t no_notebooks` option to `sphinx-build`:

```
$ sphinx-build -a -t no_notebooks src build/html
```

The `-t tests` and `-t no_notebooks` can be used together.

The *sphinx* system is deeply customizable; most of the options are set in `doc/src/conf.py`, which is fairly well documented.

4.4.1 Version numbering weirdness

By default, the version number in the *sphinx* documentation is extracted from the **installed** version of *valjean*, and **not** the version in the current directory. If you are building the package documentation locally, then it doesn't really matter, but if you are building the documentation because you want to distribute the code to your users, **remember to install the package first!** It is simple:

```
$ cd /path/to/valjean  # the path containing pyproject.toml
$ pip install -U .[dev]
$ cd doc && make html
```

You will find the full recipe in *Distributing the code*.

4.4.2 Extensions

We use a few *sphinx* extensions:

autodoc
: Extracts the docstrings from the Python code and turns them into documentation.

doctest
: Runs the example code included in the docstrings, in the form of code execution at a Python prompt.

intersphinx
: Add hyperlinks to *sphinx* documentation outside the current project (for instance, in the Python standard library).

graphviz
: Include `dot` graphs inline, render them when the documentation is built.

todo
: Add TODO items, collect all of them in one place.

coverage
: Measure documentation coverage. To use it:

```
$ cd doc
$ make coverage
```

viewcode
: Add links to the source code.

imgmath
: Allows to write in math mode.

plot_directive
: Generate matplotlib plots from code included in the docs.

4.4.3 Checking references

To check internal references the `nitpicky` option can be used:

```
$ sphinx-build -a -n src build/html
```

from the `doc` folder, `-n` to activate the `nitpicky` option and `-a` (optional) to reconstruct documentation for all files.

4.4.4 Checking external links

The special `linkcheck` builder can be used to check any external links found in the documentation. Of course you must run the check from a machine with good network connectivity. The command is:

```
$ sphinx-build -a -b linkcheck src build/linkcheck
```

4.4.5 Building the documentation for the tests

The documentation for the unit tests is not built by default. If you want to build it, you should pass the `tests` tag to **`sphinx-build`**:

```
$ cd doc
$ sphinx-build -a -t tests src build/html
```

4.4.6 `tox` integration

There is a specific `tox` test environment to build the documentation. Check the page about *using tox for continuous integration*.

4.5 Distributing the code

Two package management systems are currently supported: `pip` and `conda`.

All the packages, including the documentation package, are available in the *Fichiers* area of the valjean Tuleap page.

4.5.1 Source distribution

Creation

A source distribution can be created with the command:

```
$ poetry build
```

This will produce a `valjean-x.y.z.tar.gz` source archive and a `valjean-x.y.z-py3-none-any.whl` in the `dist` folder. These files are suitable for the distribution of *valjean* to users and for installing it.

Installation

The source distribution may be installed with:

```
$ pip install /path/to/valjean-{x.y.z}.tar.gz
```

You can also install the wheel:

```
$ pip install /path/to/valjean-{x.y.z}-py3-none-any.whl
```

Both commands should work.

Note: If the archives are built from an untagged commit, the archive name will contain the hash of the commit, like `valjean-x.y.devz+ghash.tar.gz`.

4.5.2 Conda package

Creation

It is possible to create a conda package of *valjean* following these steps:

1. Install *conda* with miniconda or anaconda
2. Setup your conda workspace (`source MY/CONDA/PATH/bin/activate`)
3. Create the `conda` package:

```
$ cd /path/to/valjean
$ conda build conda.recipe --python=PY_VERSION
```

The conda package should appear in the conda installation at the path: `MY/CONDA/PATH/conda-bld/linux-64/valjean-v`*`VERSION-NUMBER_HASH`*`_py`*`PY_VERSION`*`.tar.bz2` with:

- `VERSION`: last tag from valjean in the branch used to build the archive
- `NUMBER`: number of commits since this tag
- `HASH`: short hash of the commit used
- `PY_VERSION`: Python version.

The option `--python` is needed in the command to get the correct python version in the archive name (else it uses the default version of conda).

If the build corresponds to the tag and the hash seems useless, it is possible to modify or comment the string. Without the string the default name would be `valjean-v`*`VERSION`*`-py`*`PY_VERSION_NUMBER`*`.tar.bz2`.

Installation

To install the conda package:

1. Setup your favorite conda workspace
2. Install the package:

```
$ conda install -c file://PATH/TO/valjean-DETAILS.tar.bz2 --use-local valjean
```

with `DETAILS=v`*`VERSION-NUMBER_HASH`*`_py`*`PY_VERSION`*.

Offline installation can be done adding the `--offline` option.

4.5.3 Documentation package

Creation

Follow the documentation build steps in *Building the documentation*, then archive the *html* folder:

```
$ cd doc/build
$ tar czf valjean-doc-XXX.tar.gz --transform 's,^,valjean-doc-XXX/,' html
```

Installation

To install the documentation:

```
$ tar xzf valjean-doc-XXX.tar.gz
```

4.6 Using `tox` for continuous integration

Unit tests, linting and documentation checks are part of the continuous integration (CI) suite. We use tox to orchestrate the tests and run the tests with different Python versions.

4.6.1 Configuration

The `tox` configuration is included in the `pyproject.toml` file.

4.6.2 Available test environments

In CI, the unit tests are run for several versions of Python (at the time of writing, all the supported versions are tested). For the most recent Python version (3.9 at the time of writing), the CI environment provides the optional ROOT dependency, which makes it possible to run the tests for the *`depletion`* module. For this reason, we pass the `PYTHONPATH` environment variable into the test environment.

4.6.3 Additional environments

There are three additional test environments, with specific purposes:

- `docs`: *builds the documentation*, with and without the `tests` tag, in *nitpicky mode*. It also runs *the linkcheck builder* to check for broken or redirecting external links.
- `linting`: *runs the linters*.
- `parsing`: runs all the unit tests, plus the specific (slow) parsing tests on the outputs of older TRIPOLI-4 versions.

4.6.4 Using `tox` from the command line

You can use `tox` to run your tests, build the documentation or lint the code from the command line. First, install `tox` in your *virtual environment*:

```
$ pip install tox
```

- To run the unit tests:

  ```
  $ tox -e py39 valjean tests  # for Python 3.9, for example
  ```

  You need to specify the Python version on the command line.

- Building the documentation:

  ```
  $ tox -e docs
  ```

- Linting:

  ```
  $ tox -e linting valjean tests
  ```

4.7 Release checklist

This page contains a list of things that you should check before issuing a new *valjean* release. In the future, some or all of the things below may (should!) be automated.

1. Choose the version number of the next release. Use Semantic Versioning.
2. Create a new branch off the development branch. If you are releasing version *x.y.z*, call the branch `releng-v`*x.y.z*. Commit all the following steps on this branch.
3. Revise this TODO list, if necessary!
4. Run the code *linters* and fix all the warnings.
5. Run all the *unit tests*, including the slow ones. Make sure they all pass. Use parallelism:

   ```
   $ pytest -m slow -n 4   # for 4 tests in parallel
   ```

   Push the `-n` option and overload the test machine a bit (e.g. use `-n 6` if you have 4 cores). If the tests start failing because they run too slowly, they are probably quite close to failing the health checks/deadlines in sequential mode anyway, and they may do so on another machine. Fix them so that they run faster.

   Also, run the command above a few times, in case some tests fail erratically.

   Running *pytest* also runs doctests. Make sure they all pass.
6. *Build archives* for the code and the documentation
7. Test *package installation*, with and without optional dependencies (`pip`, `conda` and documentation). Check the install/setup/tests/extras requirements: are they up to date?
8. Build the HTML documentation *in nitpicky mode*. Fix all the warnings, except the intentional ones.
9. *Check external documentation links* with the `linkcheck` builder. Fix the broken ones.
10. Check the *TODO list* and update it. Can anything be removed?
11. Update the *changelog*.
12. Commit and make a pull request.
13. Once the pull request is accepted, tag the new version:

    ```
    $ git tag vx.y.z  # for appropriate values of x, y and z
    ```

    Remember to push the tag to the shared repository.
14. *Create a source tarball and deploy it on the local network*. This requires reinstalling the package and rebuilding the documentation.
15. Remember to merge any release changes back into the development branch.
16. Congratulations, you have made a new *valjean* release!

5 Package API

5.1 valjean — VALidation, Journal d'Évolution et ANalyse

Fig. 1: Jean Valjean avec ses chandeliers, illustré par Émile Bayard (1862).

Top-level module for the valjean package.

class valjean.**ValjeanFormatter**(*info_fmt*, *fmt*)

Custom formatter for log messages.

This formatter class uses a different format for the `INFO` message level.

__init__(*info_fmt*, *fmt*)

Initialize the formatter.

Parameters

- **info_fmt** (*str*) – The format string for the INFO level.
- **fmt** (*str*) – The format string for the other message levels.

format(*record*)

Format the given record.

valjean.**set_log_level**(*level*)

Set the verbosity level for the default logger.

5.1.1 chrono — A context manager to time code sections

This module provides the *Chrono* context manager.

class valjean.chrono.**Chrono**

Time some code and store the elapsed time.

This class is a simple context manager that measures the execution time of a code fragment. The elapsed time is stored as a float in *self.elapsed* and can be accessed as such.

Examples:

```
>>> with Chrono() as chrono:
...     print(42)
42
>>> print(f'printing 42 took {chrono} seconds')
printing 42 took ... seconds
```

__init__()

__str__()

Return the elapsed time, as a string.

__float__()

Return the elapsed time, as a float.

__int__()

Return the elapsed time, as an int.

__format__(*format_spec*)

Return the elapsed time, formatted according to *format_spec*.

```
>>> chrono = Chrono()
>>> chrono.elapsed = 1e-6
>>> f'{chrono}'
'1e-06'
>>> f'{chrono:f}'
'0.000001'
```

5.1.2 config — Configuration object

Config objects encapsulate a set of configuration options for a *valjean* run. Here is how you create one:

```
>>> from valjean.config import Config
>>> config = Config()
```

By default, *Config* objects come with a `'path'` configuration section, which may be used to set default values for any configuration option. A few options are set from the beginning:

```
>>> for opt, val in sorted(config['path'].items()):
...     print(f'{opt} = {val}')
log-root = /.../log
output-root = /.../output
report-root = /.../report
```

The *Config* class behaves like a simple dictionary:

```
>>> print(config.query('path', 'report-root'))
/.../report
```

Module API

class valjean.config.**Config**(*dictionary=None*)

The base configuration class for *valjean*.

classmethod from_file(*path*)

Construct a configuration object from a TOML file.

Parameters
path (*pathlib.Path or str*) – A path for the configuration file.

__init__(*dictionary=None*)

Construct a configuration object from a dictionary.

The configuration will be initialized to contain a few default options.

Parameters
dictionary (*dict*) – The configuration object.

Returns
The constructed Config object.

__eq__(*other*)

Return self==value.

__ne__(*other*)

Return self!=value.

__str__()

Return str(self).

__repr__()

Return repr(self).

query(*section*, *option*)
Return the value of *option* from *section*.

set(*section*, *option*, *value*)
Set the value of *option* in *section* to be *value*.

__hash__ = None

5.1.3 dyn_import — Utilities for dynamically importing Python modules

This module contains somme common utility functions for dynamically importing Python source files as modules.

valjean.dyn_import.**split_module_path**(*file_name*)
Split a path to a Python module into a path and a module name.

Parameters
path (str or path-like object) – A path to a Python file.

Returns
a *(path, name)* tuple.

valjean.dyn_import.**dyn_import**(*file_name*)
Load a Python module from the given file name.

Note that this function adds the path to `file_name` to `sys.path`, so that local imports in `file_name` work as expected.

Parameters
file_name (*str*) – the name of the file containing the module.

Returns
the loaded module.

5.1.4 path — Utilities for accessing the filesystem

Utility functions to access the filesystem.

valjean.path.**ensure**(**paths*, *is_dir=False*)
Make sure that the given path exists.

Parameters

- **paths** (*str or pathlib.Path*) – One or more paths. Multiple arguments will be concatenated into a single path.
- **is_dir** (*bool*) – If *True*, the path will be constructed as a directory.

valjean.path.**sanitize_filename**(*name*)
Check that the *name* string can be used as a filename.

Raises
ValueError – if the string contains characters that are forbidden in a filename.

Returns
name unchanged

5.2 cambronne — Commandes, Actions et MoBilisation Rapide des OpératioNs Exécutables

Fig. 2: Pierre Cambronne (1770-1842), général de Napoléon, dans une gravure de l'époque.

5.2.1 `main` — Main executable

Main **valjean** executable.

`valjean.cambronne.main.`**`main`**(*argv=None*)
Main entry point for the **valjean** executable.

`valjean.cambronne.main.`**`make_parser`**()
Construct the argument parser.

`valjean.cambronne.main.`**`process_options`**(*args*)
Process the parsed options.

Returns
The configuration.

5.2.2 common — Common tools for the valjean executable

Common utilities for **valjean** commands.

class valjean.cambronne.common.**Command**
Base class for all **valjean** subcommands.

class valjean.cambronne.common.**DictKwargAction**(*option_strings*, *dest*, *nargs=None*, *const=None*, *default=None*, *type=None*, *choices=None*, *required=False*, *help=None*, *metavar=None*)
An argparse.Action subclass that parses arguments as key=value pairs and stores the resulting associations in a dictionary.

__call__(*parser*, *namespace*, *option*, *option_string=None*)
Add a key-value pair to the dictionary.

class valjean.cambronne.common.**JobCommand**
Base class for all **valjean** subcommands that take a job file and job arguments.

register(*parser*)
Add the *job_file* and *job_args* positional arguments to the parser.

valjean.cambronne.common.**run_job**(*job_file*, *job_args*, *job_kwargs*)
Run the *job()* function from the specified job file and return its result.

Parameters
- **job_file** (*str*) – the name of the file containing the *job()* function.
- **job_args** (*list(str)*) – the list of arguments to be passed to the *job()* function.
- **job_kwargs** (*dict*) – a dictionary of keyword arguments for *job()*

Returns
whatever *job()* returns; expected to be a list of *Task* objects.

Return type
list(*Task*)

valjean.cambronne.common.**check_unique_task_names**(*tasks*)
Check that the tasks have unique names.

Parameters
tasks (*list*) – A list of tasks.

Raises
ValueError – if two or more tasks have the same name.

valjean.cambronne.common.**collect_tasks**(*job_file*, *job_args*, *job_kwargs*)
Collect tasks from a job file, along with all their dependencies.

Parameters
- **job_file** (*str*) – the name of the file containing the *job()* function.

- **job_args** (*list(str)*) – the list of arguments to be passed to the *job()* function.
- **job_kwargs** (*dict*) – a dictionary of keyword arguments for *job()*

Returns
the collected tasks.

Return type
list(*Task*)

valjean.cambronne.common.**build_graphs**(*args*)

Build the dependency graphs according to the CLI parameters.

valjean.cambronne.common.**read_env**(*, *root*, *names*, *filename*, *fmt*)

Create an initial environment for the given task names, possibly merging a set of serialized environments.

The environment will be created from the partial environments that were serialized for the given task names. Missing partial environments will be silently ignored.

If *filename* is *None*, no de-serialization will take place and an empty environment will be returned.

Parameters

- **root** (*str*) – path to the root directory containing all the environment files.
- **names** (*list(str)*) – the list of task names that will be deserialized.
- **filename** (*str or None*) – Name of the file containing the serialized environment. If *None*, no de-serialzation will take place.
- **fmt** (*str*) – Environment serialization format (only `'pickle'` is supported at the moment).

Returns
an environment.

Return type
Env

valjean.cambronne.common.**write_env**(*env*, *, *filename*, *fmt*)

Serialize the environment to files.

The environment will be written to one file per task (i.e. one per environment key). The name of the environment file is given by the *filename* parameter, and the directory is the output directory (`'output_dir'` key) of the task. If the task does not have an `'output_dir'` key, serialization for that task will be skipped.

If *filename* is *None*, no serialization will take place at all.

Parameters

- **filename** (*str or None*) – Name of the file containing the serialized environment. If *None*, no serialzation will take place.
- **fmt** (*str*) – Environment serialization format (only `'pickle'` is supported at the moment).

5.2.3 env — The env subcommand

Module for the env subcommand.

class valjean.cambronne.commands.env.**EnvCommand**

Command class for the env subcommand.

register(*parser*)

Register options for this command in the parser.

static env(*args*, *_config*)

Execute the show-env command.

5.2.4 graph — The graph subcommand

Module for the graph subcommand.

class valjean.cambronne.commands.graph.**GraphCommand**

Command class for the build subcommand.

register(*parser*)

Register options for this command in the parser.

static execute(*args*, *_config*)

Execute the graph command.

static merge_graph_str(*solid_graph*, *dashed_graph*)

Merge two strings representing graphviz graphs into one graph. The graphs are merged in such a way that the edges of *solid_graph* and *dashed_graph* are respectively represented as solid and dashed.

For example:

```
>>> print(solid_graph)
digraph {
  node1 -> node2;
  node2 -> node3;
}
>>> print(dashed_graph)
digraph {
  node1 -> node3;
}
>>> print(GraphCommand.merge_graph_str(solid_graph, dashed_graph))
digraph {
  subgraph G1 {
  node1 -> node2;
  node2 -> node3;
  }
  subgraph G2 {
  edge [style=dashed];
  node1 -> node3;
  }
}
```

Parameters

- **solid_graph** (*str*) – a graph, as produced by *to_graphviz*.

- **dashed_graph** (*str*) – another graph, as produced by *to_graphviz*.

Returns
the merged graph.

Return type
str

5.2.5 run — The run subcommand

Module for the run subcommand.

class valjean.cambronne.commands.run.**RunCommand**

Command class for the run subcommand.

register(*parser*)

Register options for this command in the parser.

execute(*args*, *config*)

Execute the run command.

classmethod task_diagnostics(*, *tasks*, *env*, *config*)

Emit diagnostic messages about the status of the tasks. Count how many have succeeded, how many have failed, etc. If any tasks have failed, this method writes their names in a file called 'failed_tasks' in the log directory.

Parameters

- **tasks** (*list(Task)*) – the tasks that have been scheduled.
- **env** (Env) – the environment.
- **config** (Config) – the configuration object.

classmethod write_failed_tasks(*, *failed*, *config*)

Write the names of the failed tasks in the failed_tasks file.

Parameters

- **failed** (*list(str)*) – the names of the failed tasks.
- **config** (Config) – the configuration object.

valjean.cambronne.commands.run.**schedule**(*, *hard_graph*, *soft_graph*, *env*, *config=None*, *workers=1*)

Schedule a graph for execution.

5.3 cosette — COnStruction et ExécuTion de TâchEs

5.3.1 task — Task specification

This module collects a few task classes that can be used with the *scheduler* and *depgraph* modules.

This module defines a dummy *Task* class that may be used as a base class and extended.

The *Task.do* method takes two arguments:

Fig. 3: Cosette, fille adoptive de Jean Valjean, illustrée par Émile Bayard (1862).

- *env* is an environment for task execution. The idea of the environment is that tasks may use it to store information about their execution. For instance, a task may create a file and store its location in the environment, so that later tasks may be able to retrieve it. The type of *env* is really immaterial, but it is probably natural to use a key-value mapping of some kind. Note, however, that most of the tasks defined in the *task* module hierarchy expect *env* to be an *Env* object.
- *config* is a *Config* object describing the configuration for the current run. Tasks may look up global configuration values here.

The *Task* class models two types of inter-task dependencies. **Hard dependencies** represent dependencies that are crucial for the execution of the task at hand. If task *A* has a hard dependency on task *B*, it means that *A* cannot run unless *B* has successfully completed. If *B* fails, then it makes no sense to run *A*. On the other hand, if task *A* has a **soft dependency** on task *B*, it means that *A* will not start before *B*'s termination, but it makes sense to run *A* even if *B* fails.

class valjean.cosette.task.**TaskStatus**(*value*)

Enumeration for the task status. The possible values are:

- `WAITING` (the task is waiting to be scheduled)
- `PENDING` (the task is under execution)
- `DONE` (the task was executed and it succeeded)
- `FAILED` (the task was executed and it failed)
- `SKIPPED` (the task was skipped by the scheduler; this may happen, for instance, if the one of the task dependencies was not successful)

exception valjean.cosette.task.**TaskError**

An error that may be raised by *Task* classes.

class valjean.cosette.task.**Task**(*name*, *, *deps=None*, *soft_deps=None*)

Base class for other task classes.

__init__(*name*, *, *deps=None*, *soft_deps=None*)

Initialize the task.

Parameters

- **name** (*str*) – The name of the task. Task names **must** be unique!
- **deps** (*list(Task) or None*) – The list of (hard) dependencies for this task. It must be either *None* (i.e. no dependencies) or list of *Task* objects.
- **soft_deps** (*list(Task) or None*) – The list of soft dependencies for this task. It must be either *None* (i.e. no dependencies) or list of *Task* objects.

abstract do(*env*, *config*)

Perform a task.

Parameters

env – The environment for this task.

__str__()

Return str(self).

__repr__()
Return repr(self).

add_dependency(*dep*)
Add an item to the list of dependencies of this task.

depends(*other*)
Return *True* if *self* depends on *other*.

soft_depends(*other*)
Return *True* if *self* has a soft dependency on *other*.

class valjean.cosette.task.**DelayTask**(*name*, *delay=1.0*)
Task that waits for the specified number of seconds. This task is useful to test scheduling algorithms under different load conditions.

__init__(*name*, *delay=1.0*)
Initialize the task from a given delay.

Parameters
delay (*float*) – The amount of time (in seconds) that this task will wait when executed.

do(*env*, *config*)
Perform the task (i.e. sleep; I wish my life was like that).

Parameters
env – The environment. Ignored.

valjean.cosette.task.**det_hash**(**args*)
Produce a deterministic hash for the collection of objects passed as an argument.

valjean.cosette.task.**close_dependency_graph**(*tasks*)
Return the tasks along with all their dependencies.

Parameters
tasks (*list*) – A list of tasks.

Returns
The list of tasks, their dependencies, the dependencies of their dependencies and so on.

Return type
list(*Task*)

5.3.2 pythontask — Wrap Python code in a task

This module implements a task class called *PythonTask*, which is able to execute arbitrary Python code (in the form of a function call) when requested to do so. This is extra useful because functions wrapped in *PythonTask* can receive the scheduling environment as an additional parameter, which makes it possible to write code that depends on the results of previous tasks.

PythonTask objects

Creating a *PythonTask* is very simple. Let us define a function that returns a constant value:

```
>>> def func():
...   return 42
```

We can wrap the function in a *PythonTask* as follows:

```
>>> task = PythonTask('answer', func)  # 'answer' is the task name
```

We can then execute the task by passing an empty environment (dictionary) and an empty *Config* to the *PythonTask.do* method:

```
>>> task.do(env={}, config=None)
42
```

Note that *PythonTask.do* simply returns the result of the wrapped function. For the sake of illustration, our function returns an integer; however, if you want to use *PythonTask* in a *DepGraph*, your wrapped function will need to return an *(env_up, status)* pair, like the other tasks.

Note: Exceptions raised by the wrapped function are not caught by the task (they should be caught by the scheduler, though).

Calling a function without any arguments is not very restrictive per se. Say you want to call a function of two arguments:

```
>>> def add(x, y):
...   return x + y
```

If you have your arguments lying around at the time you construct your task, then you may do something like

```
>>> x, y = 5, 3
>>> task_add = PythonTask('add', lambda: add(x, y))
>>> task_add.do(env={}, config=None)
8
```

Essentially, you construct a trampoline: the lambda takes no arguments, but it captures *x* and *y* from the surrounding scope and passes them to the `add` function. This works, but due to the way Python handles captured variables, it may bring a few surprises:

```
>>> x, y = 5, 3
>>> task_add = PythonTask('add', lambda: add(x, y))
>>> x, y = 1, 2
>>> task_add.do(env={}, config=None)  # this still returns 8, right?
3
>>> # WAT
```

So, unless you know what you are doing, it is better to avoid this surprising behaviour and use the *args* argument to *PythonTask*:

```
>>> x, y = 5, 3
>>> task_add = PythonTask('add', add, args=(x, y))
```

(continues on next page)

(continued from previous page)

```
>>> task_add.do(env={}, config=None)
8
```

There is also a *kwargs* argument that can be used to pass keyword arguments:

```
>>> task_add = PythonTask('add', add, kwargs={'x': x, 'y': y})
>>> task_add.do(env={}, config=None)
8
```

Passing arguments via the environment

Sometimes your function requires some arguments, but the arguments themselves are not available (e.g. they haven't been computed yet) by the time you create your *PythonTask*. For this purpose, *PythonTask* provides an additional feature that allows the called function to query the task environment and retrieve any additional information from there.

The mechanism is simple: the environment is passed to the wrapped function as a keyword argument. The keyword can be specified by the user using the *env_kwarg* argument to the *PythonTask* constructor.

As a simple example, consider the following function:

```
>>> def goodnight(*, some_dict):
...   number = some_dict['number']
...   return 'Goodnight, ' + ('ding'*number)
```

You can wrap it in a *PythonTask* as follows:

```
>>> task_gnight = PythonTask('goodnight', goodnight, env_kwarg='some_dict')
```

and this is how it works:

```
>>> env = {'number': 8}
>>> task_gnight.do(env, config=None)
'Goodnight, dingdingdingdingdingdingdingding'
```

Passing arguments via the environment: a more complex example

As an illustration of a more complex scenario, we will now implement a set of *PythonTask* objects to calculate the Pascal's triangle. In plain Python, the code to print all the rows up to the n-th would look something like this:

```
>>> import numpy as np
>>> def pascal(n):
...   res = np.zeros((n, n), dtype=int)
...   res[:, 0] = 1
...   res[0, :] = 1
...   for i in range(2, n):
...     for j in range(1, i):
...       res[i-j, j] = res[i-j-1, j] + res[i-j, j-1]
...   return res
>>> direct_pascal = pascal(8)
>>> print(direct_pascal)
```

(continues on next page)

(continued from previous page)

```
[[ 1  1  1  1  1  1  1  1]
 [ 1  2  3  4  5  6  7  0]
 [ 1  3  6 10 15 21  0  0]
 [ 1  4 10 20 35  0  0  0]
 [ 1  5 15 35  0  0  0  0]
 [ 1  6 21  0  0  0  0  0]
 [ 1  7  0  0  0  0  0  0]
 [ 1  0  0  0  0  0  0  0]]
```

The logic is that each matrix element (except for those in the first row/column) is the sum of the element above and the element on the left.

In order to compute Pascal's triangle using *PythonTask* objects, we first need to decide on a strategy. We decide to use a *PythonTask* per matrix element. We also have to choose a strategy for naming the tasks, because the content of the environment is indexed by the task name. So we decide to call `'(i, j)'` the task that computes element *(i, j)*.

Armed with these conventions, we can write the function that computes element *(i, j)*:

```
>>> from valjean.cosette.task import TaskStatus
>>> def compute(name, i, j, *, env):
...   if i == 0 or j == 0:
...     env_up = {name: {'result': 1}}
...     return env_up, TaskStatus.DONE
...   left = str((i-1, j))
...   above = str((i, j-1))
...   left_result = env[left]['result']
...   above_result = env[above]['result']
...   result = left_result + above_result
...   env_up = {name: {'result': result}}
...   return env_up, TaskStatus.DONE
```

Note that we have to return an environment update and a status. Now we construct the tasks and assemble them into a dependency dictionary:

```
>>> deps = {}
>>> name_to_task = {}
>>> n = 8
>>> for k in range(n):
...   # k is the index of the row in the triange
...   # i and j index the matrix element, so k = i + j
...   for i in range(k+1):
...     j = k - i
...     task_name = str((i, j))
...     task = PythonTask(task_name, compute, args=(task_name, i, j),
...                       env_kwarg='env')
...     name_to_task[task_name] = task
...     deps[task] = set()
...     if i > 0:
...       index_left = str((i-1, j))
...       deps[task].add(name_to_task[index_left])
...     if j > 0:
...       index_above = str((i, j-1))
...       deps[task].add(name_to_task[index_above])
```

We can then import *DepGraph* and *Scheduler* and execute the dependency graph:

```
>>> from valjean.cosette.depgraph import DepGraph
>>> graph = DepGraph.from_dependency_dictionary(deps)
>>> from valjean.cosette.scheduler import Scheduler
>>> scheduler = Scheduler(hard_graph=graph)
>>> final_env = scheduler.schedule()
```

And now we can extract the results from the final environment:

```
>>> pythontask_pascal = np.zeros_like(direct_pascal)
>>> for k in range(n):
...   for i in range(k+1):
...     j = k - i
...     task_name = str((i, j))
...     pythontask_pascal[i, j] = final_env[task_name]['result']
>>> print(pythontask_pascal)
[[ 1  1  1  1  1  1  1  1]
 [ 1  2  3  4  5  6  7  0]
 [ 1  3  6 10 15 21  0  0]
 [ 1  4 10 20 35  0  0  0]
 [ 1  5 15 35  0  0  0  0]
 [ 1  6 21  0  0  0  0  0]
 [ 1  7  0  0  0  0  0  0]
 [ 1  0  0  0  0  0  0  0]]
>>> np.all(pythontask_pascal == direct_pascal)
True
```

Passing arguments via the configuration

The function wrapped in a *PythonTask* can also inspect the global *valjean* configuration object. This may be useful to retrieve global settings for paths, for instance. Like the environment, you can specify that the configuration should be passed to the wrapped function as a keyword argument. The keyword is specified by the *config_kwarg* parameter to the *PythonTask* constructor. For example:

```
>>> from valjean.config import Config
>>> def print_output_dir(*, config):
...     print(config.query('path', 'output-root'))
>>> task = PythonTask('output-dir', print_output_dir,
...                   config_kwarg='config')
>>> config = Config()
>>> task.do(env={}, config=config)
/.../output
```

Module API

exception valjean.cosette.pythontask.**TaskException**(*reason='unknown'*)

An exception that can be raised by any functions wrapped in *PythonTask*. It causes the task to fail. A reason can be specified in the constructor.

__init__(*reason='unknown'*)

Construct a *TaskException*.

Parameters

reason (*str*) – why this exception was raised.

class valjean.cosette.pythontask.**PythonTask**(*name*, *func*, *, *args=None*, *kwargs=None*, *env_kwarg=None*, *config_kwarg=None*, *deps=None*, *soft_deps=None*)

Task that executes specified Python code.

__init__(*name*, *func*, *, *args=None*, *kwargs=None*, *env_kwarg=None*, *config_kwarg=None*, *deps=None*, *soft_deps=None*)

Initialize the task with a function, a tuple of arguments and a dictionary of kwargs.

Parameters

- **name** (*str*) – The name of the task.
- **func** – A function to be executed.
- **args** (*tuple*) – A tuple of positional arguments to *func*, or *None* if none are required.
- **kwargs** (*dict*) – A dictionary of keyword arguments to *func*, or *None* if none are required.
- **env_kwarg** (*str*) – The name of the keyword argument that will be used to pass the environment to the function, or *None* if the environment should not be passed.
- **config_kwarg** (*str*) – The name of the keyword argument that will be used to pass the config to the function, or *None* if the config should not be passed.
- **deps** (None or collection of *Task* objects.) – If this task depends on other tasks (and valjean cannot automatically discover this), pass them (as a list) to the *deps* parameter.
- **soft_deps** (None or collection of *Task* objects.) – If this task has a soft dependency on other tasks (and valjean cannot automatically discover this), pass them (as a list) to the *soft_deps* parameter.

do(*env*, *config*)

Execute the function.

Parameters

- **env** – The environment. It will be passed to the executed function as the *env_kwarg* keyword, if specified.
- **config** – The config. It will be passed to the executed function as the *config_kwarg* keyword, if specified.

5.3.3 run — Task for generic command execution

This module defines a few useful functions and classes to embed generic command execution in a *depgraph*.

Spawning external processes

The *RunTask* class is the basic building block to run tasks that consist in spawning external processes and waiting for their completion. It makes it possible to execute arbitrary commands. Consider:

```
>>> from valjean.cosette.run import RunTask
>>> task = RunTask.from_cli(name='say', cli=['echo', 'ni!'])
>>> env_update, status = task.do(env=dict(), config=config) # prints 'ni!'
>>> print(status)
TaskStatus.DONE
```

The task succeeded, but where is the output of our command?! Note that *RunTask* captures standard output and standard error and redirects them to files. If you want to see what was printed, you have to look there:

```
>>> def print_stdout(env_up, name):
...   """A small function to print the stdout of a task."""
...   stdout = env_up[name]['stdout']
...   with open(stdout) as stdout_f:
...     print(stdout_f.read(), end='')
>>> print_stdout(env_update, 'say')
ni!
```

Note that *command* is not parsed by a shell. So the following may not do what you expect:

```
>>> task = RunTask.from_cli(name='want',
...                         cli=['echo', 'We want... ', '&&',
...                              'echo', 'a shrubbery!'])
>>> env_update, status = task.do(env=dict(), config=config)
>>> print_stdout(env_update, 'want')
We want...  && echo a shrubbery!
```

Indeed, *RunTask* ran **echo** only once. If you need to execute several commands, you can wrap them in a shell script and execute it. Alternatively, you can directly invoke the *RunTask.from_clis* class method:

```
>>> task = RunTask.from_clis(name='want',
...                          clis=[['echo', '-n', 'We want... '],
...                                ['echo', 'a shrubbery!']])
>>> env_update, status = task.do(env=dict(), config=config)
>>> print_stdout(env_update, 'want')
We want... a shrubbery!
```

Creating tasks using a factory

When you want to create multiple *RunTask* objects using the same executable, it can be convenient to use *RunTaskFactory*. This class can be parametrized to create tasks by specifying the path to the executable once and for all, for instance, and providing the missing arguments later.

The simplest way to create a *RunTaskFactory* is to use one of the *RunTaskFactory.from_executable* or *RunTaskFactory.from_task* class methods. For example, this will create a factory instance that generates *RunTask* objects for the **echo** executable:

```
>>> factory = RunTaskFactory.from_executable('echo', name='echo')
```

You can use it to generate tasks by invoking the *RunTaskFactory.make* method:

```
>>> task = factory.make(name='task', extra_args=['spam'])
```

This creates an *task.echo* object (of type *RunTask*) that executes `echo spam` when run:

```
>>> env_up, status = task.do(env={}, config=config)
>>> print_stdout(env_up, 'task.echo')
spam
```

You can also leave the *name* parameter out. If you do so, *RunTaskFactory* will generate a name for you:

```
>>> task_sausage = factory.make(extra_args=['sausage'])
>>> task_sausage.name
'....echo'
```

Of course you can generate multiple tasks using the same factory (this is the whole point of *RunTaskFactory*, really):

```
>>> task_spam = factory.make(name='task_spam', extra_args=['spam'])
>>> task_eggs = factory.make(name='task_eggs', extra_args=['eggs'])
>>> task_bacon = factory.make(name='task_bacon', extra_args=['bacon'])
```

You can also specify a few arguments beforehand and provide the rest later:

```
>>> factory = RunTaskFactory.from_executable('echo', name='echo',
...                                          default_args=['spam'])
>>> task = factory.make(name='task', extra_args=['eggs'])
>>> env_up, status = task.do(env={}, config=config)
>>> print_stdout(env_up, 'task.echo')
spam eggs
```

Finally, you can parametrize your arguments on arbitrary keywords that will be provided when the task is created:

```
>>> args = ['{food}', 'with', '{side}']
>>> factory = RunTaskFactory.from_executable('echo', name='echo',
...                                          default_args=args)
>>> task = factory.make(name='task', food='lobster', side='spam')
>>> env_up, status = task.do(env={}, config=config)
>>> print_stdout(env_up, 'task.echo')
lobster with spam
```

Default values for the parameters may be specified when creating the factory and can be overridden when the task is created:

```
>>> args = ['{food}', 'with', '{side}']
>>> factory = RunTaskFactory.from_executable('echo', default_args=args,
...                                          side='spam', name='echo')
>>> beans = factory.make(name='baked beans', food='baked beans')
>>> eggs = factory.make(name='eggs', food='eggs',
...                     side='bacon and spam')
>>> env_up, status = beans.do(env={}, config=config)
>>> print_stdout(env_up, 'baked beans.echo')
```

(continues on next page)

(continued from previous page)

```
baked beans with spam
>>> env_up, status = eggs.do(env={}, config=config)
>>> print_stdout(env_up, 'eggs.echo')
eggs with bacon and spam
```

Note also that you can refer to the environment or the configuration in your command-line arguments:

```
>>> args = ['{env[side]}']
>>> factory = RunTaskFactory.from_executable('echo', name='echo',
...                                          default_args=args)
>>> task = factory.make(name='task')
>>> env_up, status = task.do(env={'side': 'spam'}, config=config)
>>> print_stdout(env_up, 'task.echo')
spam
```

Caching

The *RunTaskFactory* class caches generated tasks under the hood. Repeated calls to *RunTaskFactory.make* from the same factory with the same arguments will result in the same task:

```
>>> factory = RunTaskFactory.from_executable('echo', name='echo')
>>> task_sausage = factory.make(extra_args=['sausage'])
>>> task_sausage_again = factory.make(extra_args=['sausage'])
>>> task_sausage is task_sausage_again
True
```

If you instantiate another factory, the caching mechanism is defeated:

```
>>> other_factory = RunTaskFactory.from_executable('echo', name='echo')
>>> task_sausage_other = other_factory.make(extra_args=['sausage'])
>>> task_sausage is task_sausage_other
False
```

Module API

valjean.cosette.run.**run**(*clis*, *stdout*, *stderr*, ***subprocess_args*)

Run the given command lines and capture their stdout/stderr.

Execution stops at the first failure (result value != 0).

Parameters

- **clis** (*list*) – The list of commands to execute.
- **stdout** (file object) – File handle to capture the stdout stream.
- **stderr** (file object) – File handle to capture the stderr stream.
- **subprocess_args** (*dict*) – Parameters to be passed to `subprocess.call`.

valjean.cosette.run.**make_cap_paths**(*base_path*)

Construct filenames to capture stdout and stderr.

class valjean.cosette.run.**RunTask**(*name*, *clis_closure*, *, *deps=None*, *soft_deps=None*, ***subprocess_args*)

Task that executes the specified shell commands and waits for their completion.

classmethod from_cli(*name*, *cli*, ***kwargs*)

Create a *RunTask* from a single command line.

Use the *RunTask.from_clis* method if you want to run several commands in a row.

Parameters

- **name** (*str*) – The name of this task.
- **cli** (*list*) – The command line to be executed, as a list of strings. The first element of the list is the command and the following ones are its arguments.
- **kwargs** – Any other keyword arguments will be passed on to the constructor (see *RunTask.__init__*).

classmethod from_clis(*name*, *clis*, ***kwargs*)

Create a *RunTask* from a list of command lines.

Parameters

- **name** (*str*) – The name of this task.
- **clis** (*list*) – The command lines to be executed, as a list of lists of strings. The first element of each sub-list is the command and the following ones are its arguments.
- **kwargs** – Any other keyword arguments will be passed on to the constructor (see *RunTask.__init__*).

__init__(*name*, *clis_closure*, *, *deps=None*, *soft_deps=None*, ***subprocess_args*)

Initialize this task from a list of command lines.

The *clis_closure* argument must be a closure. It will be invoked at execution time as:

```
clis_closure(env, config)
```

and it must return the command lines to be executed, as a list of lists of strings.

Parameters

- **name** (*str*) – The name of this task.
- **clis_closure** (*list*) – A closure to generate the command lines.
- **subprocess_args** (*dict*) – Any remaining options will be passed to the subprocess.Popen constructor.
- **deps** (*list(Task) or None*) – The dependencies for this task (see *Task.__init__* for the format), or *None*.
- **soft_deps** (*list(Task) or None*) – The dependencies for this task (see *Task.__init__* for the format), or *None*.

run_task(*clis_closure*, *name*, ***subprocess_args*)

Execute the specified command and wait for its completion.

On completion, this method proposes the following updates to the environment:

```
env[task.name]['clis'] = clis
env[task.name]['return_codes'] = return_codes
env[task.name]['elapsed_time'] = wallclock_time
env[task.name]['stdout'] = stdout
env[task.name]['stderr'] = stderr
```

Here `clis` is the list of command lines that were executed, `return_codes` is the list of return codes of the executed commands, and `elapsed_time` is the time the whole list took. The keys `stdout` and `stderr` hold the paths to the files containing respectively the captured standard output and standard error streams.

Parameters

- **env** (Env) – The task environment.
- **config** (*Config or None*) – The configuration object.

Returns
The proposed environment update.

class valjean.cosette.run.**RunTaskFactory**(*make_closure*, ***, *deps=None*, *soft_deps=None*, *name*, ***kwargs*)

Create multiple tasks from the same executable without even breaking a sweat.

classmethod from_task(*task*, ***, *relative_path*, *name=None*, *default_args=None*, ***kwargs*)

This class method creates a *RunTaskFactory* from a *Task*. The command to be executed must appear in the *output_dir* directory of the given task; its relative location must be specified with the *relative_path* argument.

This method can be used, among other things, to create a *RunTaskFactory* from a *CheckoutTask* or a *BuildTask*.

Parameters

- **task** (*Task*) – The *Task* object.
- **relative_path** (*str*) – The path to the executable, relative to the the output directory (*output_dir*) of the task.
- **name** (*str or None*) – a unique identifier for this factory. The factory name is used to produce unique names for the generated tasks. If *None* is given, the factory name will be constructed by hashing the other arguments.
- **default_args** – see *RunTaskFactory.from_executable*.
- **kwargs** – Any remaining arguments will be passed to the *__init__*.

classmethod from_executable(*path*, *name=None*, *default_args=None*, ***kwargs*)

This class method creates a *RunTaskFactory* from the path to an existing executable.

Parameters

- **path** (*str*) – the path to the executable.
- **name** (*str or None*) – a unique identifier for this factory. If *None* is given, the factory will use a hash of the other arguments.
- **default_args** (*list(str) or None*) – The list of arguments that will be passed to the executed command by *RunTask*. It may contain expressions understood by Python's format mini-language, such as `{foo}`.

These expressions can be filled when the task is generated by passing appropriate keyword arguments to the *make* method.

- **kwargs** – Any remaining arguments will be passed to the *__init__*.

__init__(*make_closure*, *, *deps=None*, *soft_deps=None*, *name*, ***kwargs*)

make(*, *name=None*, *extra_args=None*, *subprocess_args=None*, *deps=None*, *soft_deps=None*, ***kwargs*)

Create a *RunTask* object.

Parameters

- **name** (*str or None*) – the name of the task to be generated, as a string. If *None* is passed, *RunTaskFactory* will generate a name by hashing the contents of all the other arguments.
- **extra_args** (*list or None*) – A list of additional arguments that will be appended at the end of the command line.
- **subprocess_args** (*dict or None*) – A dictionary of arguments for the call to the `subprocess.Popen` constructor.
- **deps** (*list(Task) or None*) – A list of dependencies for the generated task. Note that factories built with *from_task* automatically inject dependencies on the given task.
- **soft_deps** (*list(Task) or None*) – A list of soft dependencies for the generated task.
- **kwargs** – Any remaining keyword arguments will be used to format the command line before execution. The environment and the configuration are available at formatting time as `env` and `config`, respectively.

copy()

Return a copy of this object.

5.3.4 code — Tools for checking out and building code

This submodule contains a few useful tasks for checking out, configuring, and building arbitrary code.

The *CheckoutTask* task class checks out a version-controlled repository. For the moment, only `git` repositories are supported. The path to the `git` executable may be specified through the *CheckoutTask.GIT* class variable.

Todo: Implement `svn` and `cvs` checkout; `copy` checkout (i.e. copy a directory from somewhere) may also be useful.

The *BuildTask* task class builds code from a given source directory. A build directory must also be specified and will be created if necessary. For the moment, *BuildTask* only supports ` `cmake` builds, but there are plans to add support for `autoconf`/`configure`/`make` builds. The path to the `cmake` executable may be specified through the *BuildTask.CMAKE* class variable.

Todo: Implement `autoconf`/`configure`/`make` builds.

To describe the usage of *CheckoutTask* and *BuildTask*, let us assume that `repo_dir` contains a `git` repository with a CMake project. We use a temporary directory `work_dir` for our test:

```
>>> checkout_dir = work_dir / 'checkout'
>>> build_dir = work_dir / 'build'
>>> log_dir = work_dir / 'log'
```

Now we can build checkout and build tasks for this repository:

```
>>> from valjean.cosette.code import CheckoutTask, BuildTask
>>> from pprint import pprint
>>> ct = CheckoutTask(name='project_checkout',
...                   repository=repo_dir,
...                   checkout_root=checkout_dir,
...                   log_root=log_dir)
>>> bt = BuildTask(name='project_build',
...                source=ct,
...                build_root=build_dir,
...                build_flags=['--' ,'-j4'],
...                log_root=log_dir)
```

Note how we passed the *ct* object directly to the *source* argument of the *BuildTask* constructor: we are telling the *BuildTask* to look for the sources to build in the checkout directory. You can also pass a normal path to the *source* argument instead.

```
>>> from valjean.cosette.env import Env
>>> env = Env()
>>> ct_up, ct_status = ct.do(env, config=None)
>>> print(ct_status)
TaskStatus.DONE
>>> pprint(ct_up)
{'project_checkout': {'checkout_log': '.../log/project_checkout.log',
                      'elapsed_time': ...,
                      'output_dir': '.../checkout/project_checkout',
                      'repository': '.../repo'}}
>>> env.apply(ct_up)  # apply CheckoutTask's environment update
...                   # for this example, this is actually optional
>>> bt_up, bt_status = bt.do(env=env, config=None)
>>> print(bt_status)
TaskStatus.DONE
>>> pprint(bt_up)
{'project_build': {'build_log': '.../log/project_build.log',
                   'elapsed_time': ...,
                   'output_dir': '.../build/project_build'}}
```

class valjean.cosette.code.**CheckoutTask**(*name*, *, *repository*, *checkout_root=None*, *log_root=None*, *flags=None*, *ref=None*, *vcs='git'*, *deps=None*, *soft_deps=None*)

Task to check out code from a version-control system. The actual code checkout is performed when the task is executed.

__init__(*name*, *, *repository*, *checkout_root=None*, *log_root=None*, *flags=None*, *ref=None*, *vcs='git'*, *deps=None*, *soft_deps=None*)

Construct a *CheckoutTask*.

Parameters

- **name** (*str*) – The name of this task.

- **checkout_root** (*str*) – The directory where the code will be checked out, or *None* for the configuration default.
- **repository** (*str*) – The repository for checkout.
- **log_root** (*str*) – The path to the log directory, or *None* for the configuration default.
- **flags** (*list(str) or None*) – The flags to be used at checkout time, as a list of strings.
- **ref** (*str or None*) – The reference to check out.
- **vcs** (*str or None*) – The version-control system to use. Must be one of: `'git'` (default), `'svn'`, `'cvs'`, `'copy'`.
- **deps** (*list(Task) or None*) – The dependencies for this task (see *Task.__init__()* for the format), or *None*.
- **soft_deps** (*list(Task) or None*) – The soft dependencies for this task (see *Task.__init__()* for the format), or *None*.

GIT = 'git'

Path to the `git` executable. May be overridden before class instantiation.

class valjean.cosette.code.**BuildTask**(*name*, *source*, ***, *build_root=None*, *log_root=None*, *targets=None*, *build_system='cmake'*, *configure_flags=None*, *build_flags=None*, *deps=None*, *soft_deps=None*)

Task to build an existing source tree. The build is actually performed when the task is executed.

__init__(*name*, *source*, ***, *build_root=None*, *log_root=None*, *targets=None*, *build_system='cmake'*, *configure_flags=None*, *build_flags=None*, *deps=None*, *soft_deps=None*)

Construct a *BuildTask*.

Parameters

- **name** (*str*) – The name of this task.
- **source** (*str*) – The path to the directory containing the sources, or a *CheckoutTask* object (in which case the checkout directory will be assumed to contain the sources).
- **build_root** (*str*) – The path to the build directory (a subdirectory will be created).
- **log_root** (*str*) – The path to the log directory.
- **targets** (*list(str) or None*) – A list of targets to build, or *None* for the default target.
- **build_system** (*str*) – The name of the build system to use. Must be one of: `'cmake'` (default), `'configure'`.
- **configure_flags** (*list*) – The flags that will be passed to the build tool at configuration time, as a list of strings.
- **build_flags** (*list*) – The flags that will be passed to the build tool at build time, as a list of strings.

- **deps** (*list(Task) or None*) – The dependencies for this task (see *Task.__init__()* for the type), or *None*.
- **soft_deps** (*list(Task) or None*) – The soft dependencies for this task (see *Task.__init__()* for the type), or *None*.

cmake_build_sys(*targets*, *configure_flags*, *build_flags*)

Return a function that builds a project using cmake.

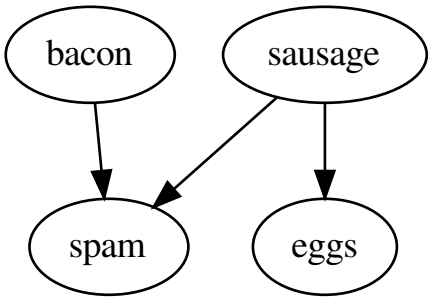
CMAKE = 'cmake'

Path to the `cmake` executable. May be overridden before class instantiation.

5.3.5 depgraph — Dependency graphs

The *DepGraph* class encapsulates a number of useful methods to deal with interdependent items. It represents the workhorse for the scheduling algorithms in the *scheduler* module.

A dependency graph is a directed acyclic graph which may be represented as follows:



The convention here is that if an edge goes from A to B, then A depends on B.

Building

You can create a *DepGraph* in of two ways. Either you pass a dictionary representing the dependencies between items to the *from_dependency_dictionary* class method:

```
>>> from valjean.cosette.depgraph import DepGraph
>>> from pprint import pprint
>>> deps = {'bacon': ['spam'], 'sausage': ['eggs', 'spam']}
>>> pprint(deps)
{'bacon': ['spam'], 'sausage': ['eggs', 'spam']}
>>> g = DepGraph.from_dependency_dictionary(deps)
```

or you create an empty graph and add nodes and dependencies by hand:

```
>>> h = DepGraph().add_dependency('bacon', on='spam') \
...               .add_dependency('sausage', on='eggs') \
```

(continues on next page)

(continued from previous page)

```
...                 .add_dependency('sausage', on='spam')
>>> g == h
True
```

In these examples, *sausage* depends on *eggs* and *spam*, and *bacon* depends on *spam*; *spam* and *eggs* have no dependencies. Note that the dependency dictionary may be seen as a sparse representation of the graph adjacency matrix.

You can recover the dependency dictionary by passing the graph to `dict()`:

```
>>> pprint(dict(g))
{'bacon': {'spam'}, 'eggs': set(), 'sausage': {'eggs', 'spam'}, 'spam': set()}
```

If you print the dictionary, you will notice that *spam* and *eggs* have been added as nodes with no dependencies:

```
>>> print(g)
[('bacon', ['spam']), ('eggs', []), ('sausage', ['eggs', 'spam']), ('spam', [])]
```

What if a node has no dependencies and no other node depends on it? Just add it to the dictionary with the empty list as a value:

```
>>> free = DepGraph.from_dependency_dictionary({'kazantzakis': []})
```

You can also add it after creating the graph:

```
>>> also_free = DepGraph().add_node('kazantzakis')
>>> free == also_free
True
```

Querying

You can inspect the nodes of the graph:

```
>>> sorted(g.nodes())
['bacon', 'eggs', 'sausage', 'spam']
```

or ask for the dependencies of a node:

```
>>> sorted(g.dependencies('sausage'))
['eggs', 'spam']
>>> sorted(g['sausage'])  # equivalent, shorter syntax
['eggs', 'spam']
```

or ask for the nodes that depend on another node (careful though, this operation has *O(N)* time complexity, *N* being the number of nodes in the graph):

```
>>> sorted(g.dependees('spam'))
['bacon', 'sausage']
```

You can also iterate over graphs:

```
>>> for k, vs in sorted(g):
...     for v in sorted(vs):
```

(continues on next page)

(continued from previous page)

```
...         print(f"You can't have {k} without {v}!")
You can't have bacon without spam!
You can't have sausage without eggs!
You can't have sausage without spam!
```

Finally, you can check if a graph is a subgraph of another one:

```
>>> sub_g = DepGraph().add_dependency('bacon', on='spam') \
...                   .add_dependency('sausage', on='eggs')
>>> sub_g <= g
True
```

Merging, sorting and other operations

Given two graphs, possibly sharing some nodes and edges, you can construct the *union g12* as follows:

```
>>> g1 = sub_g
>>> g2 = DepGraph().add_dependency('sausage', on='spam')
>>> g12 = g1 + g2
>>> g12 == g
True
>>> _ = g2.merge(g1)  # in-place merge
>>> g2 == g
True
>>> g2 += g1          # equivalent syntax
>>> g2 == g
True
```

It is also possible to compute the transitive reduction of the graph. Let *g* be an acyclic graph. The transitive reduction *tr(g)* is the minimal (in the number of edges), provably unique subgraph of *g* over the same vertices with the following property: for each pair of nodes *A* and *B*, *A* is reachable from *B* within *g* iff it is reachable within *tr(g)*.

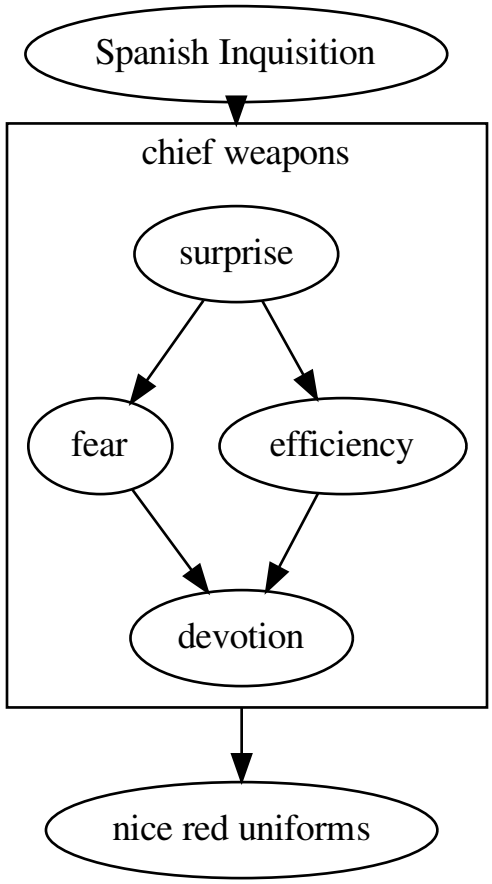
```
>>> g_redundant = DepGraph() \
...     .add_dependency('eggs', on='bacon') \
...     .add_dependency('bacon', on='spam') \
...     .add_dependency('eggs', on='spam')  # this edge is redundant
>>> g_tr = g_redundant.copy()
>>> print(g_tr.transitive_reduction())
[('bacon', ['spam']), ('eggs', ['bacon']), ('spam', [])]
>>> 'spam' in g_tr.dependencies('eggs')
False
```

You can also do a topological sort of the graph. The result is a list of the graph nodes, with the property that each node is guaranteed to appear after all the nodes it depends on. Note that in general there are several possible topological sorts for a given graph.

```
>>> g.topological_sort()
['eggs', 'spam', 'bacon', 'sausage']
```

Grafting sub-graph nodes into the graph itself

A nice feature of *DepGraph* is that you can have graph nodes that are themselves instances of *DepGraph*! Consider the following graph:



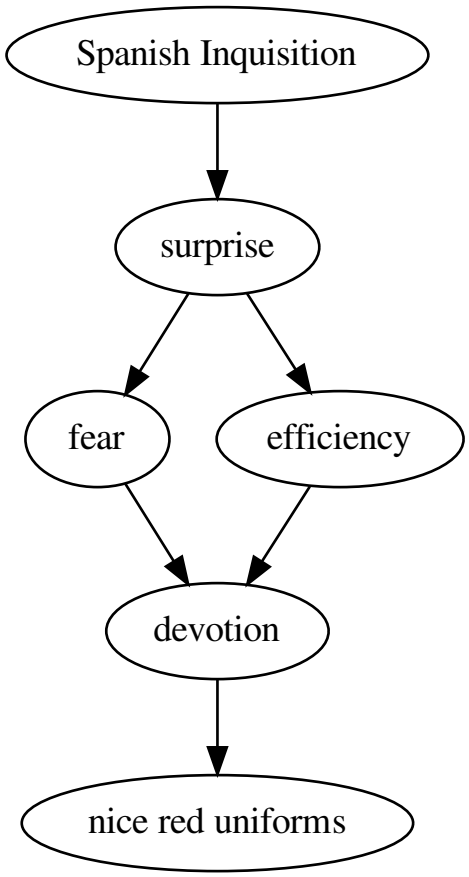
Here is the code that generates it:

```
>>> spanish_inq = 'Spanish Inquisition'
>>> surprise, fear, efficiency = 'surprise', 'fear', 'efficiency'
>>> devotion, uniforms = 'devotion', 'nice red uniforms'
>>> chief_weapons = DepGraph.from_dependency_dictionary({
...     surprise: [fear, efficiency],
...     fear: [devotion],
...     efficiency: [devotion]
...     })
>>> spanish = DepGraph() \
...      .add_dependency(spanish_inq, on=chief_weapons) \
...      .add_dependency(chief_weapons, on=uniforms)
```

Here *chief_weapons* is a *DepGraph* itself, but it is also a node of *spanish*. You can graft *chief_weapons* inside *spanish* like so:

```
>>> spanish.graft(chief_weapons)
DepGraph(...)
```

This yields the following graph



as you can verify yourself:

```
>>> full_graph = DepGraph.from_dependency_dictionary({
...     spanish_inq: [surprise],
...     surprise: [fear, efficiency],
...     fear: [devotion],
...     efficiency: [devotion],
...     devotion: [uniforms]
...     })
>>> full_graph == spanish
True
```

The nice thing about this feature is that it makes it easier (more modular) to build complex graphs. Just build smaller subgraphs and assemble them together as if they were nodes! The *flatten* method will recursively graft all graph nodes into the main graph.

(Fun fact: the *DepGraph* type is a monad, with *flatten* playing the role of `join`. If what I just wrote makes no sense to you, don't worry.)

Caveats

Some things you should be aware of when using *DepGraph*:

- The type of the items in the graph is irrelevant, but if you want to use the *from_dependency_dictionary* constructor they need to be stored in a dictionary, and therefore they must be *hashable*;
- You need to use a single-element list if you want to express a single dependency, as in the case of *bacon*. So this is wrong:

```
>>> bad_deps = {0: 1, 7: [0, 42]}  # error, should be 0: [1]
>>> bad = DepGraph.from_dependency_dictionary(bad_deps)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    [...]
TypeError: 'int' object is not iterable
```

  Here 1 is not iterable, and the test crashed. However, if your graph nodes happen to be iterable, the *DepGraph* constructor will not crash, but you will not get what you expect.

exception valjean.cosette.depgraph.**DepGraphError**

An exception raised by *DepGraph*.

class valjean.cosette.depgraph.**DepGraph**(*nodes=None*, *edges=None*)

A dependency graph.

There are two preferred ways to instantiate this class:

- using the *from_dependency_dictionary* class method;
- constructing an empty graph and repeatedly calling *add_dependency* and/or *add_node*.

Alternatively, you may also use the full form of the constructor; in this case, the graph *edges* are represented as a dictionary between integers, with the convention that each node is represented by its index in the *nodes* list. So *nodes* can be seen as a mapping from integers to nodes. The inverse mapping is called *index* and may be passed to the constructor if it is available to the caller as a dictionary; if not, it will be constructed internally.

classmethod from_dependency_dictionary(*dependencies*)

Generate a *DepGraph* from a dependency dictionary.

__init__(*nodes=None*, *edges=None*)

Initialize the object from a list of nodes and an edge dictionary.

Parameters

- **nodes** (iterable) – An iterable over the nodes of the graph, or *None* for an empty graph.
- **edges** (mapping) – A mapping between integers representing the nodes, or *None* for an empty graph.

__str__()
Return str(self).

__repr__()
Return repr(self).

__add__(*other*)
Merge two graphs, return the result as a new graph.

__radd__(*other*)
Merge two graphs, return the result as a new graph.

__len__()
Return the number of vertices in the graph.

__contains__(*node*)
Returns *True* if *x* is one of the nodes.

__le__(*other*)
g <= *h* if *g* is a subgraph of *h*

isomorphic_to(*other*)
Returns *True* if this graph is isomorphic to *other*.

__eq__(*other*)
Returns *True* if this graph is isomorphic to *other*.

add_node(*node*)
Add a new node to the graph.

Parameters
node – The new node.

remove_node(*node*)
Remove a node from the graph.

Any edges going in and out of the node will be removed, too.

Parameters
node – The node to be removed.

add_dependency(*node*, *on*)
Add a new dependency to the graph.

Parameters

- **node** – The node for which the dependency is specified.
- **on** – The node *node* depends on.

remove_dependency(*node*, *on*)
Remove a dependency from the graph.

Parameters

- **node** – The node for which the dependency is specified.
- **on** – The node *node* depends on.

invert()

Invert the graph.

Returns
A new graph having the same nodes but all edges inverted.

topological_sort()

Perform a topological sort of the graph.

Returns
The nodes of the graph, as a list, with the invariant that each node appears in the list after all the nodes it depends on.

Raises
DepGraphError – if the graph is cyclic.

copy()

Return a copy of this graph.

merge(*other*)

Merge this graph in place with another one.

Parameters
other (DepGraph) – The graph to merge.

__iadd__(*other*)

Merge this graph in place with another one.

Parameters
other (DepGraph) – The graph to merge.

nodes()

Returns the graph nodes.

dependencies(*node*, *recurse=False*)

Query the graph about the dependencies of *node*. If *recurse* is True, also return indirect dependencies (the transitive closure of the specified node). With *recurse=False*, this operation is *O(1)*.

Parameters
recurse (*bool*) – If true, return indirect dependencies as well.

Returns
A list containing the dependencies of *node*.

__getitem__(*node*, *recurse=False*)

Query the graph about the dependencies of *node*. If *recurse* is True, also return indirect dependencies (the transitive closure of the specified node). With *recurse=False*, this operation is *O(1)*.

Parameters
recurse (*bool*) – If true, return indirect dependencies as well.

Returns
A list containing the dependencies of *node*.

dependees(*node*)

Collect the nodes that depend on the given node. This operation is *O(N)*, where *N* is the number of nodes in the graph.

Returns
A set containing the dependees of *node*.

to_graphviz()
Convert the graph to graphviz format.

Returns
A string describing the file in graphviz format.

transitive_reduction()
Perform a transitive reduction of the graph in place.

transitive_closure()
Perform a transitive closure of the graph in place.

initial()
Return the initial nodes of the graph.

The initial nodes are the nodes that have no ingoing edge; i.e., no other node depends on them.

Returns
The list of initial nodes.

terminal()
Return the terminal nodes of the graph.

The terminal nodes are the nodes that have no outgoing edge; i.e., they do not depend on any other node.

Returns
The list of terminal nodes.

graft(*node*)
Graft the given node into the graph.

Parameters
node (DepGraph) – A DepGraph embedded as a graph node.

__hash__ = None

flatten(*recurse=True*)
Graft all DepGraph nodes into this graph.

Parameters
recurse (*bool*) – If true, recursively graft DepGraph nodes until all nodes are flat.

depends(*node1*, *node2*, *recurse=False*)
Return `True` if the node *node1* depends on *node2*.

Parameters

- **node1** – The first node.
- **node2** – The second node.
- **recurse** (*bool*) – If true, look at indirect dependencies, too.

Returns
`True` if *node1* directly (`recurse == False`) or indirectly (`recurse == True`) depends on *node2*.

5.3.6 `rlist` — Reversible lists

A reversible list (*`RList`* for short) keeps track of the indices of its elements for fast reverse lookup. It has mostly the same semantics as lists:

```
>>> from valjean.cosette.rlist import RList
>>> dead, stiff, bereft, rests = ('dead', 'stiff', 'bereft of life',
...                               'rests in peace')
>>> parrot = RList([dead, stiff, bereft, rests])
>>> parrot[0]
'dead'
>>> parrot[-2]
'bereft of life'
>>> del parrot[1]
>>> print(parrot)
['dead', 'bereft of life', 'rests in peace']
>>> parrot == ['dead', 'bereft of life', 'rests in peace']
True
>>> parrot == RList(['a stiff'])
False
```

Additionally, you can quickly (in *O(1)* time on average) look up the index of an item or check if an item is contained in the list:

```
>>> parrot.index(rests)  # this call is O(1)
2
>>> rests in parrot  # and so is this
True
```

This operation takes *O(n)* average time in normal lists.

The most important differences with respect to the semantics of traditional Python lists are:

1. Slicing operations are not supported.
2. The notion of "containment" is user-defined. By default, a value belongs to an *`RList`* if it has the same *key* as one of the list elements. The *key*, by default, is the value `id`. In other words, normal lists compare items with the == operator, while *`RList`* compares object IDs by default. Objects that compare equal (with the standard == operator) may be used interchangeably in a list, but not in a *`RList`*. For example, here is a simple class:

   ```
   >>> class A:
   ...     def __init__(self, x):
   ...         self.x = x
   ...     def __eq__(self, other):
   ...         return self.x == other.x
   ```

   and here is how it behaves in a normal list:

   ```
   >>> a1 = A(42)
   >>> a2 = A(42)
   >>> a1 == a2
   True
   >>> a1 in [a2]
   True
   ```

   The objects `a1` and `a2` compare equal, so to the eyes of the list `[a2]` they are "the same". *`RList`*, on the other hand, does not play along with this charade by default:

```
>>> a1 is a2
False
>>> id(a1) == id(a2)  # equivalent to the line above
False
>>> a1 in RList([a2])
False
```

Since `a1` and `a2` are distinct objects (they live in different memory locations), they are different in the eyes of *RList*. This may result in unexpected behaviour, especially with strings, ints or other small objects for which the Python interepreter may provide some kind of caching optimisation:

```
>>> lst = RList([1])
>>> 1 in lst
True
>>> lst = RList([1234567])
>>> 1234567 in lst
False
>>> # wat?
>>> 1234567 in RList([1234567])
True
>>> # WAT?
```

This weird behaviour is actually a well-documented quirk of the CPython implementation.

If you want, you can define your own notion of *key* for your objects by passing a suitable function to the `key` argument of the *RList* constructor; the only constraint is that the value returned by *key* must be hashable. For example, if you have a list of strings, you can use the string itself as a key as follows:

```
>>> parrot_by_value = RList(['Norwegian blue', 'plumage', 'pining'],
...                         key=lambda x: x)
>>> 'plumage' in parrot_by_value
True
>>> 'bereft of life' in parrot_by_value
False
```

You can also use more sophisticated *key* functions:

```
>>> a_rlist = RList([A(0), A(1), A(2)], key=lambda a: a.x)
>>> A(2) in a_rlist
True
>>> A(5) in a_rlist
False
```

class valjean.cosette.rlist.**RList**(*args=None*, *, *key=<built-in function id>*)

Create a reversible list from an iterable.

Parameters
args – An iterable containing some elements.

__init__(*args=None*, *, *key=<built-in function id>*)

__repr__()
Return repr(self).

__str__()
Return str(self).

__eq__(*other*)
Return self==value.

__ne__(*other*)
Return self!=value.

insert(*index*, *value*)
Insert an element at a given index.

index(*value*, *start=0*, *stop=None*)
Return the index of the given value, if present.

Parameters

- **value** – The object to search for.
- **start** (*int*) – The index to search from.
- **stop** (*int*) – The index to search up to.

Raises
ValueError – if the element is not present in the container.

Returns
The index of (the first occurrence of) *value* in the list. The returned value *i* always satisfies *start* <= *i* < *stop*.

indices(*value*)
Return all the indices of the given value, if present.

Parameters
value – The object to search for.

Raises
KeyError – if the element is not present in the container.

Returns
All the list indices where *value* occurs.

get_index(*value*, *default*)
Return the index of the given value, or a default if the value is missing.

Parameters

- **value** – The object to search for.
- **default** – The value to be returned if *value* is not present in the container.

swap(*i*, *j*)
Swap two elements of the list.

After this operation, the ith and jth elements will be swapped.

Parameters

- **i** (*int*) – The index of the first element.
- **j** (*int*) – The index of the second element.

copy()
Return a shallow copy of this object.

__hash__ = None

5.3.7 env — Environment for task execution

This module defines *Env*, a class that makes it simpler to add information about running tasks. An *Env* object can be created from an existing dictionary as follows:

```
>>> from valjean.cosette.env import Env
>>> quest = {'name': 'Sir Galahad', 'favourite colour': 'blue'}
>>> env_quest = Env(quest)  # a shallow copy of `quest` is performed
```

You can use an *Env* object as a glorified dicitionary (*Env* inherits from `dict`), but its main purpose is really to store information about concurrently running tasks (see *Task*). By convention, the keys in *Env* are assumed to be task names; the associated values are dictionaries storing whatever information may be useful about the task. The dictionaries are also expected to have a `'status'` key describing the current task status (see *TaskStatus*). An example of *Env* respecting these conventions is the following:

```
>>> from valjean.cosette.task import Task, TaskStatus
>>> class FindHolyGrail(Task):
...   '''We derive a class from Task, which is abstract.'''
...   def do(self, env, config):
...     # find the Holy Grail
...     pass
>>> quest = FindHolyGrail('quest')  # 'quest' is the task name
>>> tasks = {quest.name: {'name': 'Sir Galahad',
...                       'favourite colour': 'blue',
...                       'status': TaskStatus.FAILED}}
>>> env = Env(tasks)
```

The *Env* API integrates well with the *task* module and provides a number of practical methods for dealing with tasks. For instance, there are `is_*` methods for all members of the *TaskStatus* enumeration:

```
>>> env.is_failed(quest)
True
>>> print(env.get_status(quest))    # equivalently
TaskStatus.FAILED
```

Additionally, you can change the status of a task with

```
>>> env.set_status(quest, TaskStatus.DONE)
>>> env.set_done(quest)  # equivalent, shorter version
```

Information about the tasks, incuding their status, is stored with the task name as the key:

```
>>> print(env[quest.name]['status'])
TaskStatus.DONE
```

The *Env* class tries hard to be thread-safe; that is, all its methods will operate atomically. Internally, thread safety is enforced by locking the object whenever its contents are accessed. This, however, does not help in case of read-and-modify operations, as in the following example:

```
>>> if env.is_done(quest):      # WARNING: do not try this at home
...     env.set_skipped(quest)  # race condition here!
```

This snippet is racy in multithreaded mode because another thread may change the status of `quest` after *is_done* has released the lock but before *set_skipped* has had the chance

to acquire it. For these scenarios, *Env* offers the *atomically* method, which accepts as an argument the action that should be performed. When called, *atomically* first acquires the lock on the object, and then passes the *Env* object to the action. A thread-safe implementation of the read-and-modify trip above is implemented as follows:

```
>>> def modify_task1(env):
...     if env.is_done(quest):
...         env.set_skipped(quest)
>>> env.atomically(modify_task1)
>>> env.is_skipped(quest)
True
```

exception valjean.cosette.env.**EnvError**

An error that may be raised by the *Env* class.

class valjean.cosette.env.**Env**(*dictionary=None*)

The *Env* class can be used to store and dynamically update information about concurrently running tasks, and offers thread-safety guarantees.

__init__(*dictionary=None*)

Construct an object based on an existing dictionary.

__repr__()

Return repr(self).

classmethod from_file(*path*, *, *fmt='pickle'*)

Deserialize an *Env* object from a file.

Parameters

- **path** (*str*) – Path to the file.
- **fmt** (*str*) – Serialization format (only `'pickle'` is supported for the moment).

Returns

The deserialized object.

to_file(*path*, *, *task_name=None*, *fmt='pickle'*)

Serialize an *Env* object to a file.

Parameters

- **path** (*str*) – path to the file.
- **task_name** (*str or None*) – name of the task to serialize, or *None* to serialize the whole environment.
- **fmt** (*str*) – serialization format (only `'pickle'` is supported for the moment).

merge_done_tasks(*other*)

Merge task status from another environment.

This method takes an additional environment *other* as an argument. If the same key appears in *self* and *other* and `other[key]['status'] == TaskStatus.DONE`, then it sets `self[key] = other[key]`.

The idea is that *self* might contain a pristine environment, while *other* might provide the results of a previous run. We want to mark completed tasks as *DONE*, but we also want to re-run those that failed.

Parameters
other (Env) – The environment providing the updates.

set_status(*task*, *status*)
Set *task*'s status to *status*.

get_status(*task*)
Return *task*'s status.

atomically(*action*)
Perform an action atomically on the environment dictionary. The dictionary passes itself as the first argument to *action*, which must be callable.

apply(*env_update*)
Apply un update to the dictionary.

set_start_end_clock(*task*, *, *start*, *end*)
Set the start and end time for the given task.

get_start_clock(*task*)
Return the start time for the given task.

get_end_clock(*task*)
Return the end time for the given task.

copy()
Return a shallow copy of *self*.

__getstate__()
Do not serialize the lock, as doing so results in exceptions with Python >=3.6.

__setstate__(*state*)
The serialized state does not contain a lock; create a new one instead.

is_done(*task*)
Returns True if *task*'s status is *DONE*.

is_failed(*task*)
Returns True if *task*'s status is *FAILED*.

is_pending(*task*)
Returns True if *task*'s status is *PENDING*.

is_skipped(*task*)
Returns True if *task*'s status is *SKIPPED*.

is_waiting(*task*)
Returns True if *task*'s status is *WAITING*.

set_done(*task*)
Sets *task*'s status to *DONE*.

set_failed(*task*)
Sets *task*'s status to *FAILED*.

set_pending(*task*)
Sets *task*'s status to *PENDING*.

set_skipped(*task*)
Sets *task*'s status to *SKIPPED*.

set_waiting(*task*)
Sets *task*'s status to *WAITING*.

5.3.8 scheduler — Task scheduling and dependency handling

This module provides classes to schedule the execution of several tasks, possibly dependent on each other.

Example usage:

```
>>> from valjean.cosette.task import DelayTask
>>> spam = DelayTask('spam', 0.1)
>>> eggs = DelayTask('eggs', 0.2)
>>> bacon = DelayTask('bacon', 0.2)
```

```
>>> from valjean.cosette.depgraph import DepGraph
>>> g = DepGraph.from_dependency_dictionary({
...         spam: [],
...         bacon: [spam],
...         eggs: []
...     })
```

```
>>> from valjean.cosette.scheduler import Scheduler
>>> s = Scheduler(hard_graph=g)
>>> env = s.schedule()  # executes the tasks in the correct order
```

exception valjean.cosette.scheduler.**SchedulerError**
An error that may be raised by the *Scheduler* class.

class valjean.cosette.scheduler.**Scheduler**(*, *hard_graph*, *soft_graph=None*, *backend=None*)
Schedule a number of tasks.

The Scheduler class has the responsibility of scheduling and executing a number of tasks. Here *hard_graph* is a *DepGraph* describing the *hard* dependencies among the tasks to be executed (see *task* for details about *hard* vs. *soft* dependencies); *soft_graph* is a *DepGraph* describing the *soft* dependencies alone; finally, *backend* should be an instance of a **Scheduling* class such as *QueueScheduling*, or at any rate a class that exhibits an *execute_tasks* method with the correct signature (see *QueueScheduling.execute_tasks*). If *backend* is *None*, the default backend will be used.

Parameters

- **hard_graph** (DepGraph) – The dependency graph for hard dependencies.
- **soft_graph** (*DepGraph or None*) – The dependency graph for soft dependencies.
- **backend** (*None or* QueueScheduling) – The scheduling backend.

Raises

- **ValueError** – if *depgraph* is not an instance of *DepGraph*.
- ***SchedulerError*** – if the tasks do not have any `do()` method.

__init__(*, *hard_graph*, *soft_graph=None*, *backend=None*)

Initialize the scheduler with a graph.

schedule(*, *config=None*, *env=None*)

Schedule the tasks!

Parameters

- **env** (Env) – An initial environment for the scheduled tasks. This allows completed tasks to inform other, dependent tasks of e.g. the location of interesting files.
- **config** (Config) – the configuration object.

Returns

The modified environment.

Scheduling backends

queue — Producer-consumer-queue backend

This module contains an implementation of a scheduling backend that leverages Python "threads" (`threading` module) and producer-consumer queues (`queue` module).

class valjean.cosette.backends.queue.**QueueScheduling**(*n_workers=10*)

The default scheduling backend.

Uses `threading` and `queue` to handle concurrent execution of tasks.

__init__(*n_workers=10*)

Initialize the queue backend.

Parameters

n_workers (*int*) – The number of worker threads to use.

classmethod decide_new_state(*task*, *deps*, *hard_deps*, *env*)

Look at the dependencies of the current task and make a decision about what we should do with this task.

This function returns a *TaskStatus* that represents the suggested new state for this task, or *None* in case the task should be ignored and removed from the queue.

Parameters

- **task** (Task) – the task
- **deps** (*list(*Task*)*) – the dependencies of *task*
- **hard_deps** (*list(*Task*)*) – the hard dependencies of *task*
- **env** (Env) – the environment.

Return type

TaskStatus or None

classmethod last_end_time(*tasks*, *env*)

Return the latest `end_clock` time of the given tasks, or *None* if some of the tasks do not have an `end_clock`.

classmethod decide_new_state_waiting(*task*, *deps*, *hard_deps*, *env*)

Decide what to do with a task that is in *WAITING* state.

execute_tasks(*, *full_graph*, *hard_graph*, *env*, *config*)

Execute the tasks.

Parameters

- **full_graph** (DepGraph) – Full dependency graph for the tasks, i.e. including both hard and soft dependencies.
- **hard_graph** (DepGraph) – Hard-dependency graph for the tasks.
- **env** (Config) – An initial environment for the scheduled tasks.
- **env** – The configuration object (for things like paths, etc.).

class WorkerThread(*queue*, *env*, *config*, *cond_var*)

Workhorse class for *QueueScheduling*. This class consumes (i.e. executes) tasks passed to it through the queue.

__init__(*queue*, *env*, *config*, *cond_var*)

Initialize the thread.

Parameters

- **queue** – The producer-consumer task queue.
- **env** – The execution environment for the tasks.
- **config** – The configuration for the tasks.
- **cond_var** – A condition variable to notify when we are finished running a task

run()

Main method: run this thread.

5.3.9 use — Integrate Python functions in DepGraph objects

This module provides a decorator that simplifies the integration of free Python functions in the *valjean* dependency graph.

An example

Let us consider the following simple function, that counts the number of characters in a file:

```
>>> def how_many_chars(filename):
...     return len(open(filename).read())
```

Suppose that we have a task that produces some text and we want to count the number of characters in the output. For this example, we will take the task to be:

```
>>> from valjean.cosette.run import RunTask
>>> cmd = ['echo', 'spam']
>>> run_task = RunTask.from_cli('run_task', cmd)
```

This task will produce exactly five characters of standard output[1], but assume we want to check this with our *how_many_chars* function. If the task is part of a dependency graph,

[1] The proof is left to the reader as an exercise.

we cannot call *how_many_chars* until the task has been executed, simply because its output (which is the input to *how_many_chars*) does not exist until then.

The solution is to bridge the gap using an *Use* object:

```
>>> from valjean.cosette.use import Use
>>> use_arg_echo = Use.from_func(func=how_many_chars, task=run_task)
```

Use lifts the naked function *how_many_chars* into the world of *valjean* dependency graphs. The *how_many_chars* function is wrapped in a *PythonTask* that can be retrieved with the *get_task* method:

```
>>> how_many_task = use_arg_echo.get_task()
>>> from valjean.cosette.pythontask import PythonTask
>>> isinstance(how_many_task, PythonTask)
True
```

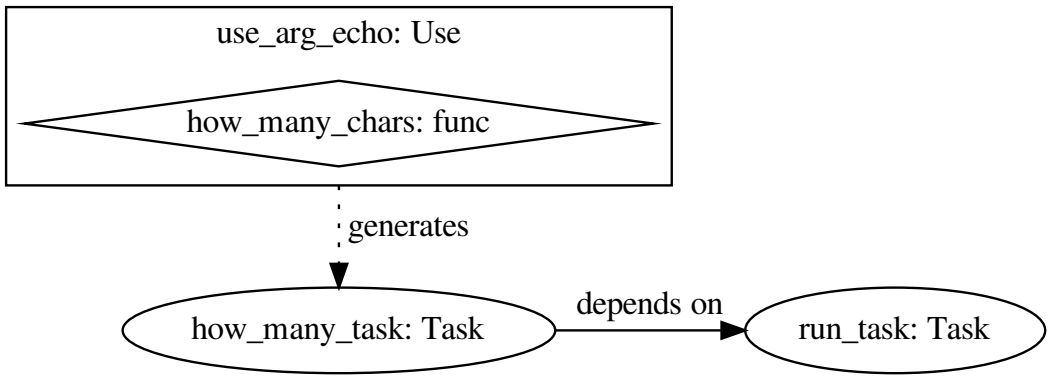
The *how_many_task* object is wired in such a way that it will retrieve the value to its argument from the `'stdout'` key of *run_task*'s environment section on execution. If *run_task* has not been executed at that point or it has failed, then of course *how_many_task* will fail, too. The dependency of *how_many_task* on *run_task* is made explicit:

```
>>> run_task in how_many_task.depends_on
True
```

Note: If you want to introduce a **soft** dependency on *run_task* (instead of a hard one), you can use the *deps_type* constructor argument:

```
>>> use_arg_echo_soft = Use.from_func(func=how_many_chars, task=run_task,
...                                   deps_type='soft')
>>> how_many_task_soft = use_arg_echo_soft.get_task()
>>> run_task in how_many_task_soft.soft_depends_on
True
```

This allows *valjean* to seamlessly integrate the execution of *how_many_task* in the dependency graph. We can illustrate the relations between the objects that we have created with the following diagram:



Here we see that *use_arg_echo* has type *Use* and contains the *how_many_chars* function. Additionally, it generates *how_many_task*, which is a *Task* object that depends on *run_task*.

We can test that *how_many_task* works by manually executing *run_task* first and passing the resulting environment to *how_many_task*:

```
>>> from valjean.cosette.env import Env
>>> env, status = run_task.do(env=Env(), config=config)
>>> print(status)
TaskStatus.DONE
```

Note that the environment now contains the name of the file written by *run_task* in the `'stdout'` key:

```
>>> env[run_task.name]['stdout']
'/.../run_task/stdout'
```

We can now execute *how_many_task*:

```
>>> env_up, status = how_many_task.do(env=env, config=config)
>>> print(status)
TaskStatus.DONE
```

The `'result'` key now holds the return value of *how_many_chars*:

```
>>> env_up[how_many_task.name]['result']
5
```

Use objects are wrappers

Use also behaves as a wrapper of *how_many_chars*. You can still call the underlying wrapped function with explicit parameters if you wish:

```
>>> with open('test.txt', 'w') as f:
...     _ = f.write('lobster Thermidor\n')
>>> use_arg_echo('test.txt')
18
>>> len('lobster Thermidor\n')
18
```

This simplifies testing.

Argument injection via the `using` decorator

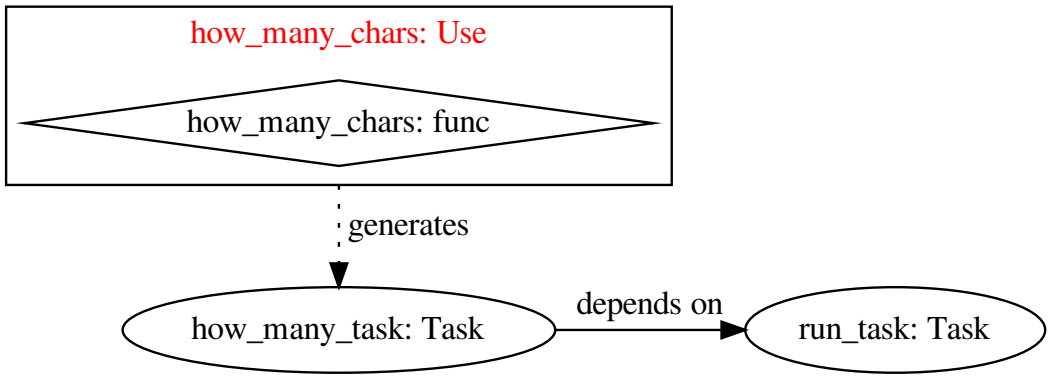
A practical way to create *Use* objects is to call the *using* decorator:

```
>>> @using(task=run_task)
... def how_many_chars(filename):
...     return len(open(filename).read())
```

Now *how_many_chars* is itself a *Use* object:

```
>>> isinstance(how_many_chars, Use)
True
```

This is exactly equivalent to what we did earlier, except for the fact that the *Use* object now does not have a different name:



You can still call the underlying *how_many_chars* by calling the *Use* object as if it were a normal function, as discussed above. Just like before, we can generate a task from the *Use* object:

```
>>> how_many_task = how_many_chars.get_task()
>>> isinstance(how_many_task, PythonTask)
True
```

Injecting multiple arguments

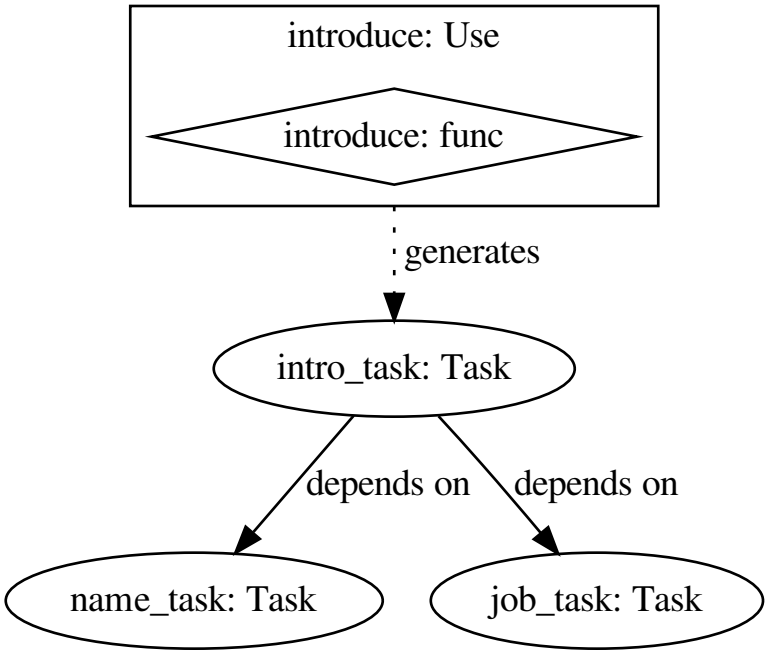
If your function expects multiple arguments to be fed from previous tasks, you can stack multiple *Use* decorators:

```
>>> name_task = RunTask.from_cli('name_task', ['echo', 'Arthur'])
>>> job_task = RunTask.from_cli('job_task',
...                             ['echo', 'King of the Britons'])
>>> @using(task=name_task)
... @using(task=job_task)
... def introduce(name_filename, job_filename):
...     with open(name_filename) as name_file:
...         name = name_file.read().rstrip()
...     with open(job_filename) as job_file:
...         job = job_file.read().rstrip()
...     return 'I am ' + name + ', ' + job + '!'
```

The resulting task has the correct dependencies, i.e. it depends on both *name_task* and *job_task*:

```
>>> intro_task = introduce.get_task()
>>> name_task in intro_task.depends_on
True
>>> job_task in intro_task.depends_on
True
```

We can represent the dependency structure of the generated tasks with the following graph:



If we manually execute *name_task* and *job_task*, we can then execute *intro_task*:

```
>>> env = Env()
>>> for task in [name_task, job_task, intro_task]:
...     env_up, _ = task.do(env=env, config=config)
...     env.apply(env_up)
>>> print(env[intro_task.name]['result'])
I am Arthur, King of the Britons!
```

Keyword arguments

In the previous example, note that the order of the decorators is important! The outer decorator (*name_task*) injects the first argument (*name_filename*), and the inner one (*job_task*) injects the second argument (*job_filename*). If we nest the decorators the other way around, we inverse the order of the arguments:

```
>>> @using(task=job_task)
... @using(task=name_task)
... def introduce_inv(name_filename, job_filename):
...     with open(name_filename) as name_file:
...         name = name_file.read().rstrip()
...     with open(job_filename) as job_file:
...         job = job_file.read().rstrip()
...     return 'I am ' + name + ', ' + job + '!'
>>> intro_inv_task = introduce_inv.get_task()
```

(continues on next page)

(continued from previous page)

```
>>> for task in [name_task, job_task, intro_inv_task]:
...     env_up, _ = task.do(env=env, config=config)
...     env.apply(env_up)
>>> print(env[intro_inv_task.name]['result'])
I am King of the Britons, Arthur!
```

If you are injecting many arguments, it may be a good idea to use the *kwarg* argument to *using*, which allows you to inject values into arguments by name instead of by position:

```
>>> @using(kwarg='name_filename', task=name_task)
... @using(kwarg='job_filename', task=job_task)
... def introduce_kwargs(name_filename, job_filename):
...     return introduce(name_filename, job_filename)
>>> @using(kwarg='job_filename', task=job_task)
... @using(kwarg='name_filename', task=name_task)
... def introduce_kwargs_swap(name_filename, job_filename):
...     return introduce(name_filename, job_filename)
```

Note that the *introduce_kwargs* and *introduce_kwargs_swap* functions are identical, but the *@using* decorators appear in reverse order. In the previous example, this mattered and we ended up with `I am King of the Britons, Arthur!`. But since now we are using the *kwarg* argument, the order of the decorators does not matter any more. Both *introduce_kwargs* and *introduce_kwargs_swap* will inject the arguments as expected:

```
>>> intro_kwargs_task = introduce_kwargs.get_task()
>>> intro_kwargs_swap_task = introduce_kwargs_swap.get_task()
>>> env_up, _ = intro_kwargs_task.do(env=env, config=config)
>>> print(env_up[intro_kwargs_task.name]['result'])
I am Arthur, King of the Britons!
>>> env_up, _ = intro_kwargs_swap_task.do(env=env, config=config)
>>> print(env_up[intro_kwargs_swap_task.name]['result'])
I am Arthur, King of the Britons!
```

Modifying an existing Use decorator

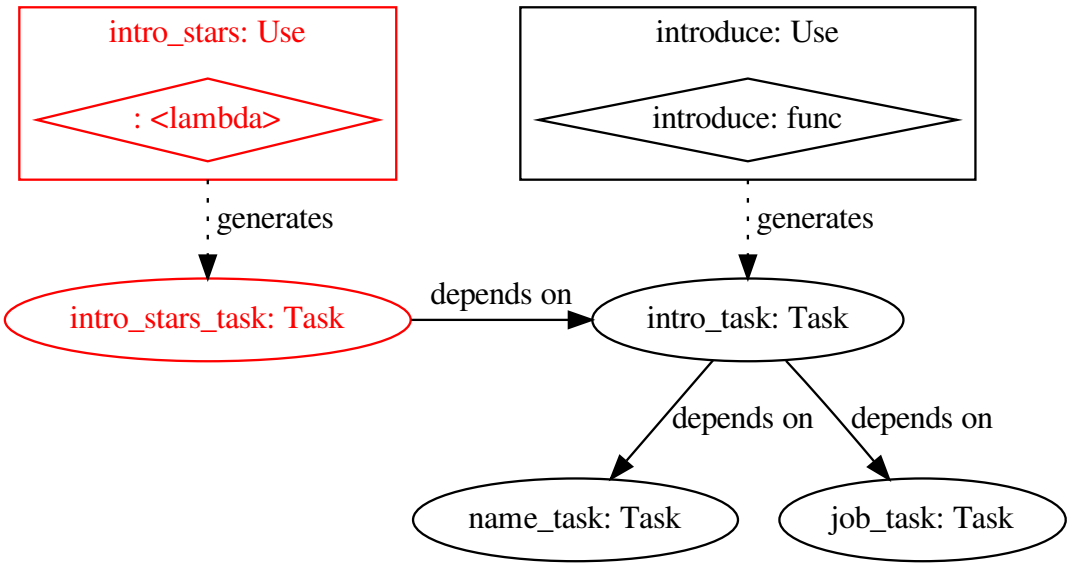
You can pipe the result of the task generated by a *Use* decorator by using the *Use.map* method:

```
>>> intro_stars = introduce.map(lambda x: '★★★' + x + '★★★')
```

Here *intro_stars* is a new *Use* object that will apply the specified function (here, a lambda that surrounds its argument with ★★★ cute little stars ★★★) to the result of *intro_task* (see above). The task associated to *intro_stars* depends on *intro_task*:

```
>>> intro_stars_task = intro_stars.get_task()
>>> intro_task in intro_stars_task.depends_on
True
```

In our schematic representation, *intro_stars* and *intro_stars_task* would appear like this:



If we execute *intro_stars_task*, we find that our string now has stars!

```
>>> env_up, _ = intro_stars_task.do(env=env, config=config)
>>> env.apply(env_up)
>>> print(env[intro_stars_task.name]['result'])
***I am Arthur, King of the Britons!***
```

Injecting arguments into tasks created by a RunTaskFactory

There is a typical use case for *Use* which consists in injecting the result of a *RunTask* generated by a *RunTaskFactory* as an argument to a subsequent task *B*. Since this use case is common, there is a special *UseRun* class that helps reducing the boilerplate.

First, we create a *RunTaskFactory* object:

```
>>> from valjean.cosette.run import RunTaskFactory
>>> factory = RunTaskFactory.from_executable('echo')
```

Instead of creating *RunTask* objects and wrapping them in *Use* by hand, we instantiate a *UseRun* object:

```
>>> using_run = UseRun.from_factory(factory)
```

The *UseRun* object will create the *RunTask* objects on the fly, when invoked as a decorator:

```
>>> @using_run(extra_args=['spam'])
... def read_and_check_text(filename):
...     with open(filename) as f:
...         return f.read() == 'spam\n'
```

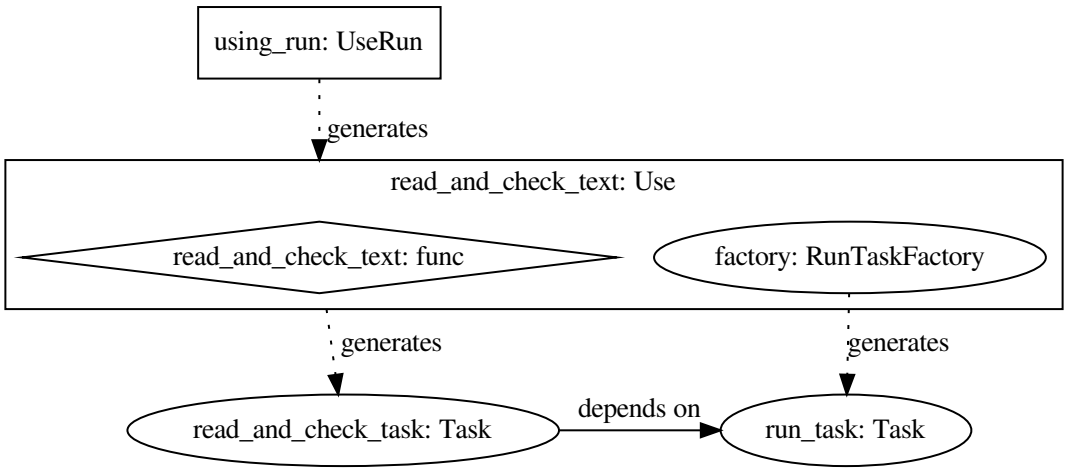
The decorated function is simply an instance of the *Use* class that we all know and love:

```
>>> isinstance(read_and_check_text, Use)
True
```

so we can call *Use.get_task* to generate a task:

```
>>> read_and_check_task = read_and_check_text.get_task()
>>> run_task = next(iter(read_and_check_task.depends_on))
```

Since a picture is worth a thousand words, the objects have the following structure:



Let's check that our tasks work as expected:

```
>>> env, _ = run_task.do(env=Env(), config=config)
>>> env, _ = read_and_check_task.do(env=env, config=config)
>>> print(env[read_and_check_task.name]['result'])
True
```

Pipelines

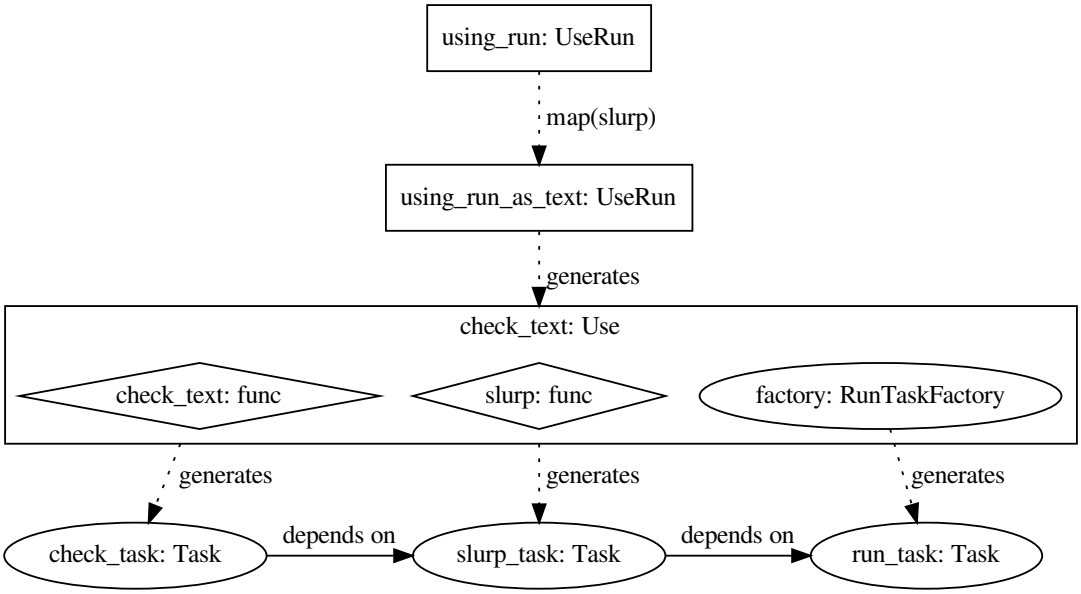
This is already nice! However, if you often want to perform the same additional actions after creating your *RunTask*, you can cut the boilerplate further with the *UseRun.map* method. In our example, we might want to factorize out the `with open(filename) as f` bit and focus in the text comparison:

```
>>> def slurp(filename):
...     with open(filename) as f:
...         return f.read()
>>> using_run_as_text = using_run.map(slurp)
```

The *UseRun.map* method creates a new *UseRun* object which will enqueue the specified function (as a task) to be executed on the result of the *RunTask*. We can simplify *read_and_check_text* from above as follows:

```
>>> @using_run_as_text(extra_args=['spam'])
... def check_text(text):
...     return text == 'spam\n'
```

The diagram now looks like this:



Checking that this works is a bit more laborious, because we now have one extra task to run; in the real world, of course, *valjean* would take care of running the tasks via a dependency graph and we wouldn't need to worry about any of this:

```
>>> check_task = check_text.get_task()
>>> slurp_task = next(iter(check_task.depends_on))
>>> run_task = next(iter(slurp_task.depends_on))
>>> env = Env()
>>> for task in [run_task, slurp_task, check_task]:
...     env_up, _ = task.do(env=env, config=config)
...     env.apply(env_up)
>>> print(env[check_task.name]['result'])
True
```

You can also chain *UseRun.map* calls:

```
>>> using_run_as_stripped_text = using_run.map(slurp) \
...                                        .map(lambda x: x.strip())
>>> @using_run_as_stripped_text(extra_args=['spam'])
... def check_stripped_text(text):
...     return text == 'spam'  # note that the \n is gone
```

Proof that it works:

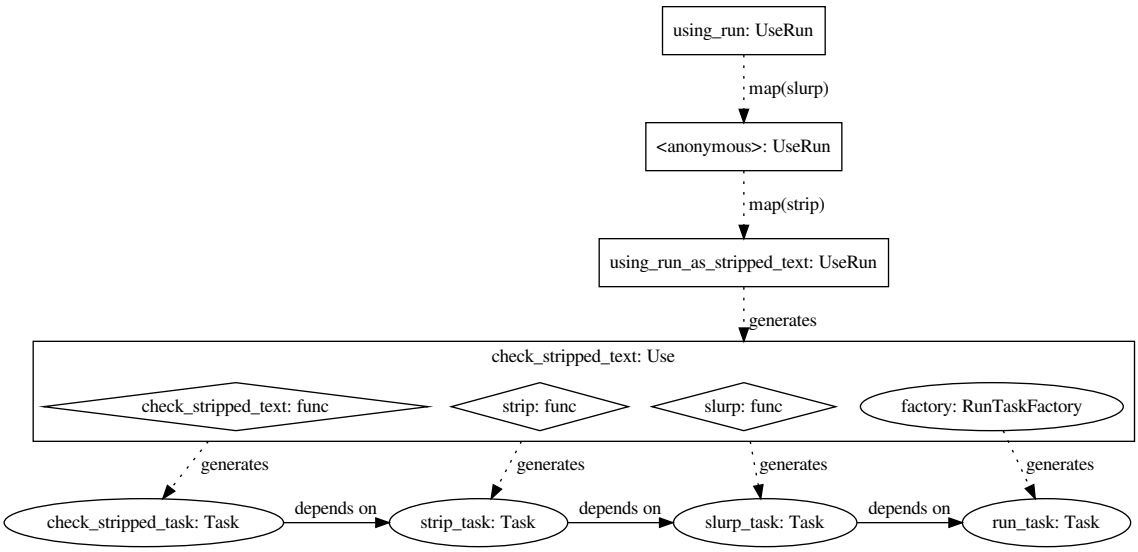
```
>>> check_stripped_task = check_stripped_text.get_task()
>>> strip_task = next(iter(check_stripped_task.depends_on))
```

(continues on next page)

(continued from previous page)

```
>>> slurp_task = next(iter(strip_task.depends_on))
>>> run_task = next(iter(slurp_task.depends_on))
>>> env = Env()
>>> for task in [run_task, slurp_task, strip_task, check_stripped_task]:
...     env_up, _ = task.do(env=env, config=config)
...     env.apply(env_up)
>>> print(env[check_stripped_task.name]['result'])
True
```

and a final diagram to show the dependencies among the objects:



Module API

valjean.cosette.use.**from_env**(*, *env*, *task_name*, *key*)

Helper function to extract a value from the environment for argument injection.

Parameters

- **env** (*Env*) – the environment.
- **task_name** (*str*) – the name of the task to look up.
- **key** (*str or None*) – the key to look up. If *None* is given, return the whole key-value pair associated to *task_name*.

Raises

KeyError – if the required task_name or key are not available.

class valjean.cosette.use.**Use**(*, *inj_args=None*, *inj_kwargs=None*, *wrapped*, *deps_type='hard'*, *serialize=False*)

A function wrapper around a free Python function. Lifts the function into a *valjean Task*.

classmethod from_func(*, *func*, *task*, *key='result'*, *kwarg=None*, *deps_type='hard'*, *serialize=False*)

Create a *Use* from a function.

Parameters

- **func** – a function or a callable object.
- **task** (*Task*) – the task whose result should be injected as an argument to *func*.
- **key** (*str*) – the name of the key that contains the task result in the environment.
- **kwarg** (*None or str*) – the name of the keyword argument to *func* that must be fed with the *task* result. If None, the result of *task* will be passed as a positional argument.
- **deps_type** (*str*) – whether the created task should have a hard or a soft dependency towards the injected tasks. Possible values are `'hard'` and `'soft'`.
- **serialize** (*bool*) – if *True*, the result of this task will be written to the output directory.

__init__(*, *inj_args=None*, *inj_kwargs=None*, *wrapped*, *deps_type='hard'*, *serialize=False*)

Instantiate a *Use* by providing all the necessary information.

Parameters

- **inj_args** (*tuple((Task, str or None)) or list((Task, str or None))*) – an iterable of *(task, key)* pairs specifying which part of what task result should be injected as a positional argument. See *Use* for the meaning of *key*.
- **inj_kwargs** (*dict(str, (Task, str or None))*) – a dictionary associating any kwarg names to a *(task, key)* pair. The specified task result will be injected as the given kwarg.
- **wrapped** – the function to wrap.
- **deps_type** (*str*) – whether the created task should have a hard or a soft dependency towards the injected tasks. Possible values are `'hard'` and `'soft'`.
- **serialize** (*bool*) – if *True*, the result of this task will be written to the output directory.

get_task()

Return the task that executes the decorated function.

__call__(**args*, ***kwargs*)

Redirect the call to the wrapped function.

map(*func*)

Create a new *Use* object that applies *func* to the result of the task defined by *self*.

valjean.cosette.use.**using**(*, *key='result'*, *task*, *kwarg=None*)

Make it possible to instantiate *Use* as a decorator.

See *Use.__init__* for a description of the parameters.

class valjean.cosette.use.**UseRun**(*factory*, *posts*)

Produce *Use* decorators from a *RunTaskFactory*.

classmethod from_factory(*factory*)

Create a *UseRun* from a *RunTaskFactory*.

Given a *RunTaskFactory*, the *UseRun* class can be used to construct *RunTask* objects on demand and inject their results into the decorated function.

Parameters
factory (RunTaskFactory) – a *RunTaskFactory* object.

__init__(*factory*, *posts*)

Instantiate a *UseRun* object.

Given a *RunTaskFactory*, the *UseRun* class can be used to construct *RunTask* objects on demand and inject their results into the decorated function.

The *posts* parameters is a list of functions that will be sequentially applied to the result of the generated *RunTask*. Each function becomes a new *PythonTask* object depending on the result of the previous one.

Parameters

- **factory** (RunTaskFactory) – a *RunTaskFactory* object.
- **posts** (*list*) – a collection of post-processing functions. Each function will be converted to a *PythonTask* object.

__call__(*kwarg=None*, ***kwargs*)

Call self as a function.

copy()

Return a copy of this object.

map(*func*)

Create a new instance of *UseRun* by extending the list of post-processing functions with a new one.

Parameters
func – a function with one argument to be applied to the result of *self*.

5.4 eponine — ExPloratiON et INtErfaçage

5.4.1 dataset – Unified data format

Common module to structure data in same way for all codes.

Fig. 4: Éponine, fille aînée des Thénardier, amoureuse de Marius, illustrée par Gustave Brion (2e édition ou première édition illustrée, entre 1862 et 1865 probablement)

Dataset definition and initialisation

A dataset is composed by 5 members:

- value: a `numpy.ndarray` or a `numpy.generic` (scalar from numpy represented like the arrays, with dim, etc)
- error: an object of same type as value
- bins: an `collections.OrderedDict` (optional and named argument)
- name: name of the dataset (optional and named argument)
- what: can be used to store the name of the quantity represented by the dataset (optional and named argument)

The bins object should have the same dimension as the value, the order matches the dimensions. If no bins are available it is still possible to use an empty `collections.OrderedDict`.

```
>>> from valjean.eponine.dataset import Dataset
>>> import numpy as np
>>> from collections import OrderedDict
>>> bins = OrderedDict([('e', np.array([1, 2, 3])), ('t', np.arange(5))])
>>> ds1 = Dataset(np.arange(10).reshape(2, 5),
...               np.array([0.3]*10).reshape(2, 5),
...               bins=bins, name='ds1', what='spam')
>>> ds1.name
'ds1'
>>> ds1.what
'spam'
>>> len(bins) == ds1.ndim
True
>>> ds1.error.shape == ds1.value.shape
True
>>> ds1.value.shape == (2, 5)
True
>>> np.array_equal(ds1.value, [[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]])
True
>>> np.array_equal(ds1.error,
...                [[0.3, 0.3, 0.3, 0.3, 0.3], [0.3, 0.3, 0.3, 0.3, 0.3]])
True
>>> list(bins.keys())
['e', 't']
>>> np.array_equal(bins['e'], [1, 2, 3])
True
>>> np.array_equal(bins['t'], [0, 1, 2, 3, 4])
True
```

A new dataset can also be created from an already existing one, using `copy`. No matter how the dataset is generated, attributes can be changed afterwards:

```
>>> nds = ds1.copy()
>>> nds.name = 'egg'
>>> nds.what = ''
>>> print(f'name: ds1={ds1.name!r}, nds={nds.name!r}')
name: ds1='ds1', nds='egg'
>>> print(f'what: ds1={ds1.what!r}, nds={nds.what!r}')
what: ds1='spam', nds=''
>>> np.array_equal(ds1.value, nds.value)
True
```

Errors are emitted if the arguments do not have the expected type or if the shapes or dimensions are not consistent:

```
>>> tds = Dataset([1, 2, 3], [0.5, 0.5, 0.5])
Traceback (most recent call last):
        [...]
TypeError: value does not have the expected type (numpy.ndarray or numpy.generic =␣
↪scalar)
```

```
>>> tds = Dataset(np.arange(6).reshape(2, 3), np.arange(6).reshape(3, 2))
Traceback (most recent call last):
        [...]
ValueError: Value and error do not have the same shape
```

```
>>> tds = Dataset(np.arange(6).reshape(2, 3),
...               np.array([0.5]*6).reshape(2, 3),
...               bins={'spam': [1, 2], 'egg': [1, 2, 3]})
Traceback (most recent call last):
        [...]
TypeError: bins should be an OrderedDict
```

```
>>> tds = Dataset(np.arange(6).reshape(2, 3),
...               np.array([0.5]*6).reshape(2, 3),
...               bins=OrderedDict([('spam', [1, 2])]))
Traceback (most recent call last):
        [...]
ValueError: Number of dimensions of bins does not correspond to number of dimensions␣
↪of value
```

Squeezing a dataset

If there are useless dimensions, the dataset can be squeezed:

```
>>> vals = np.arange(6).reshape(1, 2, 1, 3)
>>> errs = np.array([0.1]*6).reshape(1, 2, 1, 3)
>>> bins2 = OrderedDict([('bacon', np.array([0, 1])),
...                      ('egg', np.array([0, 2, 4])),
...                      ('sausage', np.array([10, 20])),
...                      ('spam', np.array([-5, 0, 5, 10]))])
>>> ds = Dataset(vals, errs, bins=bins2)
>>> ds.value.shape == (1, 2, 1, 3)
True
>>> len(ds.bins) == 4
True
>>> np.array_equal(ds.value, np.array([[[[0, 1, 2]], [[3, 4, 5]]]]))
True
>>> sds = ds.squeeze()
>>> sds.ndim
2
>>> len(sds.bins) == 2
True
>>> sds.shape
(2, 3)
>>> list(len(x)-1 for x in sds.bins.values())  # edges of bins, so N+1
[2, 3]
```

(continues on next page)

(continued from previous page)

```
>>> list(sds.bins.keys()) == ['egg', 'spam']
True
>>> np.array_equal(sds.value, np.array([[0, 1, 2], [3, 4, 5]]))
True
```

The dimensions with only one bin are squeezed, the same is done on the bins.

```
>>> nsds = ds.squeeze()
>>> np.array_equal(nsds.value, sds.value)
True
>>> nsds.bins == sds.bins
True
```

Operations available on Datasets

Standard operations, `(+, -, *, /, [])`, are available for *Dataset*. Examples of their use are given, including failing cases. Some are used in various methods but shown only once.

All operations conserve the name of the initial dataset.

The **what** attribute can be used to store the name of the quantity represented by the dataset. It is updated if it involves another dataset:

- in addition or substraction, if **what** is different it contains both separated by the symbol
- in multiplication or division it always contains both separeted by the symbol

Addition and subtraction

Addition and subtraction are possible between a *Dataset* and a scalar (`numpy.generic`), a `numpy.ndarray` and another *Dataset*. The operator to use for addition is `+` and for subtraction `-`.

Restrictions in addition or subtraction with a `numpy.ndarray` are handled by *NumPy*.

The addition or subtraction of two *Dataset* can be done if

- both values have the same shape (*consistent_datasets*)
- bins conditions (*same_coords*):
  - EITHER the second *Dataset* does not have any bins
  - OR bins are the same, i.e. have the same keys and bins values

Example addition of a scalar value (only on value)

```
>>> ds1p10 = ds1 + 10
>>> np.array_equal(ds1p10.value,
...                [[10, 11, 12, 13, 14], [15, 16, 17, 18, 19]])
True
>>> np.array_equal(ds1p10.error, ds1.error)
True
>>> ds1p10.bins == ds1.bins
```

(continues on next page)

(continued from previous page)

```
True
>>> ds1p10.what == ds1.what
True
```

As expected it 'only' acts on the value, error and bins are unchanged.

Example of subtraction of a `numpy.ndarray`

```
>>> a = np.array([100]*10).reshape(2, 5)
>>> ds1.value.shape == a.shape
True
>>> ds1ma = ds1 - a
>>> ds1ma.__class__
<class 'valjean.eponine.dataset.Dataset'>
>>> np.array_equal(ds1ma.value, [[-100, -99, -98, -97, -96],
...                              [-95, -94, -93, -92, -91]])
True
>>> np.array_equal(ds1ma.error, ds1.error)
True
>>> ds1ma.bins == ds1.bins
True
>>> ds1ma.what == ds1.what
True
```

`a` and `ds1` have the same shape to everything is fine.

```
>>> b = np.array([100]*10)
>>> ds1.value.shape == b.shape
False
>>> ds1pb = ds1 + b
Traceback (most recent call last):
    ...
ValueError: operands could not be broadcast together with shapes (2,5) (10,)
```

If the shapes are not the same *NumPy* raises an exception (and ds1pb is not defined).

Example of addition or subtraction of another `Dataset`

```
>>> ds2 = Dataset(value=np.arange(20, 30).reshape(2, 5),
...               error=np.array([0.4]*10).reshape(2, 5),
...               bins=bins, name='ds2', what='spam')
>>> ds1.bins == ds2.bins
True
>>> ds1p2 = ds1 + ds2
>>> ds1p2.name == ds1.name
True
>>> ds1p2.what == ds1.what
True
>>> np.array_equal(ds1p2.value, [[20, 22, 24, 26, 28],
...                              [30, 32, 34, 36, 38]])
True
>>> np.array_equal(ds1p2.error, np.array([0.5]*10).reshape(2, 5))
True
```

The error is calulated considering both datasets are independent, so quadratically ($e = \sqrt{ds1.e^2 + ds2.e^2}$).

Datasets without binning can also be added:

```
>>> ds3 = Dataset(value=np.arange(200, 300, 10).reshape(2, 5),
...               error=np.array([0.4]*10).reshape(2, 5), name='ds3')
>>> ds3.bins
OrderedDict()
>>> ds1p3 = ds1 + ds3
>>> ds1p3.name == ds1.name
True
>>> ds1p3.what == ds1.what
False
>>> ds1p3.what
'spam+'
>>> np.array_equal(ds1p3.value, [[200, 211, 222, 233, 244],
...                              [255, 266, 277, 288, 299]])
True
>>> np.array_equal(ds1p3.error, np.array([0.5]*10).reshape(2, 5))
True
>>> same_coords(ds1p3, ds1)
True
```

Bins of the dataset on the left are kept.

Like in *NumPy* array addition, values need to have the same shape:

```
>>> ds4 = Dataset(np.arange(5), np.array([0.01]*5), name='ds4')
>>> f"shape ds1 {ds1.value.shape}, ds4 {ds4.value.shape} -> comp = {ds1.value.shape␣
↪== ds4.value.shape}"
'shape ds1 (2, 5), ds4 (5,) -> comp = False'
>>> ds1 + ds4
Traceback (most recent call last):
    [...]
ValueError: Datasets to add do not have same shape
```

If bins are given, they need to have the same keys and the same values.

```
>>> bins5 = OrderedDict([('E', np.array([1, 2, 3])), ('t', np.arange(5))])
>>> ds5 = Dataset(np.arange(0, -10, -1).reshape(2, 5),
...               np.array([0.01]*10).reshape(2, 5),
...               bins=bins5, name='ds5')
>>> ds1 + ds5
Traceback (most recent call last):
    [...]
ValueError: Datasets to add do not have same bin names
>>> f"bins ds1: {list(ds1.bins.keys())}, bins ds5: {list(ds5.bins.keys())}"
"bins ds1: ['e', 't'], bins ds5: ['E', 't']"
```

```
>>> bins6 = OrderedDict([('e', np.array([1, 2, 30])), ('t', np.arange(5))])
>>> ds6 = Dataset(np.arange(0, -10, -1).reshape(2, 5),
...               np.array([0.01]*10).reshape(2, 5),
...               bins=bins6, name='ds6')
>>> ds1 - ds6
Traceback (most recent call last):
    [...]
ValueError: Datasets to subtract do not have the same bins
```

(continues on next page)

(continued from previous page)

```
>>> same_coords(ds1, ds6)
False
>>> list(ds1.bins.keys()) == list(ds6.bins.keys())
True
>>> np.array_equal(ds1.bins['e'], ds6.bins['e'])
False
>>> np.array_equal(ds1.bins['t'], ds6.bins['t'])
True
```

Multiplication and division

Multiplication and division are possible between a *Dataset* and a scalar (numpy.generic), a numpy.ndarray and another *Dataset*. The operator to use for multiplication is * and for division /.

Restrictions in multiplication and division with a numpy.ndarray are handled by *NumPy*.

The multiplication or division of 2 *Dataset* can be done if

- both values have the same shape (*consistent_datasets*)
- bins conditions (*same_coords*):
  - EITHER the second *Dataset* does not have any bins
  - OR bins are the same, i.e. have the same keys and bins values

Division by zero, nan or inf are handled by *NumPy* and return a RuntimeWarning from *NumPy* (only in the zero case).

Example multiplication of a scalar value

```
>>> ds1m10 = ds1 * 10
>>> ds1m10.__class__
<class 'valjean.eponine.dataset.Dataset'>
>>> np.array_equal(ds1m10.value, [[0, 10, 20, 30, 40],
...                               [50, 60, 70, 80, 90]])
True
>>> np.array_equal(ds1m10.error, np.array([3.]*10).reshape(2, 5))
True
>>> same_coords(ds1m10, ds1)
True
```

As expected it acts on the value and on the error. Bins are unchanged.

Example of division of a numpy.ndarray

```
>>> ds1da = ds1 / a
>>> ds1da.name == ds1.name
True
>>> ds1da.what == ds1.what
True
>>> np.array_equal(ds1da.value, [[0., 0.01, 0.02, 0.03, 0.04],
...                              [0.05, 0.06, 0.07, 0.08, 0.09]])
True
>>> np.array_equal(ds1da.error, np.array([0.003]*10).reshape(2, 5))
True
>>> same_coords(ds1da, ds1)
True
```

a and ds1 have the same shape to everything is fine.

```
>>> ds1 / b
Traceback (most recent call last):
    ...
ValueError: operands could not be broadcast together with shapes (2,5) (10,)
```

If the shapes are not the same *NumPy* raises an exception.

If the numpy.ndarray: contains 0, nan or inf, *NumPy* deals with them. It sends a RunningWarning about the division by zero.

```
>>> c = np.array([[2., 3., np.nan, 4., 0.], [1., np.inf, 5., 10., 0.]])
>>> ds1 / c
class: <class 'valjean.gavroche.dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        name: ds1, with shape (2, 5),
        value: [[0.         0.33333333        nan 0.75              inf]
 [5.         0.         1.4        0.8               inf]],
        error: [[0.15       0.03333333        nan 0.075             inf]
 [0.3        0.         0.06       0.03              inf]],
        bins: OrderedDict([('e', array([1, 2, 3])), ('t', array([0, 1, 2, 3, 4]))])
>>> # prints *** RuntimeWarning: divide by zero encountered in true_divide
```

Example of multiplication or division of another Dataset

```
>>> ds1m2 = ds1 * ds2
>>> ds1m2
class: <class 'valjean.gavroche.dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape (2, 5),
        value: [[  0  21  44  69  96]
 [125 156 189 224 261]],
        error: [[6.         6.31268564 6.64830806 7.00357052 7.37563557]
 [7.76208735 8.16088231 8.57029754 8.98888202 9.4154129 ]],
        bins: OrderedDict([('e', array([1, 2, 3])), ('t', array([0, 1, 2, 3, 4]))])
>>> ds1m2.what
'spam*spam'
>>> same_coords(ds1m2, ds2)
True
```

```
>>> ds1o2 = ds1 / ds2
>>> ds1o2
class: <class 'valjean.gavroche.dataset.Dataset'>, data type: <class 'numpy.ndarray'>
        shape (2, 5),
        value: [[0.         0.04761905 0.09090909 0.13043478 0.16666667]
 [0.2        0.23076923 0.25925926 0.28571429 0.31034483]],
        error: [[0.015      0.01431448 0.01373617 0.01323926 0.01280492]
 [0.01241934 0.01207231 0.01175624 0.01146541 0.0111955 ]],
        bins: OrderedDict([('e', array([1, 2, 3])), ('t', array([0, 1, 2, 3, 4]))])
>>> ds1o2.what
'spam/spam'
```

In both cases the error is calulated considering both datasets are independent, so quadratically $e = v * \sqrt{(\frac{ds_1.e}{ds_1.v})^2 + (\frac{ds_2.e}{ds_2.v})^2}$.

The same restictions on bins as for addition and subtraction are set for multiplication and division, same `AssertError` are raised, see *Example of addition or subtraction of another Dataset*.

About the division by `0`, `nan` or `inf`, it acts like in the multiplication or division by a `numpy.ndarray`, see *Example of division of a numpy.ndarray* (warnings and `nan`, `inf`, etc.)

```
>>> np.isnan((ds1/ds1).value[0][0])
True
>>> np.isinf(((ds1+1)/ds1).value[0][0])
True
```

Indexing and slicing

It is only possible to get a **slice** of a dataset, getting an Dataset at a given index is not possible (for dimensions consistency reasons). Requiring a given index can then be done using a slice.

Getting a subset of the `Dataset`

Time is the second dimension, to remove first and last bins the usual slice is `[1:-1]`, the first dimension, energy, is conserved, so its slice is `[:]`. The slice to apply is then `[:, 1:-1]`.

```
>>> ds1sltfl = ds1[:, 1:-1]
>>> ds1sltfl.__class__
<class 'valjean.eponine.dataset.Dataset'>
>>> np.array_equal(ds1sltfl.value, [[1, 2, 3], [6, 7, 8]])
True
>>> np.array_equal(ds1sltfl.error, np.array([0.3]*6).reshape(2, 3))
True
>>> list(ds1sltfl.bins.keys()) == list(ds1.bins.keys())
True
>>> np.array_equal(ds1sltfl.bins['e'], ds1.bins['e'])
True
>>> np.array_equal(ds1sltfl.bins['t'], ds1.bins['t'])
False
>>> np.array_equal(ds1sltfl.bins['t'], [1, 2, 3])
True
>>> ds1sltfl.name == ds1.name
True
```

Slicing is also applied on bins.

Warning: Requiring a slice when there are not enough elements on the dimension give empty arrays.

For example: removing first and last bin in energy on `ds1`. The slice is `[1:-1, :]` in that case, but `ds1` has only 2 bins in energy.

```
>>> ds1slefl = ds1[1:-1, :]
>>> ds1slefl.value.shape == (0, 5)
True
>>> np.array_equal(ds1slefl.value, np.array([]).reshape(0, 5))
True
>>> np.array_equal(ds1slefl.error, np.array([]).reshape(0, 5))
True
>>> np.array_equal(ds1slefl.bins['t'], ds1.bins['t'])
True
>>> np.array_equal(ds1slefl.bins['e'], ds1.bins['e'])
False
>>> np.array_equal(ds1slefl.bins['e'], [2])
True
```

Note that in this case, as bins are in reality the edges of the bins, so we have N+1 values in the bins compared to the value/error where we have N values. Slicing then give one value in energy bins, so unusable here (it would be empty if we have values and centers of bins instead of edges of bins).

All dimensions have to be present in the slice

Use `:` for the untouched dimensions. Number of `,` has to be dimension -1.

Let's consider a new Dataset, with 4 dimensions:

```
>>> bins2 = OrderedDict([('e', np.arange(4)), ('t', np.arange(3)),
...                      ('mu', np.arange(3)), ('phi', np.arange(5))])
>>> ds6 = Dataset(np.arange(48).reshape(3, 2, 2, 4),
...               np.array([0.5]*48).reshape(3, 2, 2, 4),
...               bins=bins2, name='ds6')
>>> ds6.value.ndim == 4
True
```

To remove first bin on energy dimension and last bin on phi dimension, the slice to be used is: `[1:, :, :, :-1]`.

```
>>> ds6_1 = ds6[1:, :, :, :-1]
>>> ds6.value.shape == (3, 2, 2, 4)
True
>>> ds6_1.value.shape == (2, 2, 2, 3)
True
>>> np.array_equal(ds6_1.value, [[[[16, 17, 18], [20, 21, 22]],
...                               [[24, 25, 26], [28, 29, 30]]],
...                              [[[32, 33, 34], [36, 37, 38]],
...                               [[40, 41, 42], [44, 45, 46]]]])
True
>>> np.array_equal(ds6_1.error, np.array([0.5]*24).reshape(2, 2, 2, 3))
```

(continues on next page)

(continued from previous page)

```
True
>>> list(ds6_1.bins.keys()) == list(ds6.bins.keys())
True
>>> np.array_equal(ds6_1.bins['e'], ds6.bins['e'][1:])
True
>>> np.array_equal(ds6_1.bins['t'], ds6.bins['t'])
True
>>> np.array_equal(ds6_1.bins['mu'], ds6.bins['mu'])
True
>>> np.array_equal(ds6_1.bins['phi'], ds6.bins['phi'][:-1])
True
```

If we only want the second bin in time keeping all bins in energy and direction angles, the slice is `[:, 1:2, :, :]`.

```
>>> ds6_2 = ds6[:, 1:2, :, :]
>>> ds6_2.value.shape == (3, 1, 2, 4)
True
>>> np.array_equal(ds6_2.value, [[[[8,  9, 10, 11], [12, 13, 14, 15]]],
...                              [[[24, 25, 26, 27], [28, 29, 30, 31]]],
...                              [[[40, 41, 42, 43], [44, 45, 46, 47]]]])
True
>>> list(ds6_2.bins.keys()) == list(ds6.bins.keys())
True
>>> all(x.size == y+1
...     for x, y in zip(ds6_2.bins.values(), ds6_2.value.shape)) == True
True
>>> np.array_equal(ds6_2.bins['t'], [1, 2])
True
```

Bins are changed accordingly.

Warning: Comparison to *NumPy*: index and ellipsis are other slicing possibilities on `numpy.ndarray` (see numpy indexing for current version of *NumPy*), but they are disabled here to avoid confusions. Errors are raised if required.

```
>>> ds1[1]
Traceback (most recent call last):
    [...]
TypeError: Index can only be a slice or a tuple of slices
```

```
>>> ds6_2e = ds6[:, 1, :, :]
Traceback (most recent call last):
    [...]
TypeError: Index can only be a slice or a tuple of slices
```

```
>>> ds6e = ds6[1:, ..., :-1]
Traceback (most recent call last):
    [...]
TypeError: Index can only be a slice or a tuple of slices
```

It also need to have the same dimension as the value:

```
>>> ds6_2e = ds6[:, 1:2]
Traceback (most recent call last):
    [...]
ValueError: len(index) should have the same dimension as the value numpy.ndarray, i.
↪e. (# ',' = dim-1).
':' can be used for a slice (dimension) not affected by the selection.
Slicing is only possible if ndim == 1
```

A single slice is only possible for arrays for dimension 1:

```
>>> ds6_2f = ds6[1:2]
Traceback (most recent call last):
    [...]
ValueError: len(index) should have the same dimension as the value numpy.ndarray, i.
↪e. (# ',' = dim-1).
':' can be used for a slice (dimension) not affected by the selection.
Slicing is only possible if ndim == 1
```

```
>>> ds7 = Dataset(value=np.arange(48), error=np.array([1]*48))
>>> ds7.ndim
1
>>> ds7.shape
(48,)
>>> ds7_extract = ds7[20:24]
>>> np.array_equal(ds7_extract.value, [20, 21, 22, 23])
True
```

Warning: Slicing can also only be applied on `numpy.ndarray`, not on `numpy.generic`:

```
>>> ds8 = Dataset(value=100, error=1)
>>> ds8[0:1]
Traceback (most recent call last):
    [...]
TypeError: [] (__getitem__) can only be applied on numpy.ndarrays
```

Masked datasets

In some case it can be useful to mask some elements of a dataset. This functionality is provided by the numpy masked arrays module. In the case of datasets, once the mask is given it is applied to both the value and the error.

```
>>> mask = np.ma.masked_greater(ds1.value, 6).mask
>>> np.array_equal(mask, [[False, False, False, False, False],
...                       [False, False, True, True, True]])
True
>>> mds = ds1.mask(mask)
>>> np.ma.is_masked(ds1.value)
False
>>> np.ma.is_masked(mds.value)
True
>>> np.ma.is_masked(mds.error)
True
```

Bins and shape are kept.

```
>>> mds.shape == ds1.shape
True
>>> np.array_equal(mds.bins['e'], ds1.bins['e'])
True
>>> np.array_equal(mds.bins['t'], ds1.bins['t'])
True
```

The mask is propagated when performing operations on dataset.

```
>>> np.sum(ds1.value) == 45
True
>>> np.sum(mds.value) == 21
True
>>> sds = ds1 + mds
>>> np.ma.is_masked(sds.value)
True
>>> np.ma.is_masked(sds.error)
True
>>> np.sum(sds.value) == 42
True
```

class valjean.eponine.dataset.**Dataset**(*value*, *error*, *, *bins=None*, *name=''*, *what=''*)

Common class for data from all codes.

For the moment, units are not treated (removed).

Todo: Think about units. Possibility: using a units package from scipy.

Todo: How to deal with bins of N values (= center of bins)

__init__(*value*, *error*, *, *bins=None*, *name=''*, *what=''*)

Dataset class initialization.

Parameters

- **value** (*int or float or numpy.ndarray or numpy.generic*) – array of N dimensions representing the values
- **error** (*int or float or numpy.ndarray or numpy.generic*) – array of N dimensions representing the **absolute** errors
- **bins** (*collections.OrderedDict (str, numpy.ndarray)*) – bins corresponding to value (named optional parameter)
- **name** (*str*) – name of the dataset (used in test representation)
- **what** (*str*) – name of the quantity represented by the dataset

copy()

Return a deep copy of *self*.

__repr__()

Return repr(self).

__str__()

Return str(self).

squeeze()

Squeeze dataset: remove useless dimensions.

Squeeze is based on the shape and on the bins dim ??? To confirm... First squeeze bins, then arrays. Edges, if only one bin are not kept. Example: spectrum with one bin in energy (quite common)

property shape

Return the data shape, as a read-only property.

property ndim

Return the data dimension, as a read-only property.

property size

Return the data size (total number of elements in the array), as a read-only property.

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

```
>>> from valjean.fingerprint import fingerprint
>>> vals = np.arange(10).reshape(2, 5)
>>> errs = np.array([1]*10).reshape(2, 5)
>>> ds = Dataset(vals, errs)
>>> fingerprint(ds)
'a8411470d7766c543e90f0f38241dc918b9448d1b9d19b0a9b8b6c91f61944d0'
>>> bins = OrderedDict([('bacon', np.arange(3)),
...                     ('egg', np.arange(5))])
>>> ds = Dataset(vals, errs, bins=bins)
>>> fingerprint(ds)
'dfec4e6ac118d30b7a83cfc500fefaec94f93a76a024d7550732cd08fbfb0fee'
>>> ds = Dataset(vals, errs, bins=bins, name='name')
>>> fingerprint(ds)
'441623c096bf917414013ebbfe345c66124f9a8dfbbd7051cf733ea20d7b651a'
>>> ds = Dataset(vals, errs, bins=bins, name='name', what='what!')
>>> fingerprint(ds)
'9a42c91381d696c1af651665f84efc43efa14d9c471c4ac851239ed0779a48a0'
```

mask(*mask*)

Apply a mask to the Dataset value and error.

Value and error become in that case masked_arrays instead of usual arrays, calcuations are preserved, not using the masked elements.

Return type
Dataset

valjean.eponine.dataset.**consistent_datasets**(*dss1*, *dss2*)

Return *True* if datasets are consistent = same shape.

valjean.eponine.dataset.**same_coords**(*ds1*, *ds2*)

Return *True* if coordinates (bins) are compatible.

Parameters

- **ds1** – the first array of coordinate arrays.
- **ds2** – the second array of coordinate arrays.

Comparison on keys and values.

5.4.2 `browser` – Indexing results stored as lists of dictionaries

Module to access in easy way results stored in list of dictionaries.

This module is composed of 2 classes:

- *Browser* that stores the list of dictionaries and builds an *Index* to facilitate selections;
- *Index* based on `collections.defaultdict` to perform selections on the list of dictionaries

The classes *Index* and *Browser* are meant to be general even if they will be shown and used in a specific case: parsing results from Tripoli-4.

The `Index` class

This class is based on an inheritance from `collections.abc.Mapping` from `collections`. It implements a `defaultdict(defaultdict(set))` from `collections.defaultdict`.

`set` contains *int* that corresponds to the index of the dictionary in the list of dictionaries.

Index is not supposed to be used standalone, but called from *Browser*, but this is still possible.

The `Browser` class

This class is analogue to a phonebook: it contains an index and the content, here stored as a list of dictionaries. It commands the index (building and selections). Examples are shown below.

Building the browser

Let's consider a bunch of friends going to the restaurant and ordering their menus. For each of them the waiter has to remember their name, under `'consumer'`, their choice of menu under `'menu'`, their drink, what they precisely order as dish under `'results'` and optionally the number corresponding to their choice of dessert. He will represent these orders as a list of orders, one order being a dictionary.

```
>>> from valjean.eponine.browser import Browser
>>> from pprint import pprint
>>> orders = [
... {'menu': '1', 'consumer': 'Terry', 'drink': 'beer',
...  'results': {'ingredients_res': ['egg', 'bacon']}},
... {'menu': '2', 'consumer': 'John',
...  'results': [{'ingredients_res': ['egg', 'spam']},
...              {'ingredients_res': ['tomato', 'spam', 'bacon']}]},
... {'menu': '1', 'consumer': 'Graham', 'drink': 'coffee',
...  'results': [{'ingredients_res': ['spam', 'egg', 'spam']}]},
... {'menu': '3', 'consumer': 'Eric', 'drink': 'beer',
...  'results': {'ingredients_res': ['sausage'],
...              'side_res': 'baked beans'}},
... {'menu': 'royal', 'consumer': 'Michael', 'drink': 'brandy', 'dessert': 3,
...  'results': {'dish_res': ['lobster thermidor', 'Mornay sauce']}}]
>>> com_br = Browser(orders)
>>> print(com_br)
```

(continues on next page)

(continued from previous page)

```
Browser object -> Number of content items: 5, data key: 'results', available metadata␣
↪keys: ['consumer', 'dessert', 'drink', 'index', 'menu']
               -> Number of globals: 0
```

Some global variables can be added, as a dictionary, normally common to all the results sent under the argument `content`.

```
>>> global_vars = {'table': 42, 'service_time': 300, 'priority': -1}
>>> com_br = Browser(orders, global_vars=global_vars)
>>> print(com_br)
Browser object -> Number of content items: 5, data key: 'results', available metadata␣
↪keys: ['consumer', 'dessert', 'drink', 'index', 'menu']
               -> Number of globals: 3
>>> pprint(com_br.globals)
{'priority': -1, 'service_time': 300, 'table': 42}
```

Selection of a given items or of a list of items from content

Various methods are available to select one order, depending on requirements:

- get a new Browser:

```
>>> sel_br = com_br.filter_by(menu='1', drink='beer')
>>> pprint(sel_br.content)
[{'consumer': 'Terry',  'drink': 'beer', 'index': 0, 'menu': '1', 'results': {
↪'ingredients_res': ['egg', 'bacon']}}]
```

  - check if a key is present or not:

```
>>> 'drink' in sel_br
True
>>> 'dessert' in sel_br
False
>>> 'dessert' in com_br
True
```

  The `'dessert'` key has been removed from the Browser issued from the selection while it is still present in the original one.

- get the available keys (sorted to be able to test them in the doctest, else list is enough):

```
>>> sorted(sel_br.keys())
['consumer', 'drink', 'index', 'menu']
>>> sorted(com_br.keys())
['consumer', 'dessert', 'drink', 'index', 'menu']
```

- if the required key doesn't exist a warning is emitted:

```
>>> sel_br = com_br.filter_by(quantity=5)
>>> # prints  WARNING     browser: quantity not a valid key. Possible ones are [
↪'consumer', 'dessert', 'drink', 'index', 'menu']
>>> 'quantity' in com_br
False
```

- if the value corresponding to the key doesn't exist another warning is emitted:

```
>>> sel_br = com_br.filter_by(drink='wine')
>>> # prints  WARNING     browser: wine is not a valid drink
```

- to know the available values corresponding to the keys (without the corresponding items indexes):

```
>>> sorted(com_br.available_values('drink'))
['beer', 'brandy', 'coffee']
```

- if the key doesn't exist an 'empty generator' is emitted:

```
>>> sorted(com_br.available_values('quantity'))
[]
```

- to directly get the content items corresponding to the selection, use the method *Browser.select_by*

```
>>> sel_br = com_br.select_by(consumer='Graham')
>>> type(sel_br)
<class 'dict'>
>>> len(sel_br)
5
>>> pprint(sel_br)
{'consumer': 'Graham', 'drink': 'coffee', 'index': 2, 'menu': '1', 'results': [{
 ↪'ingredients_res': ['spam', 'egg', 'spam']}]}
```

- this does not work when several items correspond to the selection:

```
>>> sel_br = com_br.select_by(drink='beer')
Traceback (most recent call last):
        [...]
valjean.eponine.browser.TooManyItemsBrowserError: Several content items␣
 ↪correspond to your choice, please refine your selection using additional␣
 ↪keywords
```

- if no item corresponds to the selection another exception is thrown:

```
>>> sel_br = com_br.select_by(menu='4')
Traceback (most recent call last):
        [...]
valjean.eponine.browser.NoItemBrowserError: No item corresponding to the␣
 ↪selection.
```

Module API

class valjean.eponine.browser.**Index**

Class to describe index used in Browser.

The structure of Index is a `defaultdict(defaultdict(set))`. This class was derived mainly for pretty-printing purposes.

Quick example of index (menu for 4 persons, identified by numbers, one has no drink):

```
>>> from valjean.eponine.browser import Index
>>> myindex = Index()
```

(continues on next page)

(continued from previous page)

```
>>> myindex.index['drink']['beer'] = {1, 4}
>>> myindex.index['drink']['wine'] = {2}
>>> myindex.index['menu']['spam'] = {1, 3}
>>> myindex.index['menu']['egg'] = {2}
>>> myindex.index['menu']['bacon'] = {4}
>>> myindex.dump(sort=True)
"{'drink': {'beer': {1, 4}, 'wine': {2}}, 'menu': {'bacon': {4}, 'egg': {2}, 'spam
↪': {1, 3}}}"
>>> 'drink' in myindex
True
>>> 'consumer' in myindex
False
>>> len(myindex)
2
>>> for k in sorted(myindex):
...     print(k, sorted(myindex[k].keys()))
drink ['beer', 'wine']
menu ['bacon', 'egg', 'spam']
```

The *keep_only* method allows to get a sub-Index from a given set of ids (int), removing all keys not involved in the corresponding ids:

```
>>> myindex.keep_only({2}).dump(sort=True)
"{'drink': {'wine': {2}}, 'menu': {'egg': {2}}}"
>>> menu_clients14 = myindex.keep_only({1, 4})
>>> sorted(menu_clients14.keys()) == ['drink', 'menu']
True
>>> list(menu_clients14['drink'].keys()) == ['beer']
True
>>> list(menu_clients14['drink'].values()) == [{1, 4}]
True
>>> sorted(menu_clients14['menu'].keys()) == ['bacon', 'spam']
True
>>> menu_clients14['menu']['spam'] == {1}
True
>>> 3 in menu_clients14['menu']['spam']
False
>>> menu_client3 = myindex.keep_only({3})
>>> list(menu_client3.keys()) == ['menu']
True
>>> 'drink' in menu_client3
False
```

The key `'drink'` has been removed from the last index as 2 did not required it.

If you print an *Index*, it looks like a standard dictionary (`{ ... }` instead of `defaultdict(...)`) but the keys are not sorted:

```
>>> print(myindex)
{...: {...: {...}...}}
```

__init__()

__str__()

Return str(self).

__repr__()

Return repr(self).

__getitem__(*item*)

__len__()

__iter__()

__contains__(*key*)

keep_only(*ids*)

Get an *Index* containing only the relevant keywords for the required set of ids.

Parameters

ids (*set(int)*) – index corresponding to the required elements of the list of content items

Returns

Index only containing the keys involved in the ids

dump(*, *sort=False*)

Dump the Index.

If `sort == False` (default case), returns *__str__* result, else returns sorted Index (alphabetic order for keys).

__abstractmethods__ = frozenset({})

class valjean.eponine.browser.**Browser**(*content*, *data_key='results'*, *global_vars=None*)

Class to perform selections on results.

This class is based on four objects:

- the content, as a list of dictionaries (containing data and metadata)
- the key corresponding to data in the dictionary (default='results')
- an index based on content elements allowing easy selections on each metadata
- a dictionary corresponding to global variables (common to all content items).

Initialization parameters:

Parameters

- **content** (*list(dict)*) – list of items containing data and metadata
- **data_key** (*str*) – key in the content items corresponding to results or data, that should not be used in index (as always present and mandatory)
- **global_vars** (*dict*) – global variables (optional, default=None)

An additional key is added at the Index construction: `'index'` in order to keep track of the order of the list and being able to do selection on it.

Examples on development / debugging methods:

Let's use the example detailled above in *module introduction*:

```
>>> from valjean.eponine.browser import Browser
>>> orders = [
... {'menu': '1', 'consumer': 'Terry', 'drink': 'beer',
...  'results': {'ingredients_res': ['egg', 'bacon']}},
... {'menu': '2', 'consumer': 'John',
...  'results': [{'ingredients_res': ['egg', 'spam']},
...              {'ingredients_res': ['tomato', 'spam', 'bacon']}]},
... {'menu': '1', 'consumer': 'Graham', 'drink': 'coffee',
...  'results': [{'ingredients_res': ['spam', 'egg', 'spam']}]},
... {'menu': '3', 'consumer': 'Eric', 'drink': 'beer',
...  'results': {'ingredients_res': ['sausage'],
...              'side_res': 'baked beans'}},
... {'menu': 'royal', 'consumer': 'Michael', 'drink': 'brandy',
...  'dessert': 3,
...  'results': {'dish_res': ['lobster thermidor', 'Mornay sauce']}}]
>>> com_br = Browser(orders)
```

- possibility to get the item id directly (internally used method):

```
>>> ind = com_br._filter_items_id_by(drink='coffee')
>>> isinstance(ind, set)
True
>>> print(ind)
{2}
```

- possibility to get the index of the content element stripped without rebuilding the full Browser:

```
>>> ind = com_br._filter_index_by(menu='1')
>>> isinstance(ind, Index)
True
>>> ind.dump(sort=True)
"{'consumer': {'Graham': {2}, 'Terry': {0}}, 'drink': {'beer': {0}, 'coffee':
↪{2}}, 'index': {0: {0}, 2: {2}}, 'menu': {'1': {0, 2}}}"
```

  The 'dessert' key has been stripped from the index:

```
>>> 'dessert' in ind
False
```

Debug print is available thanks to *__repr__*:

```
>>> small_order = [{'dessert': 1, 'drink': 'beer', 'results': ['spam']}]
>>> so_br = Browser(small_order)
>>> f"{so_br!r}"
"<class 'valjean.eponine.browser.Browser'>, (Content items: ..., Index: ...)"
```

__init__(*content*, *data_key='results'*, *global_vars=None*)

__eq__(*other*)

Return self==value.

__ne__(*other*)

Return self!=value.

__contains__(*key*)

__len__()

is_empty()

Check if the Browser is empty or not.

Empty meaning no elements in content AND no globals.

merge(*other*)

Merge two browsers.

This method merge 2 browsers: the *other* one appears then at the end of the *self* one. Global variables are also merged. The new index correspond to the merged case.

Parameters
other (Browser) – another browser

Return type
Browser

keys()

Get the available keys in the index (so in the items list). As usual it returns a generator.

available_values(*key*)

Get the available keys in the *second* level, i.e. under the given external one.

Parameters
key (*str*) – 'external' key (from outer defaultdict)

Returns
generator with corresponding keys (or empty one)

__str__()

Return str(self).

__repr__()

Return repr(self).

filter_by(*include=()*, *exclude=()*, ***kwargs*)

Get a Browser corresponding to selection from keyword arguments.

Parameters

- ****kwargs** – keyword arguments to specify the required item. More than one are allowed.
- **include** (*tuple(str)*) – metadata keys required in the content items but for which the value is not necessarly known
- **exclude** (*tuple(str)*) – metadata that should not be present in the items and for which the value is not necessarly known

Returns
Browser (subset of the default one, corresponding to the selection)

select_by(*, *include=()*, *exclude=()*, ***kwargs*)

Get an item or the list of items from content corresponding to selection from keyword arguments.

Parameters

- ****kwargs** – keyword arguments to specify the required items. More than one are allowed.
- **include** (*tuple(str)*) – metadata keys required in the items but for which the value is not necessarly known
- **exclude** (*tuple(str)*) – metadata that should not be present in the items and for which the value is not necessarly known

Raises

- ***NoItemBrowserError*** – if no item corresponds to the selection
- ***TooManyItemsBrowserError*** – if more than one item corresponds to the provided keywords

Return type
dict

__abstractmethods__ = frozenset({})

__hash__ = None

exception valjean.eponine.browser.**TooManyItemsBrowserError**
Error to *Browser* when too many items correspond to the requested selection.

exception valjean.eponine.browser.**NoItemBrowserError**
Error to *Browser* when no item corresponds to the selection.

5.4.3 tripoli4 – Parsing for Tripoli-4

Introduction

The goal of this sub-package is to parse Tripoli-4 results giving them an easy access and transforming them in the standard data format of *valjean*.

It contains various modules from scanning the Tripoli-4 output to conversion of its data in datasets, listed here starting from the most external modules to the innest ones:

- Integration of parsing in the *valjean* workflow is handled by *use*;
- Conversion of data to *Dataset* in *data_convertor*;
- Parsing module *parse* calling the scanning one *scan*;
- The parsing itself is done thanks to 3 modules:
  - *grammar*: the grammar of the Tripoli-4 ASCII output using *pyparsing*;
  - *transform*: methods to transform the parsing result in standard *python* and *NumPy* objects called in *grammar* via `setParseAction` applied on the `pyparsing.ParseResults`;
  - *common*: methods to transform Tripoli-4 data in *NumPy* objects.
  - One additional module parsing module is available for debugging and development: *parse_debug*
- Access to Tripoli-4 results from depletion, so reading ROOT files is handled by *depletion*.

Main modules

data_convertor – Convert Tripoli-4 results to Dataset

This module converts Tripoli-4 data from parsing in *Dataset*.

valjean.eponine.tripoli4.data_convertor.**bins_reduction**(*def_bins*, *reduced_dims*)

Reduce number of bins for integrated results (on one or more dimension).

For the reduced dimensions, all the bounds are suppressed except the first and the last one. This is also the case if they represent center of bins. The dimension is supposed to be squeezed afterwards.

Parameters

- **def_bins** (collections.OrderedDict (str, numpy.ndarray)) – default bins (all of them)
- **list(bool)** – dimensions to be reduced or not

Returns

collections.OrderedDict (str, numpy.ndarray), i.e. adapted bins

```
>>> ibins = OrderedDict([('a', np.array([0, 1, 2, 3, 4])),
...                      ('b', np.array([-10, 0, 10]))])
>>> dshape_1 = (5, 3)
>>> b11 = bins_reduction(ibins, [True, True])
>>> b11['a']
array([0, 4])
>>> b11['b']
array([-10,  10])
>>> b51 = bins_reduction(ibins, [False, True])
>>> b51['a']
array([0, 1, 2, 3, 4])
>>> b51['b']
array([-10,  10])
```

valjean.eponine.tripoli4.data_convertor.**special_array**(*array*, *score*, *bins*, *array_key='array'*, *name=''*, *what=''*)

Convert special arrays in *Dataset*.

These special cases are typically vov results or uncertainty spectrum associated to a perturbation result. No error is associated to the score in that case. Bins are given by the spectrum.

Parameters

- **array** (*dict*) – result
- **score** (*str*) – key of the score to get
- **bins** (*collections.OrderedDict*) – bins correspondig to required array
- **array_key** (*str*) – default=``'array'`` but it can be an integrated array for example, should be a key inside farray_res
- **name** (*str*) – name of dataset
- **what** (*str*) – what attribute of dataset

Return type
Dataset

valjean.eponine.tripoli4.data_convertor.**array_result**(*farray_res*, *res_type*, *name=''*, *what=''*, *array_key='array'*, *score='score'*)

Conversion of arrays in *Dataset*.

Parameters

- **farray_res** (*dict*) – results dictionary containing `res_type` key
- **res_type** (*str*) – result type, like `'spectrum'`, `'mesh'`
- **array_key** (*str*) – default=``'array'`` but it can be an integrated array for example, should be a key inside `farray_res`
- **name** (*str*) – name of dataset
- **what** (*str*) – what attribute of dataset

Returns
Dataset

valjean.eponine.tripoli4.data_convertor.**masked_array_result**(*farray_res*, *res_type*, *name=''*, *what=''*, *array_key='array'*, *score='score'*)

Mask invalid (`np.nan` and `np.inf`) values.

valjean.eponine.tripoli4.data_convertor.**integrated_result**(*result*, *res_type='integrated'*, *name=''*, *what=''*, *score='score'*, *sigma='sigma'*)

Conversion of generic score (or energy integrated result) in *Dataset*.

Bins are squeezed according to the integrated dimension obtained from another array stored in the same result. Default full arrays are *'spectrum'* and *'mesh'*. A case is also foreseen for uncertainty results (in perturbation cases).

Parameters

- **result** (*dict*) – results dictionary containing `res_type` key
- **res_type** (*str*) – should be 'integrated'
- **name** (*str*) – name of dataset
- **what** (*str*) – what attribute of dataset
- **score** (*str*) – score key in results array like *'score'* or *'score/lethargy'*
- **sigma** (*str*) – sigma key in results array, usually *'sigma'*

Returns
Dataset

valjean.eponine.tripoli4.data_convertor.**results_wo_error**(*result*, *res_type*, *name=''*, *what=''*, *score=None*)

Conversion of results without error, allowing multiple scores in the same container.

Dict result
results dictionary containing `res_type` key

Parameters

- **res_type** (*str*) – key in result allowing access to the desired quantity
- **name** (*str*) – name of dataset
- **what** (*str*) – what attribute of dataset
- **score** (*str*) – score key in results array like *'score'*

Returns
Dataset

`valjean.eponine.tripoli4.data_convertor.`**`result_wo_error`**(*result*, *res_type*, *name=''*, *what=''*)

Conversion of a result without error.

The result can be of any dimension (matrix, vector, scalar) or any type (float, int, complex).

Dict result
results dictionary containing `res_type` key

Parameters

- **res_type** (*str*) – key in result allowing access to the desired quantity
- **name** (*str*) – name of dataset
- **what** (*str*) – what attribute of dataset

Returns
Dataset

`valjean.eponine.tripoli4.data_convertor.`**`result_with_error`**(*result*, *res_type*, *name=''*, *what=''*, *score='score'*, *sigma='sigma'*)

Conversion of generic score (or energy integrated result) in *Dataset*.

Parameters

- **result** (*dict*) – results dictionary containing `res_type` key
- **res_type** (*str*) – should be 'generic'
- **name** (*str*) – name of dataset
- **what** (*str*) – what attribute of dataset
- **score** (*str*) – score key in results array like *'score'* or *'score/lethargy'*
- **sigma** (*str*) – sigma key in results array, usually *'sigma'*

Returns
Dataset

`valjean.eponine.tripoli4.data_convertor.`**`unbinned_result`**(*result*, *res_type*, *name=''*, *what=''*, *score=None*, *sigma=None*)

Conversion of all unbinned results in *Dataset*.

Dict result
results dictionary containing `res_type` key

Parameters

- **res_type** (*str*) – key in result allowing access to the desired quantity
- **name** (*str*) – name of dataset
- **what** (*str*) – what attribute of dataset
- **score** (*str*) – score key in results array like *'score'* or *'score/lethargy'*
- **sigma** (*str*) – sigma key in results array, usually *'sigma'*

Returns
Dataset

`valjean.eponine.tripoli4.data_convertor.`**`nan_result`**(*name=''*, *what=''*)

Returns a NaN *Dataset* (value and error).

This *Dataset* can be returned for example in case of non converged results or when no dataset can be built.

Values are scalar per default, not arrays.

Parameters

- **name** (*str*) – name of dataset
- **what** (*str*) – what attribute of dataset

Return type
Dataset

Returns
dataset containing NaN as value and error

`valjean.eponine.tripoli4.data_convertor.`**`convert_data`**(*data*, *data_type*, *name=''*, *what=''*, ***kwargs*)

Test for data conversion using dict or default.

An exception for integrated is for the moment needed as they can come from spectrum res or generic scores but are treated a bit differently. To be homogenized.

Parameters

- **data** (*dict*) – results dictionary containing `data_type` key
- **data_type** (*str*) – data key
- **kwargs** – keyword arguements if needed

Returns
Dataset

parse – Parse Tripoli-4 outputs

Module performing scanning and parsing of Tripoli-4 outputs.

This module also allows quick checks on outputs:

- presence of `"NORMAL COMPLETION"`
- presence and values of times (simulation, exploitation)

Some options for debugging are available (end flag).

Todo: Change absolute imports in relative ones when main will be moved to *cambronne*.

`valjean.eponine.tripoli4.parse.`**`profile`**(*fprof*)

To profile memory usage.

exception `valjean.eponine.tripoli4.parse.`**`ParserException`**

An error that may be raised by the *Parser* class.

class `valjean.eponine.tripoli4.parse.`**`Parser`**(*jddname*)

Scan Tripoli-4 listings, then parse the required batches.

`__init__`(*jddname*)

Initialize the *Parser* object.

Parameters
jddname (*str*) – path to the Tripoli-4 output

It also initalizes the result of *scan.Scanner* to `None`, then executes the scan. If this step fails an exception is raised.

The Parser main object instance variable is:

***scan_res* (*Scanner*)**
result from the scan of the Tripoli-4 output. See in the related documentation the various instance variables available (like `times`). Inheriting from `collections.abc.Mapping` various default methods are available like `len`, `[]`, etc. The keys of *Scanner* are the batch numbers available from the Tripoli-4 output. To get their list, use *batch_numbers*.

Parsing (e.g. *parse_from_number*) returns a *ParseResult*.

`batch_numbers`()

Help method to get the available batch numbers.

Return type
list(int)

`parse_from_number`(*batch_number*, *name*='')

Parse from batch index or batch number.

Parameters
batch_number (*int*) – number of the batch to parse

Return type
ParseResult

parse_from_index(*batch_index=-1*, *name=''*)

Parse from batch index or batch number.

Per default the last batch is parsed (index = -1).

Parameters
batch_index (*int*) – index of the batch in the list of batches

Return type
ParseResult

print_stats()

Print Tripoli-4 statistics (warnings and errors).

check_times()

Check if running times are well written in Tripoli-4 listings. These times are at the end of the result block and mark the end flag.

Returns
boolean, True if well present, else False

Returned bool depends on the listing:

- if the job was run in parallel mode (and declared so), `"simulation time"` and `"elapsed time"` should be present in that order, only the second one is checked
- if the listing name contains `"exploit"` or `"verif"`, it is most probably an exploitation job (Green bands for example), `"exploitation time"` is checked
- else `"simulation time"` is checked

print_times()

Print time characteristics of the Tripoli-4 result considered. This print includes initialization time, simulation time, exploitation time and elapsed time.

class valjean.eponine.tripoli4.parse.**ParseResult**(*parse_res*, *scan_vars*, *name=''*)

Class containing a parsing result from Tripoli-4 output for one batch.

The *ParseResult* object is accessible from the instance attribute

res
that is a unique dictionary containing all the results from scanning and parsing steps. Variables characteristic to a batch are stored under the key `'batch_data'` no matter if they come from *Scanner* or from *Parser*. Variables characteristic to a run (= one execution of Tripoli-4) are stored under `'run_data'`, coming from the scanning step.

It is possible to transform the `res` dictionary in a *Browser* thanks to the method *to_browser*.

__init__(*parse_res*, *scan_vars*, *name=''*)

Initialize the *ParseResult* from:

Parameters

- **parse_res** (*dict*) – result from T4 parsing (for 1 batch)
- **scan_vars** (*dict*) – variables coming from *Scanner* global to job or specific to the batch.
- **name** (*str*) – name to give to the parse result (will be propagated to data)

Fill the *res* object.

to_browser()

Get a *Browser* from the *ParseResult*.

The global variables in Browser are the batch data. You can access the *run data* only from the parsed result.

Return type
Browser

scan – Scan Tripoli-4 outputs and select relevant results

Module performing a scan of Tripoli-4 output listing in order to only keep relevant parts of it = results to be used for V&V or analysis or the run.

Summary

- Quickly reads the results file
- Recognize beginning and end of results sections
- Get the required number of batchs
- Get the edition batch numbers (if exists)

Use of scan

To use this module you need to create a *Scanner* object giving at least the path to the file you want to read. This file should be a Tripoli-4 output containing at least the flags precised in *Caveats*.

```
>>> import os
>>> from valjean.eponine.tripoli4.scan import Scanner
>>> results = Scanner(os.path.join(work_dir, 'spam.res'))
>>> results.normalend
True
>>> len(results)
1
>>> results.times['initialization_time']
7
```

The expected key of the *Scanner* object is the batch number, not an index. If you have the index you need to obtain the corresponding batch number first.

```
>>> 'simulation time' in results[10]  # batch number = 10
True
>>> 'simulation time' in results[-1]  # -1 can only be an index, like 0
Traceback (most recent call last):
    ...
KeyError: -1
>>> 'simulation time' in results[results.batch_number(-1)]
True
```

Note: It will probably be better to directly load a test file...

Caveats

Beginning and end of results sections

Important for the scan: results will be kept

- **from** "RESULTS ARE GIVEN"
- **to** an end flag available in the list `Scanner.end_flags`. Possibilities are:
  - Default end flag is `"simulation time"`;
  - For exploitation jobs use `"exploitation time"` (example: Green bands);
  - for jobs running in parallel, `"elapsed time"` will also appear, after `"simulation time"` normal.

Module API

valjean.eponine.tripoli4.scan.**profile**(*fmem*)

Deactivate profiling if not required in command line.

class valjean.eponine.tripoli4.scan.**PhEmEpBalanceOutput**

Class to store photon-electron-positron balance.

__init__()

add_line(*line*)

Add line to the photon electron positron output and counts.

class valjean.eponine.tripoli4.scan.**HomogMatOutput**

Class to store the homogenized material output.

__init__()

add_line(*line*)

Add line to the homogenized material output and count.

class valjean.eponine.tripoli4.scan.**BatchResultScanner**(*current_batch*, *para*, *line*)

Class to build batchs collection.

Parameters

- **current_batch** (*int*) – current batch number
- **para** (*bool*) – flag to identify outputs run in parallel
- **line** (*str*) – current line in file

Note:

- **current_batch** normally refers to lines before flag "RESULTS ARE GIVEN". They can be used if "Edition after batch number" does not appear in the file.
- **line** should contain "RESULTS ARE GIVEN" here. It is used to initialize the list of strings corresponding to the result block.

__init__(*current_batch*, *para*, *line*)

build_result(*line*)

Scan line to build batch result: mainly deals with specific patterns and store line.

Parameters
line (*str*) – last line to be taken into account

check_batch_number()

Check batch number value and replace it by current value or by the greater value if needed.

get_result()

Send result. Called if end flag has been found, add last line and concatenates result.

Returns
string build from list of strings junction

exception valjean.eponine.tripoli4.scan.**ScannerException**

An error that may be raised by the *Scanner* class.

class valjean.eponine.tripoli4.scan.**Scanner**(*fname*, *end_flag*='')

Class to scan the Tripoli-4 listing and keep the relevant parts of it like results per batch used for edition or times.

There are no class variables, but instance variables (initialized when the object is built or when the file is read). They are directly accessible from the object. Main results are accessible directly from the *Scanner* object.

Instance variables:

fname **(str)**
name of the file that will be scanned

batches **(dict)**
keep the number of batches and the packet_length (read from file *fname*)

normalend **(bool)**
presence of "NORMAL COMPLETION"

end_flags **(list (str))**
possible end flags to stop the saving of results

para **(bool)**
True in parallel mode

countwarnings **(int)**
count number of warnings (for statistics)

counterrors **(int)**
count number of errors (for statistics)

times **(collections.OrderedDict)**
save times (initialization, simulation, exploitation and elapsed if exists). Mandatory ones are `'initialization time'` and `'simulation time'` or `'exploitation time'`. `'elapsed time'` only appears in listings from parallel jobs.

last_generator_state **(str)**
keep the random generator state (not included in the result as given after *endflag*)

phemep_balance **(dict)**
keep the photon electron positron balance as `str` indexed by batch number

homog_mat **(dict)**
keep the homogenized material dump as str indexed by batch number

Available methods:

Scanner inherits from collections.abc.Mapping so many methods are implemented or available by default: keys, items, values, get, __contains__ (used via in). __getitem (used with []), __iter__ (when iterators are required), __len__ and __reversed__ are redefined.

This class points on the collections.OrderedDict that it contains. The keys are the available batch numbers in the Tripoli-4 output, obtained using scan_obj.keys().

__init__(*fname*, *end_flag*='')

Initialize the instance from the file *fname*, meaning reads the file and store the relevant parts of it, i.e. result block for each batch edition.

Results are stored in an internal collections.OrderedDict: {batch_number_1: 'result_1', batch_number_2: 'result_2', ...}

- batch_number_* is *int*
- result_* is *str*
- Order follows the listing order, so increasing batch_number

Members needed at initialization:

Parameters

- **fname** (*str*) – name of the input file
- **end_flag** (*str*) – end flag of the results block in Tripoli-4 listing

__getitem__(*batch_number*)

Get result corresponding to batch_number.

A warning is printed if the last batch_number doesn't correspond to the number of batchs required.

Parameters
batch_number (*int*) – batch number (>0), corresponding the keys of Scanner.

Raises
KeyError – if batch_number does not exist (for example if confusion between batch_number and batch_index using -1 or 0)

Use: Scanner[X]

__iter__()

Iteration over the collection of results, on the keys to match dict and collections.OrderedDict behaviour.

__len__()

Return length of the collection of results, equivalent to get the number of edited batchs.

__reversed__()

Reversed the collections.OrderedDict order (easier to get last element).

batch_index(*batch_number*)

Get the index of the batch_number in the results list.

batch_number(*batch_index*)

Get the batch number from the batch index.

fatal_error()

Return the fatal error message if found.

global_variables(*batch_number*)

Return a dictionary of the global quantities in the TRIPOILI-4 output:

- warnings
- errors
- number of tasks (to distinguish MONO and PARA for example)
- normal end
- required batches
- partial (if the job as been stopped)
- batch number, especially when 'edition after batch number' is not in the Tripoli-4 output
- t4_file: path to the scanned file
- times of the required batch (can be simulation time, initialisation time, elapsed time)

Parameters
batch_number (*int*) – batch number (used for times)

Return type
dict

print_statistics()

Print statistics of the listing scanned: normal end, number of warnings and errors.

check_times()

Check times.

If the results come from a parallel job, the *times* dictionary should contain the `'elapsed_time'` key. In all kind of outputs or `'simulation_time'` or `'exploitation_time'` should be found (in parallel case the only possibility is in reality `'simulation_time'`). Finally, while `'initialization_time'` value is an *int* as occuring only once, the others are lists with length equal to the number of batches in the edition for `'simulation_time'` and `'exploitation_time'` and equal to the number of batches + 1 for `'elapsed_time'`. If all these checks are successful True is returned, else False.

Return type
bool

use – Inject parsing results into free functions

This module extends the functionalities provided by *use* to the commonplace task of parsing Tripoli-4 output files. The decorators defined by this module, *using_parse_result* and *using_browser*, can be used to remove boilerplate code that parses and accesses the Tripoli-4 results. At the same time, they simplify the integration of user-defined post-processing (tests) into the *valjean* workflow.

We will need to set up a few things to show an example of how to use the decorators. We don't want to build and run Tripoli-4 for the purpose of this documentation, but we can mock a Tripoli-4 run by writing an existing Tripoli-4 output to a file. Here is the sample output:

```
>>> example = """BATCH 10
... initialization time (s): 7
...  batch number : 10
... ***********************************************************
...  RESULTS ARE GIVEN FOR SOURCE INTENSITY : 8.111452e+04
... ***********************************************************
...
...
...  Mean weight leakage = 7.929746e+00      sigma = 7.929746e+00    sigma% = 1.
↪000000e+02
...
...
...  Edition after batch number : 10
...
...
...
... ****************************************************************************
... RESPONSE FUNCTION : FLUX
... RESPONSE NAME :
... ENERGY DECOUPAGE NAME : DEC_SPECTRE
...
...
...  PARTICULE : NEUTRON
... ****************************************************************************
...
...          scoring mode : SCORE_SURF
...          scoring zone :          Frontier        volumes : 14,27
...
...
...          SPECTRUM RESULTS
...          number of first discarded batches : 0
...
...          group                   score           sigma_%         score/lethargy
... Units:   MeV                     neut.s^-1       %               neut.s^-1
...
... 2.000000e+01 - 1.000000e-11     1.307419e+00    8.719708e+01    1.046074e+01
...
...          ENERGY INTEGRATED RESULTS
...
...          number of first discarded batches : 0
...
... number of batches used: 10      1.307419e+00    8.719708e+01
...
...
...
```

(continues on next page)

(continued from previous page)

```
...  simulation time (s) : 20
...
...
...  Type and parameters of random generator at the end of simulation:
...          DRAND48_RANDOM 13531 45249 20024  COUNTER      2062560
...
...
...  simulation time (s): 20
...
...
... ========================================================================
...         NORMAL COMPLETION
... ========================================================================
... """
```

We construct a *RunTaskFactory* that simply echoes any text passed as an argument:

```
>>> from valjean.cosette.run import RunTaskFactory
>>> echo_factory = RunTaskFactory.from_executable('/bin/echo', name='echo',
...                                               default_args=['{text}'])
```

In the real world, the *RunTaskFactory* would actually generate tasks to run a Tripoli-4 executable.

Injecting the raw parse results

Now we can construct the decorator that parses the Tripoli-4 results and apply it to a Python function:

```
>>> from valjean.eponine.tripoli4 import use
>>> using_last_parse_result = use.using_last_parse_result(echo_factory)
>>> @using_last_parse_result(text=example)
... def source(results):
...     return results.res['batch_data']['source_intensity']
```

If we inspect *source*, we can see that it is a *Use* object:

```
>>> type(source)
<class 'valjean.cosette.use.Use'>
```

If you are not sure what *Use* does, this is a good moment to go and read its documentation. Don't worry, I'll wait.

(time passes)

valjean can inspect this object and convert it into a *Task*:

```
>>> source_task = source.get_task()
>>> type(source_task)
<class 'valjean.cosette.pythontask.PythonTask'>
>>> source_task.name
'....source'
```

The task has explicit dependencies, which means that it will be correctly integrated in the *valjean* dependency graph. In particular, this task depends on a task that actually does the parsing:

```
>>> source_task.depends_on
{Task('....parse_batch_index')}
>>> parse_task = next(iter(source_task.depends_on))
```

It depends on the *make_parser_task*, that scans the file and build the parser

```
>>> parse_task.depends_on
{Task('....make_parser')}
>>> make_parser_task = next(iter(parse_task.depends_on))
```

and *make_parser_task* in turn depends on the *RunTask* that was generated by *echo_factory*:

```
>>> make_parser_task.depends_on
{Task('....echo')}
>>> run_task = next(iter(make_parser_task.depends_on))
```

We manually execute the tasks in the correct order and we check that we recover the right result at the end:

```
>>> from valjean.cosette.env import Env
>>> env = Env()
>>> for task in [run_task, make_parser_task, parse_task, source_task]:
...     env_up, _ = task.do(env=env, config=config)
...     env.apply(env_up)
>>> print(env[source_task.name]['result'])
81114.52
```

Using a Browser

The raw parse results are useful in some situations, but most of the time you probably want to work with a higher-level representation of the calculation result, such as a *Browser*. This module provides a function to construct a decorator that automatically creates the *RunTask* from the *RunTaskFactory*, runs the task, parses the resulting output and wraps the parse results in an *Browser*:

```
>>> using_browser = use.using_browser(echo_factory, 10)
>>> @using_browser(text=example)
... def extract_simulation_time(browser):
...     return browser.globals['simulation_time']
```

We can again check that everything went as expected by manually unwrapping the sequence of tasks and running them:

```
>>> extract_task = extract_simulation_time.get_task()
>>> browse_task = next(iter(extract_task.depends_on))
>>> parse_task = next(iter(browse_task.depends_on))
>>> make_parser_task = next(iter(parse_task.depends_on))
>>> run_task = next(iter(make_parser_task.depends_on))
>>> env = Env()
>>> for task in [run_task, make_parser_task, parse_task,
...              browse_task, extract_task]:
...     env_up, _ = task.do(env=env, config=config)
...     env.apply(env_up)
>>> print(env[extract_task.name]['result'])
20
```

Module API

valjean.eponine.tripoli4.use.**partial**(*func*, **args*, ***kwargs*)

An improved version of `functools.partial` that calls `functools.update_wrapper` on the partially applied function in order to update its metadata (name, etc.).

valjean.eponine.tripoli4.use.**make_parser**(*filename*)

Create a Parser object and scan a Tripoli-4 output file.

Parameters

- **filename** (*str*) – the name of the file to parse.
- **batch** (*int*) – the number of the batch to parse; see *__init__*.

Raises
ValueError – if parsing fails.

Returns
the parser

Return type
Parser

valjean.eponine.tripoli4.use.**parse_batch_number**(*parser*, ***, *batch_number*)

Parse a batch result from Tripoli-4.

Parameters
batch_number (*int*) – batch number

Return type
ParseResult

valjean.eponine.tripoli4.use.**parse_batch_index**(*parser*, ***, *batch_index=-1*)

Parse a batch result from Tripoli-4.

Parameters
batch_index (*int*) – index of the batch in the list of batches

Return type
ParseResult

valjean.eponine.tripoli4.use.**using_parser**(*factory*)

Construct a decorator that injects Tripoli-4 parser into a Python function.

Parameters
factory (*RunTaskFactory*) – a factory producing Tripoli-4 runs.

Returns
a decorator (see the module docstring for more information).

valjean.eponine.tripoli4.use.**using_parse_result**(*factory*, *batch_number*)

Construct a decorator that injects the raw Tripoli-4 parse results into a Python function.

Parameters

- **factory** (*RunTaskFactory*) – a factory producing Tripoli-4 runs.
- **batch_number** (*int*) – the number of the batch to parse; see *__init__*.

Returns
a decorator (see the module docstring for more information).

valjean.eponine.tripoli4.use.**using_last_parse_result**(*factory*)

Construct a decorator that injects the last raw Tripoli-4 parse results into a Python function.

Parameters
factory (*RunTaskFactory*) – a factory producing Tripoli-4 runs.

Returns
a decorator (see the module docstring for more information).

valjean.eponine.tripoli4.use.**to_browser**(*result*)

Create a *Browser* from the parsing result.

Parameters
result (ParseResult) – result from T4 parser

Returns
the browser

Return type
Browser

valjean.eponine.tripoli4.use.**using_browser**(*factory*, *batch_number*)

Construct a decorator that injects an *Browser* to the Tripoli-4 parse results into a Python function.

Parameters

- **factory** (*RunTaskFactory*) – a factory producing Tripoli-4 runs.
- **batch_number** (*int*) – the number of the required batch (will be parsed then transformed in *Browser*)

Returns
a decorator (see the module docstring for more information).

valjean.eponine.tripoli4.use.**using_last_browser**(*factory*)

Construct a decorator that injects a *Browser* to the Tripoli-4 last parse results into a Python function.

Parameters
factory (*RunTaskFactory*) – a factory producing Tripoli-4 runs.

Returns
a decorator (see the module docstring for more information).

depletion – Read Tripoli-4 depletion results

Module to deal with ROOT outputs from Tripoli-4 in depletion mode.

This module is an interface with Tripoli-4 depletion results, stored as ROOT files, see the Tripoli-4 user guide and the ROOT website for more details.

The available results match the usual Tripoli-4 results from depletion:

- k_{eff} as kcoll, ktrack, kstep;
- β_:sub:*eff* from prompt and Nauchi, called beff_prompt and beff_nauchi;
- renormalisation factor, called renorm
- total power

- power
- local burnup
- flux: fast neutron flux (`fast_flux`), thermal neutron flux (`therm_flux`) and total flux (`flux`)
- mass
- concentration
- activity
- reaction rates: for fast neutrons (`fast_reaction_rate`), for thermal neutrons (`thermal_reaction_rate`) and for one group (`reaction_rate`). The thermal group only gives non-zero results for fission rate.

Some of these results can be accessed from valjean using the *DepletionReader* class from this module. In some cases, it is necessary to provide keyword for the composition name (`componame`), the isotope name (`isotope`) or the reaction name (`reaction`).

Results can be obtained at a given depletion step (so a scalar value) or for all available steps, giving in both cases a *Dataset*. The access method of the array ends with `_time` if the required quantity is given in function of *time* and `_burnup` if it is in function of *burnup*.

A typical example of use of this module is the following:

```
from valjean.eponine.tripoli4.depletion import DepletionReader
depr = DepletionReader.from_evolution_steps(
    'evolution_1.root', 'evolution_2.root',
    root_build='/path/where/to/build/root/libraries')
# get kstep at step 1
kstep = depr.kstep(1)
# get kstep as a function of burnup
akstep = depr.kstep_burnup()
# get total power as a function of time
atotpow = depr.total_power_time()
# get U238 concentration in composition COMPO1 at step 5
conc_u238 = depr.concentration(step=5, componame='COMPO1', isotope='U238')
# get U238 fission reaction rate in composition COMP1 as a function of time
reac_u238 = depr.reaction_rate_time(componame='COMP1', isotope='U238',
                                    reaction='REAMT18')
```

Some helper methods provide the list of compositions, the list of isotopes in a given composition or the reaction rates associated to a given isotope in a given composition.

`valjean.eponine.tripoli4.depletion.`**`title_to_snake_case`**(*word*)

Convert *word* from title case to snake case.

```
>>> title_to_snake_case('ThisIsATitle')
'this_is_a_title'
```

`valjean.eponine.tripoli4.depletion.`**`generic_docstrings`**(*res_type*, **list_params*)

Define generic docstrings for the access functions of depleted results.

`valjean.eponine.tripoli4.depletion.`**`add_accessors`**(*dict_res*)

Automatic construction of result accessors.

`class` `valjean.eponine.tripoli4.depletion.`**`DepletionReader`**(*mbr*)

Class to use depletion results from Tripoli-4

__init__(*mbr*)

Initialisation of DepletionReader.

Parameters
mbr – MeanBurnupResult

static init_postscripts(*root_build=''*)

Initialize postscripts from ROOT macros.

ROOT macros are in the *resources/depletion* folder. They are compiled in the *__t4depletion__* folder to used afterwards.

classmethod from_evolution_steps(**fnames, root_build=''*)

Initialisation from *evolution.root* files (one per simulation).

Parameters
fnames (*str*) – ROOT files

classmethod from_mbr(*fname*, *mbr_name*, *root_build=''*)

Initialisation from ROOT file containing a MeanBurnupResults class.

Parameters

- **fname** (*str*) – ROOT file name
- **mbr_name** (*str*) – name of the MeanBurnupResults object in the file

save_mbr(*name*)

Save the MeanBurnupResults in a ROOT file.

Parameters
name (*str*) – name of the output ROOT file name

nb_simu()

Return the number of independent simulations.

Return type
int

nb_compositions()

Return the number of compositions.

:rtype; int

nb_steps()

Return the number of steps.

Return type
int

burnup(*step*)

Return burnup from MeanBurnupResults object.

Parameters
step (*int*) – calculation step to use to get the value

Returns
burnup value and error

Return type
Dataset

burnup_array()
Return burnup array from MeanBurnupResults object.

Returns
burnup value and error by step

Return type
Dataset

time(*step*)
Return time from MeanBurnupResults object.

Parameters
step (*int*) – calculation step to use to get the value

Returns
time value and error

Return type
Dataset

time_array()
Return time array from MeanBurnupResults object.

Returns
time value and error by step

Return type
Dataset

composition_names()
Return the list of composition names.

Return type
list(str)

isotope_names(*step*, *componame*)
Return the list of isotopes in the given composition at the given step.

Parameters

- **step** (*int*) – calculation step to use to get the value
- **componame** (*str*) – composition name in which list of isotopes is required

Returns
list of available isotopes

Return type
list(str)

reaction_names(*step*, *componame*, *isotope*)
Get reaction names in composition for the given isotope.

Parameters

- **step** (*int*) – calculation step to use to get the value
- **componame** (*str*) – composition name where reaction is required
- **isotope** (*str*) – isotope name for which reaction is required

Returns
list of available reactions

Return type
list(str)

isotope_reaction_names(*step*, *componame*)

Get dictionary of reactions per isotope for the given step and composition.

Parameters

- **step** (*int*) – calculation step to use to get the value
- **componame** (*str*) – composition name where reactions per isotope are required

Returns
reactions by isotopes

Return type
dict(str, list(str))

dump_global_results(*step*)

Print results for a given step.

This includes compositions.

activity(*step*, ***kwargs*)

Return the value of Activity from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for Activity
- **isotope** (*str*) – isotope which Activity is required

Returns
Activity and error for the given step

Return type
Dataset

activity_burnup(***kwargs*)

Return the Activity from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for Activity
- **isotope** (*str*) – isotope which Activity is required

Returns
Activity and error as a function of burnup steps

Return type
Dataset

activity_time(***kwargs*)

Return the Activity from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for Activity
- **isotope** (*str*) – isotope which Activity is required

Returns

Activity and error as a function of time steps

Return type

Dataset

beff_nauchi(*step*, ***kwargs*)

Return the value of BeffNauchi from the MeanBurnupResults object.

Parameters

step (*int*) – the index of the calculation step

Returns

BeffNauchi and error for the given step

Return type

Dataset

beff_nauchi_burnup(***kwargs*)

Return the BeffNauchi from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns

BeffNauchi and error as a function of burnup steps

Return type

Dataset

beff_nauchi_time(***kwargs*)

Return the BeffNauchi from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns

BeffNauchi and error as a function of time steps

Return type

Dataset

beff_prompt(*step*, ***kwargs*)

Return the value of BeffPrompt from the MeanBurnupResults object.

Parameters
step (*int*) – the index of the calculation step

Returns
BeffPrompt and error for the given step

Return type
Dataset

beff_prompt_burnup(***kwargs*)

Return the BeffPrompt from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
BeffPrompt and error as a function of burnup steps

Return type
Dataset

beff_prompt_time(***kwargs*)

Return the BeffPrompt from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
BeffPrompt and error as a function of time steps

Return type
Dataset

concentration(*step*, ***kwargs*)

Return the value of Concentration from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for Concentration
- **isotope** (*str*) – isotope which Concentration is required

Returns
Concentration and error for the given step

Return type
Dataset

concentration_burnup(***kwargs*)

Return the Concentration from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for Concentration
- **isotope** (*str*) – isotope which Concentration is required

Returns

Concentration and error as a function of burnup steps

Return type

Dataset

concentration_time(***kwargs*)

Return the Concentration from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for Concentration
- **isotope** (*str*) – isotope which Concentration is required

Returns

Concentration and error as a function of time steps

Return type

Dataset

fast_flux(*step*, ***kwargs*)

Return the value of FastFlux from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for FastFlux

Returns

FastFlux and error for the given step

Return type

Dataset

fast_flux_burnup(***kwargs*)

Return the FastFlux from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters
componame (*str*) – composition name for FastFlux

Returns
FastFlux and error as a function of burnup steps

Return type
Dataset

fast_flux_time(***kwargs*)

Return the FastFlux from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters
componame (*str*) – composition name for FastFlux

Returns
FastFlux and error as a function of time steps

Return type
Dataset

fast_reaction_rate(*step*, ***kwargs*)

Return the value of FastReactionRate from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for FastReactionRate
- **isotope** (*str*) – isotope which FastReactionRate is required
- **reaction** (*str*) – reaction name

Returns
FastReactionRate and error for the given step

Return type
Dataset

fast_reaction_rate_burnup(***kwargs*)

Return the FastReactionRate from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for FastReactionRate
- **isotope** (*str*) – isotope which FastReactionRate is required
- **reaction** (*str*) – reaction name

Returns
FastReactionRate and error as a function of burnup steps

Return type
Dataset

fast_reaction_rate_time(***kwargs*)

Return the FastReactionRate from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for FastReactionRate
- **isotope** (*str*) – isotope which FastReactionRate is required
- **reaction** (*str*) – reaction name

Returns
FastReactionRate and error as a function of time steps

Return type
Dataset

kcoll(*step*, ***kwargs*)

Return the value of Kcoll from the MeanBurnupResults object.

Parameters
step (*int*) – the index of the calculation step

Returns
Kcoll and error for the given step

Return type
Dataset

kcoll_burnup(***kwargs*)

Return the Kcoll from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
Kcoll and error as a function of burnup steps

Return type
Dataset

kcoll_time(***kwargs*)

Return the Kcoll from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
Kcoll and error as a function of time steps

Return type
Dataset

kstep(*step*, ***kwargs*)

Return the value of Kstep from the MeanBurnupResults object.

Parameters
step (*int*) – the index of the calculation step

Returns
Kstep and error for the given step

Return type
Dataset

kstep_burnup(***kwargs*)

Return the Kstep from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
Kstep and error as a function of burnup steps

Return type
Dataset

kstep_time(***kwargs*)

Return the Kstep from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
Kstep and error as a function of time steps

Return type
Dataset

ktrack(*step*, ***kwargs*)

Return the value of Ktrack from the MeanBurnupResults object.

Parameters
step (*int*) – the index of the calculation step

Returns
Ktrack and error for the given step

Return type
Dataset

ktrack_burnup(***kwargs*)

Return the Ktrack from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
Ktrack and error as a function of burnup steps

Return type
Dataset

ktrack_time(***kwargs*)

Return the Ktrack from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
Ktrack and error as a function of time steps

Return type
Dataset

local_burnup(*step*, ***kwargs*)

Return the value of LocalBurnup from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for LocalBurnup

Returns
LocalBurnup and error for the given step

Return type
Dataset

local_burnup_burnup(***kwargs*)

Return the LocalBurnup from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters
componame (*str*) – composition name for LocalBurnup

Returns
LocalBurnup and error as a function of burnup steps

Return type
Dataset

local_burnup_time(***kwargs*)
Return the LocalBurnup from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters
componame (*str*) – composition name for LocalBurnup

Returns
LocalBurnup and error as a function of time steps

Return type
Dataset

mass(*step*, ***kwargs*)
Return the value of Mass from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for Mass
- **isotope** (*str*) – isotope which Mass is required

Returns
Mass and error for the given step

Return type
Dataset

mass_burnup(***kwargs*)
Return the Mass from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for Mass
- **isotope** (*str*) – isotope which Mass is required

Returns
Mass and error as a function of burnup steps

Return type
Dataset

mass_time(***kwargs*)

Return the Mass from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for Mass
- **isotope** (*str*) – isotope which Mass is required

Returns

Mass and error as a function of time steps

Return type

Dataset

power(*step*, ***kwargs*)

Return the value of Power from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for Power

Returns

Power and error for the given step

Return type

Dataset

power_burnup(***kwargs*)

Return the Power from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

componame (*str*) – composition name for Power

Returns

Power and error as a function of burnup steps

Return type

Dataset

power_time(***kwargs*)

Return the Power from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters
componame (*str*) – composition name for Power

Returns
Power and error as a function of time steps

Return type
Dataset

reaction_rate(*step*, ***kwargs*)

Return the value of ReactionRate from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for ReactionRate
- **isotope** (*str*) – isotope which ReactionRate is required
- **reaction** (*str*) – reaction name

Returns
ReactionRate and error for the given step

Return type
Dataset

reaction_rate_burnup(***kwargs*)

Return the ReactionRate from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for ReactionRate
- **isotope** (*str*) – isotope which ReactionRate is required
- **reaction** (*str*) – reaction name

Returns
ReactionRate and error as a function of burnup steps

Return type
Dataset

reaction_rate_time(***kwargs*)

Return the ReactionRate from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for ReactionRate

- **isotope** (*str*) – isotope which ReactionRate is required
- **reaction** (*str*) – reaction name

Returns
ReactionRate and error as a function of time steps

Return type
Dataset

renorm(*step*, ***kwargs*)
Return the value of Renorm from the MeanBurnupResults object.

Parameters
step (*int*) – the index of the calculation step

Returns
Renorm and error for the given step

Return type
Dataset

renorm_burnup(***kwargs*)
Return the Renorm from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
Renorm and error as a function of burnup steps

Return type
Dataset

renorm_time(***kwargs*)
Return the Renorm from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
Renorm and error as a function of time steps

Return type
Dataset

therm_flux(*step*, ***kwargs*)
Return the value of ThermFlux from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for ThermFlux

Returns
ThermFlux and error for the given step

Return type
Dataset

therm_flux_burnup(***kwargs*)

Return the ThermFlux from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters
componame (*str*) – composition name for ThermFlux

Returns
ThermFlux and error as a function of burnup steps

Return type
Dataset

therm_flux_time(***kwargs*)

Return the ThermFlux from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters
componame (*str*) – composition name for ThermFlux

Returns
ThermFlux and error as a function of time steps

Return type
Dataset

thermal_reaction_rate(*step*, ***kwargs*)

Return the value of ThermalReactionRate from the MeanBurnupResults object.

Parameters

- **step** (*int*) – the index of the calculation step
- **componame** (*str*) – composition name for ThermalReactionRate
- **isotope** (*str*) – isotope which ThermalReactionRate is required
- **reaction** (*str*) – reaction name

Returns
ThermalReactionRate and error for the given step

Return type
Dataset

thermal_reaction_rate_burnup(***kwargs*)

Return the ThermalReactionRate from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for ThermalReactionRate
- **isotope** (*str*) – isotope which ThermalReactionRate is required
- **reaction** (*str*) – reaction name

Returns
ThermalReactionRate and error as a function of burnup steps

Return type
Dataset

thermal_reaction_rate_time(***kwargs*)

Return the ThermalReactionRate from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Parameters

- **componame** (*str*) – composition name for ThermalReactionRate
- **isotope** (*str*) – isotope which ThermalReactionRate is required
- **reaction** (*str*) – reaction name

Returns
ThermalReactionRate and error as a function of time steps

Return type
Dataset

total_power(*step*, ***kwargs*)

Return the value of TotalPower from the MeanBurnupResults object.

Parameters
step (*int*) – the index of the calculation step

Returns
TotalPower and error for the given step

Return type
Dataset

total_power_burnup(***kwargs*)

Return the TotalPower from the MeanBurnupResults object as a function of burnup.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
TotalPower and error as a function of burnup steps

Return type
Dataset

total_power_time(***kwargs*)
Return the TotalPower from the MeanBurnupResults object as a function of time.

Note: The unit of the x-axis is not currently preserved as units are not taken into account in *Dataset*.

Returns
TotalPower and error as a function of time steps

Return type
Dataset

exception valjean.eponine.tripoli4.depletion.**DepletionReaderException**
An error that may be raised by the *DepletionReader* class.

More details on parsing

Most of the Tripoli-4 functionalities are now parsed by pyparsing. If a new score or a new response is implemented in Tripoli-4, it would be easier for its output to match the existing responses, or else parsing will have to be updated.

When parsing fails it indicates the last line and column where parsing succeeded (relative to the beginning of the parsed string, not the full output) and the problematic line. This should help debugging.

Tests should be performed in the code to take the latter case into account (example: test presence of `'simulation time'` keyword, see *check_times*).

grammar – Pyparsing grammar

This module provides *pyparsing* grammar for Tripoli-4 output listings.

Documentation on the `pyparsing` package can be found at pyparsing.

Transformation from `pyparsing.ParseResults` to more standard python objects, including `numpy` arrays, is done with *transform*, calling *common*.

Generalitites

- This parser only parses the result part of the listing (selection done in *scan*).
- It takes into account all responses in `qualtrip` database up to Tripoli-4, version 10.2.
- If a response is not taken into account parsing will fail:
  - with a big, ugly message ending by location of the end of successful parsing in the result string (possible to print it) → normally where starts your new response
  - it seems to end normally, but did not in reality. One of the best checks in that case is to test if the `endflag` in *scan* was read in the parser, usually not. Then the new response probably have to be added.
- A general parser is proposed for use in the file, but other parsers can be built from the partial parsers written here
- Numbers are automatically converted to `numpy` numbers (possibility to choose the dtype used for numbers)
- Keywords and most of the variables used to build parsers are private

Organisation

This module is divided in 3 parts:

keywords:
: list of all keywords used to parse the listings, this part is important as these keywords trigger the parsing

parsers:
: parsers for each part of the listing (introduction, mesh, spectra, general responses, k_{eff}, etc.)

general parser:
: parser to parse the full listing, taking into accout all current response (present in V&V)

Keywords are in most of the cases used as flags and suppressed when data are stored.

A first structure is designed when building the parsers results as lists and dictionaries in the `pyparsing.ParseResults`. Then *parse actions* are used to standard python or `numpy` objects. These *parse actions*, called with `pyparsing.ParserElement.setParseAction`, can be found in *transform*.

Main parsers blocks

The main parsers blocks are defined at the end of the module, named `mygram` and `response`. The default parser is `mygram`.

Typically, each result block in the listing should start by the *intro* block, parsed by `intro`, and end with at least one *runtime* block, parsed by `runtime`. This parts follows the *scan*: `str` starting by `'RESULTS ARE GIVEN'` and ending with `'simulation time'`, `'exploitation time'` or `'elapsed time'`.

Between these blocks can be found the data blocks. The major ones are:

- one or more responses, driven by the keyword `'RESPONSE FUNCTION'`,

- the editions of IFP adjoint criticality,
- the "default" k_{eff} block, in most of the cases at the end of the listing,
- the *contributing particles block*, mainly in first pass listings,
- the perturbation block,
- an optional additional *runtime* block.

Main data blocks are described below (results taken into account, main features).

Response block, parser `response`

The core of the listings is the list of responses, including all the required scores. This big block is constructed as a `list` of `dict`, each one representing a response (key `'list_responses'` in the final result).

Response are constructed as:

- response introduction containing its definition, parsed by `respintro`:
  - a description of the response parsed by `respdesc` including:
    - `'RESPONSE FUNCTION'` keyword as mandatory (afaik)
    - `'RESPONSE NAME'`, `'SCORE NAME'` and `'ENERGY DECOUPAGE NAME'` that are present in most of the cases
  - more characteristics of the response, parsed by `respcarac`, like:
    - considered particle (`'PARTICULE'` in the listing)
    - nucleus on which the reaction happens (if `'RESPONSE FUNCTION'` is a `'REACTION'`)
    - temperature
    - compostion of the volumes considered
    - concentration
    - reaction considered (usually given as codes)
    - others like DPA type, required arguments, mode, filters, etc.
- responses themselves, using parser `responseblock`, are various:
  - responses including *score* description, all included in the `scoreblock` parser. More than one can be present, they are grouped in the `listscoreblock` parser. `scoreblock` parser contains:
    - score description (parser `scoredesc`) contains the score mode (`'TRACK'`, `'SURF'` or `'COLL'`) and the score zone (currently taken into account: mesh, results cumulated on all geometry or on all sources, Volume, Volume Sum, Frontier, Frontier Sum, Point, Cells and Maille)
    - results block, where at least one of these results can be found, are by parsed by the following parsers:
      - `spectrumblock`: spectrum
      - `meshblock`: mesh
      - `vovspectrumblock`: spectrum with variance of variance

        - `entropy`: entropy results (Boltzmann and Shannon entropies)
        - `medfile`: location of optional MED file
        - `genericscoreblock`: default result integrated over energy
        - `uncertblock`: uncertainty results
        - `uncertintegblock`: uncertainties on integrated results over energy
        - `gbblock`: Green bands results
    - k_{eff} presented as a generic response, possibly transformed in `numpy.matrix` (parser `keffblock`)
    - k_{ij} results: matrix, eigenvalues, eigenvectors (parser `kijres`)
    - k_{ij} sources (parser `kijsources`)
    - Adjoint related results (parser `adjointres`): scores ordered by precursors and families, by perturbation index, by cycle length or sensitivities (this last case is represented in a 3 dimensions `numpy.ndarray`, incident energy, energy ("leaving neutron"), direction cosine (μ)). For the moment this is only for IFP method , in close future also for Wielandt method
    - default result integrated over energy where no scoring mode and zone are precised (parser `defintegratedres`)
    - perturbation results (parser `perturbation`)

Other parsers

Various other blocks can appear in the Tripoli-4 listing, located at the same level as the response block. These parsers and the associated dictionary key (same level as `'list_responses'`) are:

- `ifpadjointcriticality`: edition of IFP adjoint criticality, key `'ifp_adjoint_crit_edition'`;
- `autokeffblock`: "default" k_{eff} block, containing for example the best estimation of k_{eff} using variable number of discarded batches, key `'keff_auto'`;
- `contribpartblock`: *contributing particles block*, key `'contributing_particles'`
- `perturbation`: perturbation results, containing a description of the calculation of the perturbation followed by the perturbation result presented like a usual response (spectrum, mesh, etc. depending on required score), key `'perturbation'`
- `runtime`: simulation, exploitation or elapsed time.

Todo: Adjoint results: for the moment only IFP is really parsed. Grammar has already more or less adapted to welcome Wielandt method that will have the same kind of outputs (renaming as adjoint_res for example). No key is set for the moment to specify the method, it can be obtained from the response function itself. Adjoint criticality editions are only done for IFP, this may change when the same will be available for Wielandt. Some renaming can also be needed.

`transform` – Transform pyparsing result

This module converts *pyparsing* objects in *python* or `numpy` objects.

It is called in the module *`grammar`* via `pyparsing.ParserElement.setParseAction` functions. It calls the general module *`common`*.

Note: This module is not standalone, needs a *pyparsing* result in input.

`valjean.eponine.tripoli4.transform.`**`compose2`**(*f*, *g*)

Functions composition (2 functions), like fog in mathematics.

Params func f
last function to compose, takes result from g as arguments

Params func g
first function to apply

Returns
composition of the 2 functions (func)

`valjean.eponine.tripoli4.transform.`**`compose`**(**fs*)

General composition in case more than 2 functions are needed. For example for fogoh(x) = f(g(h(x))). Takes as many functions as needed as argument.

`valjean.eponine.tripoli4.transform.`**`convert_spectrum`**(*toks*, *spectrum_type*)

Convert spectrum to `numpy` object using *`common`*.

Parameters

- **`toks`** (pyparsing.ParseResults) – spectrum result
- **`spectrum_type`** (*`str`*) – type of spectrum that defines the names of the columns

- Default column names: `['score', 'sigma', 'score/lethargy']`
- Other column names taken into account:
  - vov: adding `'vov'` to previous columns names
  - uncert: names are `['sigma2(means)', 'mean(sigma_n2)', 'sigma(sigma_n2)', 'fisher test']`
  - nu and za: no `'score/lethargy'` is available

Returns
dictionary containing spectrum as 7-dimensions numpy structured array for result, `numpy.ndarray` for binnings, discarded batchs, used batchs, integrated result (depending on availability)

See also:

`common.convert_spectrum` and more generally *`common`*

`valjean.eponine.tripoli4.transform.`**`convert_mesh`**(*toks*)

Convert mesh to `numpy` object using *`common`*.

Parameters
`toks` (pyparsing.ParseResults) – mesh result

Returns
dictionary containing meshes (integrated over energy or not) as 7-dimensions numpy structured array, binnings, etc. depending on availability

See also:

common.convert_mesh and more generally *valjean.eponine.tripoli4.common*

valjean.eponine.tripoli4.transform.**convert_green_bands**(*toks*)

Convert Green bands to numpy object using *common*.

Parameters
toks (pyparsing.ParseResults) – Green bands result

Returns
dictionary containing Green bands as 6-dimensions numpy structured array, numpy.ndarray for binnings, etc. depending on availability

See also:

common.convert_green_bands and more generally *common*

valjean.eponine.tripoli4.transform.**convert_correspondence_table**(*toks*)

Convert correspondence table to dict (volume id, volume name).

Returns
tuple(tuple)

valjean.eponine.tripoli4.transform.**convert_scoring_zone_id**(*toks*)

Convert scoring zone id (volume numbers, cells, points, etc) into native python objects (str, tuple, numpy object).

valjean.eponine.tripoli4.transform.**convert_list_to_tuple**(*toks*)

Convert list from pyparsing to tuple.

This function is only used to transform the parseResult in standard python list and send it to convert_list_to_tuple in *common*.

valjean.eponine.tripoli4.transform.**convert_batch_numbers**(*score*)

Convert batch numbers (used and discarded) in proper results.

Parameters
toks (pyparsing.ParseResults) – score result interpreted as dictionary

Returns
dictionary with maximum 2 keys ('used_batches_res' and 'discarded_batches_res')

Return type
dict

valjean.eponine.tripoli4.transform.**convert_score**(*toks*)

Convert score to numpy and python objects. Calls various conversion functions depending on input key (mesh, spectrum, Green bands, default python *dict*, etc.).

Parameters
toks (pyparsing.ParseResults) – score result interpreted as dictionary

Returns
dictionary using the previous keys and numpy objects as values.

valjean.eponine.tripoli4.transform.**convert_generic_adjoint**(*toks*)

Convert adjoint output in numpy object using *common*.

This method does not take into account sensitivities calculated via the IFP method.

Parameters
toks (pyparsing.ParseResults) – Adjoint result (got thanks to IFP or Wielandt method)

Returns
list(dict) compatible with *Browser* and *Index*.

See also:

common.convert_generic_adjoint and more generally *common*

valjean.eponine.tripoli4.transform.**convert_generic_kinetic**(*toks*)

Convert generic scores for kinetic simulations into a numpy object.

Parameters
toks (pyparsing.ParseResults) – parsed tokens

Returns
list(dict) compatible with *Browser* and *Index*.

See also:

convert_generic_kinetic and more generally *common*

valjean.eponine.tripoli4.transform.**convert_keff**(*toks*)

Convert k_{eff} response in python dictionary including numpy.matrix and using *common*.

Parameters
toks (pyparsing.ParseResults) – k_{eff} result interpreted as dictionary

Returns
dictionary using numpy objects including numpy.matrix

Note: For the moment, *common.convert_keff_with_matrix* is called. It is possible to call *common.convert_keff* instead.

See also:

common.convert_keff and more generally *common*

valjean.eponine.tripoli4.transform.**convert_kij_sources**(*toks*)

Convert k_{ij} sources to python dictionary containing numpy objects and using *common*.

Parameters
toks (pyparsing.ParseResults) – k_{ij} source result (interpreted as dictionary)

Returns
dictionary where k_{ij} values are inside numpy.ndarray

See also:

common.convert_kij_sources and more generally *common*

valjean.eponine.tripoli4.transform.**convert_kij_result**(*toks*)

Convert k_{ij} result to dictionary of numpy objects using *common*.

Parameters
toks (pyparsing.ParseResults) – k_{ij} results (interpreted as dictionary)

Returns
dictionary of `numpy.ndarray` and `numpy.matrix`

See also:

common.convert_kij_result and more generally *common*

`valjean.eponine.tripoli4.transform.`**`convert_kij_keff`**(*toks*)

Convert k_{eff} estimated from k_{ij} to dictionary of `numpy` objects using *common*.

Parameters
toks (pyparsing.ParseResults) – k_{eff} block result interpreted as a list t. Only last element is used here (others go to *convert_keff*).

Returns
dictionary of `numpy.ndarray` and `numpy.matrix`.

Note: It is possible to add a check on estimator if issue.

See also:

common.convert_kij_keff and more generally *common*

`valjean.eponine.tripoli4.transform.`**`convert_keff_auto`**(*toks*)

Convert auto keff estimation (old or default output, not printed as a usual response) in standard python objects.

Add the `'response_type'` key to the dictionary with `'keff_auto_res'` as associated value (to match browser and data_convertor requirements).

`valjean.eponine.tripoli4.transform.`**`convert_sensitivities`**(*toks*)

Convert sensitivity results to dictionary of `numpy` objects using *common*.

Parameters
toks (pyparsing.ParseResults) – *pyparsing* element

Returns
python list corresponding to input *pyparsing* list

`valjean.eponine.tripoli4.transform.`**`convert_shr`**(*toks*)

Convert results on spherical harmonics to dictionary of `numpy.ndarray` using *common*.

Parameters
toks (pyparsing.ParseResults) – *pyparsing* element

Returns
python list corresponding to input *pyparsing* list

`valjean.eponine.tripoli4.transform.`**`convert_ifp_adj_crit_ed`**(*toks*)

Convert IFP adjoint criticality edition in kinematic array.

`valjean.eponine.tripoli4.transform.`**`to_final_dict`**(*toks*)

Convert to dictionary result of *pyparsing*.

Parameters
toks (pyparsing.ParseResults) – *pyparsing* element

Returns
python dictionary corresponding to input *pyparsing* dictionary

valjean.eponine.tripoli4.transform.**lod_to_dot**(*toks*)

List of dictionaries to dictionary of tuples. This function is dedicated to cases where all the dictionaries (or close) in the list have the same keys.

Parameters
toks (*list(dict)*) – list of dictionaries

Returns
dict(tuple)

```
>>> from pprint import pprint
>>> from pyparsing import OneOrMore, Group, Word, nums
>>> menu = OneOrMore(Group(Word(nums)('egg') + ','
...                        + Word(nums)('bacon') + ','
...                        + Word(nums)('spam')))
>>> lod = menu.parseString('1,2,0 2,0,1 0,3,1')
>>> dot = lod_to_dot(lod)
>>> pprint(dot)
{'bacon': ('2', '0', '3'), 'egg': ('1', '2', '0'), 'spam': ('0', '1', '1')}
```

```
>>> lod = [{'egg': 1, 'bacon': 2, 'spam': 0},
...        {'egg': 2, 'bacon': 0, 'spam': 1},
...        {'egg': 0, 'bacon': 3, 'spam': 1}]
>>> dot = lod_to_dot(lod)
>>> pprint(dot)
{'bacon': (2, 0, 3), 'egg': (1, 2, 0), 'spam': (0, 1, 1)}
```

```
>>> lod = [{'egg': 1, 'bacon': 2, 'spam': 0}]
>>> dot = lod_to_dot(lod)
>>> pprint(dot)
{'bacon': (2,), 'egg': (1,), 'spam': (0,)}
```

We always get a tuple as value of the keys, even in case of a single element.

valjean.eponine.tripoli4.transform.**finalize_response_dict**(*s*, *loc*, *toks*)

Finalize the dictionary of response.

Extract the real response results from pyparsing structure and set it under the *results* key. The previous unique key of the dictionary under the *results* key is stored under the *result_type* key.

The input is the whole response, i.e. the results and the response description. This second part is not "modified" here, only copied in the new dictionary, at the exception of the metadata stored under the *compos_details* key. In that case the dictionary stored under the *compos_details* key is moved to the upper level, i.e. in the response dictionary. As a consequence, the *compos_details* key disappears.

Parameters
toks (pyparsing.ParseResults) – *pyparsing* element

Returns
python dict corresponding to input *pyparsing* response result

valjean.eponine.tripoli4.transform.**extract_all_metadata**(*list_of_dicts*)

Extract metadata from nested lists of dictionaries. The metadata to be extracted are here in the second level of list of dictionaries as in the example:

```
>>> from pprint import pprint
>>> lod = [{'bla': 'X', 'results': [
...     {'data1_res': 'D1', 'data2_res': 2, 'md1': 'MD1', 'md2': 'MD2'},
...     {'data1_res': 'D3', 'md1': 'MD1', 'md3': 3}]}]
>>> pprint(extract_all_metadata(lod))
[{'bla': 'X', 'md1': 'MD1', 'md2': 'MD2', 'results': {'data1': 'D1', 'data2': 2}},
 ↪ {'bla': 'X', 'md1': 'MD1', 'md3': 3, 'results': {'data1': 'D3'}}]
```

Parameters
list_of_dicts (*list(dict)*) – list of dictionaries

Returns
list of dictionaries with no list of dict under 'results' key

Return type
list(dict)

valjean.eponine.tripoli4.transform.**extract_metadata**(*ldict*)

Extract metadata from a list of dictionaries to put it in the surrounding dictionary.

Parameters
ldict (*dict*) – dictionary corresponding to a response

Return type
list(dict)

valjean.eponine.tripoli4.transform.**index_elements**(*key*)

Add an item *key* in dictionary corresponding to index in the list containing the dictionary.

Parameters
key (*str*) – name of the index

Returns
function that really insert the index in the dict contained in a list

```
>>> from pprint import pprint
>>> lod = [{'a': 1, 'b': 5}, {'c': -3}, {'d': 4, 'e': 6, 'f': 8}]
>>> func1 = index_elements('index')
>>> nlod = func1('', 0, lod)
>>> pprint(nlod)
[{'a': 1, 'b': 5, 'index': 0}, {'c': -3, 'index': 1}, {'d': 4, 'e': 6, 'f': 8,
 ↪'index': 2}]
```

valjean.eponine.tripoli4.transform.**propagate_all_metadata**(*list_of_dicts*)

Concatenate metadata from a list of dictionaries.

It applies *propagate_top_metadata* for all dictionaries in the list of dictionaries, one by one.

The metadata to be propagated are here in the first level of list of dictionaries as in the example:

```
>>> from pprint import pprint
>>> lod = [{'ext_metadata': {'emd1': 'EMD1', 'emd2': 'EMD2'},
...         'list_responses': [
...     {'bla': 'X', 'md1': 'MD1', 'md2': 'MD2',
...      'results': {'data1_res': 'D1', 'data2_res': 2}},
...     {'bla': 'X', 'md1': 'MD1', 'md3': 3,
```

(continues on next page)

(continued from previous page)

```
...      'results': {'data1_res': 'D3'}}],
...         'omd': 0}]
>>> pprint(propagate_all_metadata(lod))
[{'bla': 'X', 'emd1': 'EMD1', 'emd2': 'EMD2', 'md1': 'MD1', 'md2': 'MD2', 'omd':␣
↪0, 'results': {'data1_res': 'D1', 'data2_res': 2}}, {'bla': 'X', 'emd1': 'EMD1',
↪ 'emd2': 'EMD2', 'md1': 'MD1', 'md3': 3, 'omd': 0, 'results': {'data1_res': 'D3
↪'}}]
```

Parameters
list_of_dicts (*list(dict)*) – list of dictionaries

Returns
list of dictionaries with common metadata in all responses

Return type
list(dict)

valjean.eponine.tripoli4.transform.**propagate_top_metadata**(*ldict*)

Extract metadata from a list of dictionaries to put them in the surrounding dictionary.

One of the dictionary key has to be `'list_responses'`, i.e. generic scores or arrays built from a list of responses where metadata have already been extracted. This case typically apply to perturbations (not IFP ones).

Parameters
ldict (*dict*) – dictionary corresponding to a response

Return type
list(dict)

valjean.eponine.tripoli4.transform.**group_to_dict**(*toks*)

Transform a named group in a dictionary with a single key (removes duplicated levels).

This method also takes into account internal dict that should be part of the main one, like in integrated_res case when units is required and score has no unit.

Parameters
toks (pyparsing.ParseResults) – *pyparsing* element

Returns
python dict corresponding to input *pyparsing* named group

valjean.eponine.tripoli4.transform.**fail_spectrum**(*s*, *loc*, *expr*, *err*)

Parsing error when all bins were at 0 if option '-a' was used while running Tripoli-4.

valjean.eponine.tripoli4.transform.**fail_parsing**(*s*, *loc*, *expr*, *err*)

Parsing error with clear message of failing line.

`common` – Common methods to build numpy arrays from dictionaries and lists

This module provides generic functions to convert parsing outputs to *NumPy* objects.

Inputs (outputs from parsers) should be python lists or dictionary, dictionary keys should be the same in all parsers...

Todo:

Not a standalone code, needs inputs.
To be tested in a more general context.

Goal

Parsing results are normally stored as lists and dictionaries but it could be easier to use other objects, as *NumPy* arrays. In our context these objects are used to represent

- spectrum results, i.e. tables splitted at least in energy, sometimes with additional splittings in time, μ and φ direction angles
- mesh results, i.e. tables splitted in space (cartesian, cylindrical, spherical depending on case), sometimes with additional splittings in energy and/or time
- Green bands results
- IFP results
- k_{eff} results
- k_{ij} results (matrices, eigenvectors and eigenvalues)

NumPy objects are useful for future calculations or plotting for example.

Spectrum and meshes

Generalities

Spectrum and meshes results use a common representation build using *`DictBuilder`*. This common representation is a **7-dimension structured array** from *NumPy*, see numpy structured array.

Kinematic dimensions are:

u, v, w
: space coordinates (typically (x, y, z), (r, θ, z) or (r, θ, φ), depending on frame reference)

e
: energy

t
: time

mu, phi
: direction angles μ and φ whose definitions can vary depending on the reference frame of direction[1].

Order should always be that one.

The result for each bin `(u, v, w, e, t, mu, phi)` is filled in a numpy structured array whose `numpy.dtype` can be:

meshes
: normally `'score'` and `'sigma'` where *sigma* is in % and *score* in its unit (not necessarly precised in the listing)

default spectrum
: `'score'`, `'sigma'`, `'score/lethargy'` where *sigma* is in %, *score* and *score/lethargy* in the unit of the score (not necessarly precised in the listing)

spectrum with variance of variance (vov)
: as default spectrum case + a fourth element named `'vov'` (no unit)

uncertainties spectrum
: in case of *covariance perturbations*, elements are `'sigma2(means)'`, `'mean(sigma_n2)'`, `'sigma(sigma_n2)'` and `'fisher test'`[2]

Spectrum and mesh results don't only consist in arrays: binning (except in space) and number of dicarded batchs are also available for example. Other optional can also be added, like integrated result on one or more dimensions. The final result of spectrum and mesh is returned as a dictionary detailed in *result_spectrum* and *result_mesh*.

Initialization

DictBuilder cannot be instantiated as it has abstract methods (pure virtual): *fill_arrays_and_bins* and *add_last_bins*. It is mother class of *MeshDictBuilder* for mesh and *SpectrumDictBuilder* for spectrum.

Initialization is done giving names of the columns (`'score'` and `'sigma'` for mesh for example) and the list of number of bins. The length of this list should be 7 as we have 7 dimensison.

```
>>> from valjean.eponine.tripoli4.common import (DictBuilder,
...     MeshDictBuilder, SpectrumDictBuilder)
>>> db = DictBuilder(['score', 'sigma'], [1,2,3,4,5,6,7])
Traceback (most recent call last):
    [...]
TypeError: Can't instantiate abstract class DictBuilder with ...
>>> mdb = MeshDictBuilder(['score', 'sigma'], [1,2,3,4,5,6,7])
```

(continues on next page)

[1] Definition of μ and φ, direction angles (see also user guide)
- μ = cos(θ)
- if ANGULAR keyword used: result only splitted in μ, with θ is defined with respect to the normal of the surface used in SURF
- if 2D_ANGULAR keyword used: result splitted in μ and φ, defined in the global frame

[2] The structured array elements in uncertainty spectrum case are:
- `'sigma2(means)'`: variance of means or σ^2 of means
- `'mean(sigma_n2)'`: mean of variances, with variance = v_n
- `'sigma(sigma_n2)'`: σ(variances) = $\sqrt{v(v_n)}$
- `'fisher test'`: this is an estimator more than a test
- Variance, or sigma_n2 is given by: $v_n = \frac{\sum_i^n (x_i - m)^2}{n(n-1)}$

(continued from previous page)

```
>>> mdb = MeshDictBuilder(['result', 'sigma'], [1,2,3,4,5,6,7])
>>> sdb = SpectrumDictBuilder(['score', 'sigma', 'score/lethargy'],
...                           [1,2,3,4,5,6,7])
```

Errors are raised if the dimension is not correct.

```
>>> mdb = MeshDictBuilder(['score', 'sigma'], [1,2,3,4,5,6])
Traceback (most recent call last):
    ...
AssertionError
```

Number of bins should be 7 (3 space dimensions, 1 energy, 1 time, 2 direction angles).

These methods initializes both the 7-dimensions numpy structured array and the arrays of bins, first stored as list for simplicity. Arrays of bins are in reality array of the edges of bins, starting by the lower one to the higher one after flipping if needed.

Filling arrays and bins

Mesh and spectrum are read from the output listing and first stored as list and dictionary following listing structure. Building `numpy.ndarray` for arrays and bins simplifies post-processing. It calls *`fill_arrays_and_bins`*.

To fill arrays and bins needed objects are outputs from the chosen parser. Some dictionary keys may be needed:

mesh:
data is a list of all meshes available. Each mesh is a dictionary that can have the following keys:

`'meshes'`
list of dictionary containing the mesh results (mandatory), dictionaries with keys:

- `{'mesh_energyrange': [], 'mesh_vals': [[], ]}` to get mesh per energy range (mandatory)
- `{'mesh_energyintegrated':, 'mesh_vals': [[], ]}` if result is also available on the full range of energy but still splitted in space (facultative)

A mesh line in the list under `'mesh_vals'` key is constructed as a list of `[[u, v, w], score, sigma]`. In `'mesh_energyrange'`, energy range is given as `['unit', e1, e2]`.

`'time_step'`
if a time splitting is available Remark: for the moment, no splitting in μ or φ are available.

`'integrated result'`
if `'time_step'` exists, integrated result can be available, meaning integrated over space and energy (not time)

spectrum:
data is a list of dictionaries containing spectrum results.

Possible keys are:

'spectrum_vals'
spectrum values, given in a list `[e1, e2, score, sigma, score/lethargy]` (mandatory)

'time_step'
if result splitted in time

'mu_angle_zone'
if result splitted in μ angle

'phi_angle_zone'
if result splitted in φ angle

'integrated_res'
if `'time_step'` exits, integrated result can also be given in time steps, so integrated over energy.

When arrays are filled, bins are also filled on their first appearance. At the end of the filling special care needs to be taken to bins. Indeed, as usual there are Nbins+1 edges. The last edge may be the lowest one or the highest one, depending on the order required in the job. So it will inserted in first position or appended to the end.

If bins are in decreasing order in one dimension (energy, time, μ or φ), arrays will be flipped in that direction. This step as to be done on all arrays stored and on the bins array to stay consistant.

If time, μ or φ grids are given, they will always appear in the same order: t -> μ -> φ. μ and φ can exist without time; time can exist alone, like μ; φ cannot exist without μ[Page 255, 1]. If more than one is present, the first one is not repeated at each step, so needs to be propagated to the next steps (instance variables `itime`, `imu` and `iphi`).

Result and use in the framework

In the framework, *MeshDictBuilder* and *SpectrumDictBuilder* are called in *convert_mesh* and *convert_spectrum*, themselves from transformation modules (transforming parsing result in *NumPy*/*python* containers. Theses methods then returns dictionaries containing the `numpy.ndarray` and other results.

mesh:
Default keys are:

'array'
7-dimensions numpy structured array with `numpy.dtype` `('score', 'sigma')`

'bins'
`collections.OrderedDict` (str, `numpy.ndarray`), order corresponds to the order of the shape in the array. Space are normally at center of the bin, while E, t, μ and φ are given as edges. If no binning is available an empty array is present.

'units'
available limits (default set, including score and sigma)

Other keys can be available:

'eintegrated_array'
7-dimensions numpy structured array with dtype `('score', 'sigma')` and list of number of bins (`lnbins`) is `[n_u, n_v, n_w, 1, n_t, 1, 1]`

'integrated'
7-dimensions numpy structured array with dtype `('score', 'sigma')` and list of number of bins (`lnbins`) is `[1, 1, 1, 1, n_t, 1, 1]`

'used_batch'
if `'integrated_res'` exists, number of used batch is also given

spectrum:
Default keys are:

'spectrum'
7-dimensions numpy structured array with `numpy.dtype` `('score', 'sigma', 'score/lethargy')` if this is a default spectrum, `numpy.dtype` will change in some cases (vov, uncertainties), see *Generalities*

'bins'
`collections.OrderedDict` (str, `numpy.ndarray`), order corresponds to the order of the shape in the array. In spectrum no space bins are given (corresponding to empty arrays), other dimensions are normally given as edges when available, else as an empty array.

'discarded_batches'
number of discarded batchs

Optional keys are:

'integrated'
7-dimensions numpy structured array with same `numpy.dtype` as `'array'` and list of number of bins (`lnbins`) is `[1, 1, 1, 1, n_t, 1, 1]` (integrated over energy)

'used_batch'
if `'integrated_res'` exists, number of used batch is also given

Other available arrays

Green bands

Green bands are stored in `numpy.ndarray` that look like the spectrum or mesh ones but with different bins and dtypes, these are 6-dimensions arrays.

The 6 dimensions are given, in order, by:

se
`'step'` in input, bin of the energy of source (source are treated in energy steps)

ns
`'source'` in input, number of the source

u, v, w
coordinates of the source, `(0, 0, 0)` if not given

e
energy of the output neutron

The result for each bin (se, ns, u, v, w, e) is filled in a numpy structured array whose `numpy.dtype` is the default spectrum one, `('score', 'sigma', 'score/lethargy')`.

Bins are also stored for all the dimensions in same order as in the array. Empty array corresponds to unused dimensions. Like in spectrum, last bins of energy (source and followed particle) are added after the main loop.

The returned dictionary contains:

'array'
: 6-dimension numpy structured array

'bins'
: collections.OrderedDict (str, numpy.ndarray) of bins in same order as array shape

'units'
: dict of units

Adjoint results

Results obtained from the calculation of the adjoint can beassociated to array like spectra or meshes or to more generic scores. Dimensions are usually a bit different. The output from Tripoli-4 is also different and has to be treated separately.

Different knid of arrays are available, depending on the type of calculation:

Generic adjoint result:
: corresponds generic scores calculated by IFP or Wielandt methods. They are returned as standard dictionary containing usual integrated results (pair score, sigma), so no array. 'Dimensions' correspond to the dictionary keys (like nucleus, family, length, etc.).

Adjoint criticality results from specific edition:
: only calculated with the IFP method. Two kind of results are for the moment available: multi-dimensions arrays, close to meshes and arrays in (volume, energy) where volume corresponds to the geometrical id of the volume. The available dimension for the multi-dimensions arrays are: X, Y, Z (only cartesian in space), φ and θ in direction, energy. No time is considered. These spectra inherit from *KinematicDictBuilder* but have a different order of bins compared to spectra and meshes and time is automatically set to 1 bin (integrated).

More details and code are available in *convert_generic_adjoint*, *AdjointCritEdDictBuilder* and *VolAdjCritEdDictBuilder*.

Nu and (Z,A) spectrum

Spectra indexed by the number of neutrons produced in fission (Nu) and indexed by the (Z,A) of the produced fission products (isotopes) as also available as spectra.

The results are given as standard array results: a dictionary with the usual keys (`'array'`, `'bins'`, `'units'`).

More details in *convert_nu_spectrum* and *convert_za_spectrum*.

Scores on spherical harmonics

These scores are foreseen to be passed to deterministic codes. They are calculated on real spherical harmonics, using the Schmid semi-normalized harmonics as described in [SHTools] to be consistent with Apollo3. Thus the associated functions are not only Legendre polynomials but also contains a factor $cos(m\phi)$ for $m > 0$ and a factor $sin(|m|\phi)$ for $m < 0$.

In practice, they are discretized on a space mesh ((u, v, w) coordinates), on incident energy bins (ie), on energy bins (e) and on the order of the moments ((l, m)). l moments go from 0 to L (so there are $L + 1$ values), m ones go from $-l$ to l, ($2L + 1$ values).

The eight scores available in such description are converted in a single array of 7 dimensions (u, v, w, ie, e, l, m). The forbidden (l, m) pairs ($m > |l|$) are set to *numpy.nan*. A *mask* is applied to the resulting Dataset.

Note that m ids goes from 0 to 2L+1 in slicing, corresponding to bins values $-m$ to m.

All the arrays have 7 dimensions, but incident energy bins are only relevant for the scattering matrix. The fission spectrum only allows $l = 0, m = 0$.

k_{eff} results

Only k_{eff} as generic response are converted in *NumPy* objects; historical k_{eff} block is stored in a dictionary (see *valjean.eponine.tripoli4.grammar*).

In the generic response case, results (value, σ) are available for 3 estimators: KSTEP, KCOLL and KTRACK. Their correlation coefficients, combined values, combined σ (in %) and the full combination result are also given. This means that results given are in reality a matrix. One choice in order to store k_{eff} results is to use `numpy.ndarray` seen as matrix (*convert_keff_with_matrix*), the other one uses more standard arrays (*convert_keff*).

Conversion to matrices

The 3 estimators are always considered in the listing order KSTEP, KCOLL, KTRACK, so KSTEP = 0, KCOLL = 1 and KTRACK = 2.

Three arrays are filled:

'keff_matrix'
: symmetric matrix 3×3, with k_{eff} result for each estimator on diagonal and combined values off-diagonal (for 2 estimators)

'correlation_matrix'
: symmetric matrix 3×3, with 1 on diagonal and correlation cofficient off-diagonal (for 2 estimators)

'sigma_matrix'
: symmetric matrix 3×3 with σ in % for each estimator on diagonal and combined σ in % off-diagonal (for 2 estimators)

In summary:

- for k_{eff} and σ matrices (replace k_{eff} by σ in 2^d case, cb stands for combined):

	KSTEP	KCOLL	KTRACK
KSTEP	k_{eff} (KSTEP)	cb k_{eff} (KSTEP, KCOLL)	cb k_{eff} (KSTEP, KTRACK)
KCOLL	cb k_{eff} (KSTEP, KCOLL)	k_{eff} (KCOLL)	cb k_{eff} (KCOLL, KTRACK)
KTRACK	cb k_{eff} (KSTEP, KTRACK)	cb k_{eff} (KCOLL, KTRACK)	k_{eff} (KTRACK)

- the correlation matrix:

	KSTEP	KCOLL	KTRACK
KSTEP	1	corr(KSTEP, KCOLL)	corr(KSTEP, KTRACK)
KCOLL	corr(KSTEP, KCOLL)	1	corr(KCOLL, KTRACK)
KTRACK	corr(KSTEP, KTRACK)	corr(KCOLL, KTRACK)	1

Values are set to *numpy.nan* if not converged (string `"Not converged"` appearing in the listing).

These arrays can be easily converted to matrices if matrix methods are needed but array is easier to initialized and more general.

The method *`convert_keff_with_matrix`* takes as input the generic k_{eff} response as a dictionary and returns a dictinary containing different keys:

- the number of batchs used under `'used_batches'`;
- the 3 matrices mentioned above (`'keff_matrix'`, `'correlation_matrix'` and `'sigma_matrix'`) and the list of estimators (`['KSTEP', 'KCOLL', 'KTRACK']` by default) are stored under the common key `'keff_per_estimator'` as a dictionary;
- the full combination result (k_{eff} and σ in %) under `'keff_combination'` key

Not converged cases are taken into account and return a key `'not_converged'`.

Conversion to standard arrays

The conversion closer to the output listing is done in *`convert_keff`*. A dictionary is built with the following elements:

`'used_batch'`
: number of batchs used

`'full_comb_estimation'`
: full combination result (k_{eff} and σ in %), like in *Conversion to matrices*

`'res_per_estimator'`
: dictionary with estimator as key and numpy structured array with `dtype = ('keff', 'sigma')` as value

`'correlation_matrix'`
: dictionary with tuple as key and numpy structured array as value, `('estimator1', 'estimator2'): numpy.array('correlations', 'combined values', 'combined sigma%')`

In correlation matrix diagoanl is set to 1 and not converged values (str) are set to *numpy.nan*. If the full combination did not converged the string is kept.

k_{ij} results

k_{ij} matrix gives the number of neutrons produced by fission in the volume i from a neutron emited in the volume j. Its highest eigenvalue is equal to the k_{eff} of the system. The corresponding eigenvector represents the neutrons sources in the volumes (necessarly containing fissile metrial). For more details, see user guide.

Different k_{ij} results can be available:

- list of k_{ij} sources in *convert_kij_sources*
- k_{ij} matrix and associated results in *convert_kij_result*
- k_{ij} estimation in historical k_{eff} block (k_{ij} is an additional estimator in that case) in *convert_kij_keff*

k_{ij} sources

In that case the input dictionary is returned with a conversion of the `'kij_sources_vals'` as a `numpy.ndarray`. Its length corresponds to the number of volumes.

k_{ij} matrix (result)

The k_{ij} matrix results block contains various results including k_{ij} eigenvalues, k_{ij} eigenvectors and k_{ij} matrix that will be converted in `numpy.ndarray` or `numpy.matrix`. The size of the arrays depends on the number of volumes containing fissle material, N.

The returned object is a dictionary containing the following keys and objects:

`'used_batches'`
: number of batchs used (*int*)

`'kij_mkeff'`
: result of k_{ij}-k_{eff} (*float*), where k_{ij} is the hightest eigenvalue of k_{ij}

`'kij_domratio'`
: dominant ratio (*float*), ratio between the hightest k_{ij} eigenvalue and the next one

`'kij_reigenval'`
: `numpy.ndarray` of N **complex** numbers (real and imaginary parts given in the listings) corresponding to the **right** eigenvalues.

`'kij_reigenvec'`
: `numpy.ndarray` of N vectors of N elements corresponding to **right** eigenvectors.

`'kij_matrix'`
: `numpy.matrix` of N×N being the k_{ij} matrix.

k_{ij} in k_{eff} block

k_{ij} results are also present in the "historical" k_{eff} block, as an additional estimator. Results are presented in a different way and are different... Typical results are k_{ij} - k_{eff}, the eigenvector corresponding to the best estimation, k_{ij} matrix, standard deviation matrix and sensibility matrix.

The returned object is a dictionary with the following keys (faculative can be specified):

'keff_estimator'
: name of the estimator (`str`), `'KIJ'` here

'results'
: usual results block, built here for once containing the following dictionary (same keys as in the previous case when possible):

  'used_batches'
  : number of batchs used to calculate the k_{ij} (`int`)

  'kij_mkeff'
  : result of k_{ij}-k_{eff} (*float*)

  'space_bins'
  : **facultative**, list of N volumes/mesh elements considered (`numpy.ndarray` of

    - `int` for volumes,
    - `tuple` of `int` (u, v, w) for mesh elements,

  'kij_leigenvec'
  : eigenvector corresponding dominant **left** eigenvector (`numpy.ndarray` of N elements)

  'kij_matrix'
  : k_{ij} matrix (N×N `numpy.matrix`)

  'kij_stddev_matrix'
  : standard deviation matrix (N×N `numpy.matrix`)

  'kij_sensibility_matrix'
  : sensibility matrix (N×N `numpy.matrix`)

Module API

valjean.eponine.tripoli4.common.**profile**(*func*)
: No memory profiling if "mem" not in arguments of the command line.

class valjean.eponine.tripoli4.common.**DictBuilder**(*colnames*, *lnbins*)
: General class to build dictionaries.

  This class implements a pattern for array results storage as dictionaries. It transforms the Tripoli-4 array strings in numpy arrays of a given number of dimensions, depending on the kind of array (spectrum, mesh, IFP result, etc.). General methods are implemented, mandatory methods are implemented as abstract and need to be derived in daughter classes.

__init__(*colnames*, *lnbins*)

Initialization of DictBuilder.

Parameters

- **colnames** (*list(str)*) – name of the columns/results (e.g. `'score'` and `'sigma'` for mesh, or `'score'`, `'sigma'`, `'score/lethargy'` for spectrum)
- **lnbins** (*list(int)*) – number of bins for each dimension

add_array(*name*, *colnames*, *lnbins*)

Add a new array to dictionary arrays with key name.

Parameters

- **name** (*str*) – name of the new array (integrated_res, etc.)
- **colnames** (*tuple(str)*) – list of the columns names (score, sigma, etc.)
- **lnbins** (*list(int)*) – number of bins in each dimension

abstract add_last_bins(*data*)

Add last bins based on keywords presence in data.

Parameters
data (*list*) – mesh or spectrum results

_flip_bins_for_dim(*dim*, *axis*)

Flip bins for dimension dim.

Parameters

- **dim** (*str*) – dimension (examples: ‘e’, ‘t’, ‘mu’, ‘phi’)
- **axis** (*int*) – axis of the dimension (example: ‘e’ -> 3, ‘t’ -> 4, ‘mu’ -> 5, ‘phi’ -> 6)

convert_bins_to_increasing_arrays()

Flip bins if given in decreasing order in the output listing.

Depending on the required grid (GRID or DECOUPAGE) energies, times, mu and phi can be given from upper edge to lower edge. This is not convenient for post-traitements, especially plots. They have to be flipped at a moment, here or later, easiest is here, and all results will look the same :-).

This function calls an internal function and needs to match the dimension with the number of the axis: (‘e’ → 3, ‘t’ → 4, ‘mu’ → 5, ‘phi’ → 6)

abstract fill_arrays_and_bins(*data*)

Fill arrays and bins for spectrum or mesh data.

Parameters
data (*list*) – mesh or spectrum results

class valjean.eponine.tripoli4.common.**KinematicDictBuilder**(*colnames*, *lnbins*)

Class to build the dictionary for spectrum and mesh results as 7-dimensions structured arrays. 7-dimensions are: space (3, written `'u'`, `'v'`, `'w'`), energy (`'e'`), time (`'t'`), mu (`mu'`) and phi (`'phi'`) (direction angles).

This class has 2 abstract methods, *_add_last_energy_bin* and *fill_arrays_and_bins*, so it cannot be initialized directly.

VAR_FLAG
{'t': 'time_step', 'mu': 'mu_angle_zone', 'phi': 'phi_angle_zone'}: correspondance dictionary between internal name of dimensions and names in listings

__init__(*colnames*, *lnbins*)
Initialization of KinematicDictBuilder.

Parameters

- **colnames** (*list(str)*) – name of the columns/results (e.g. 'score' and 'sigma' for mesh, or 'score', 'sigma', 'score/lethargy' for spectrum)
- **lnbins** (*list(int)*) – number of bins for each dimension

abstract _add_last_energy_bin(*data*)
Add last bin in energy from spectrum or mesh.

Parameters
data (*list*) – mesh or spectrum results

_add_last_bin_for_dim(*data*, *dim*, *lastbin*)
Add last bin for the dimension dim. Depending on order of the bins the last one will be inserted as first bin or added as last bin.

Parameters

- **data** (*list*) – mesh or spectrum results
- **dim** (*str*) – dimension where the bin will be added (t, mu, phi)
- **lastbin** (*int*) – index of the bin in mesh or spectrum containing the missing edge of the bins

add_last_bins(*data*)
Add last bins in energy, time, mu and phi direction angles. Based on keywords presence in data.

Parameters
data (*list*) – mesh or spectrum results

abstract fill_arrays_and_bins(*data*)
Fill arrays and bins for spectrum or mesh data.

Parameters
data (*list*) – mesh or spectrum results

exception valjean.eponine.tripoli4.common.**MeshDictBuilderException**
Exception to mesh builder

class valjean.eponine.tripoli4.common.**MeshDictBuilder**(*colnames*, *lnbins*)
Class specific to mesh dictionary -> mainly filling of bins and arrays.

This class inherites from KinematicDictBuilder, see *KinematicDictBuilder* for initialization and common methods.

__init__(*colnames*, *lnbins*)
Initialization of MeshDictBuilder.

Parameters

- **colnames** (*list(str)*) – name of the columns/results (e.g. `'score'` and `'sigma'` for mesh, or `'score'`, `'sigma'`, `'score/lethargy'` for spectrum)
- **lnbins** (*list(int)*) – number of bins for each dimension

As no bins (centers or edges) are given for space mesh, they are initialised to index in mesh (in the considered direction), starting at 0 by convention.

classmethod from_data(*data*)

Initialize MeshDictBuilder from data.

fill_space_bins(*nb_tokens*, *vals*)

Fill the mesh space bins.

Two different cases are possible:

- the default one, where only the cell indices are given: the space bins are set to all possible cell index in the 3 dimensions. For example: if there are 3 cells in *u*, the bins will be 0, 1, 2. Only center of bins are given here (no possibility of calculation of a width). In that case the mesh contains 3 tokens: a comma-separated list of cell indices (without intervening whitespace), the value and the sigma.
- a standard MESH was required with the option `MESH_INFO` in Tripoli-4: center of cells are given on the 3 dimensions, the space bins will be set to these values if the mesh is Cartesian and its axes coincide with the coordinate axes. In MESH_INFO case the mesh line contains 6 or 7 tokens, depending on Tripoli-4 version: the comma-separated list of cell indices (as above), the three space coordinates of the midpoint of the cell, [the cell volume], the value and the sigma. When the cell coordinates are not aligned on the axes, the bins stay the cell indices and the coordinates stay available.

Parameters

- **nb_tokens** (*int*) – number of tokens by line of mesh result
- **vals** (*list*) – mesh results

_fill_mesh_array(*meshvals*, *name*, *ebin*)

Fill mesh array.

Parameters

- **meshvals** (*list*) – mesh data for a given energy bin `[[[u, v, w], score, sigma],...]`
- **name** (*str*) – name of the array to be filled ('default', 'eintegrated_mesh') for the moment
- **ebin** (*int*) – energy bin to fill in the array

_fill_entropy_array(*meshvals*, *ebin*)

Fill mesh array.

Parameters

- **meshvals** (*list*) – mesh data for a given energy bin `[[[u, v, w], score, sigma],...]`
- **ebin** (*int*) – energy bin to fill in the array

fill_arrays_and_bins(*data*)

Fill arrays and bins for mesh data.

Parameters
data (*list*) – mesh results

Different arrays can be filled. Current possibilities are:

- `'default'` (mandatory)
- `'eintegrated_mesh'` (facultative, integrated over energy, still splitted in space)
- `'integrated_res'` (over energy and space, splitted in time)

fill_score_units(*data*)

Fill score units if available in data, else leave to unknown.

fill(*nb_elts*, *data*)

Fill data in mesh.

_add_last_energy_bin(*data*)

Add last bin in energy from mesh data.

Parameters
data (*list*) – mesh results

exception valjean.eponine.tripoli4.common.**SpectrumDictBuilderException**

Exception to spectrum builder (bad bins)

class valjean.eponine.tripoli4.common.**SpectrumDictBuilder**(*colnames*, *lnbins*)

Class specific to spectrum dictionary -> mainly filling of bins and arrays.

This class inherites from KinematicDictBuilder, see *KinematicDictBuilder* for initialization and common methods.

__init__(*colnames*, *lnbins*)

Initialization of KinematicDictBuilder.

Parameters

- **colnames** (*list(str)*) – name of the columns/results (e.g. `'score'` and `'sigma'` for mesh, or `'score'`, `'sigma'`, `'score/lethargy'` for spectrum)
- **lnbins** (*list(int)*) – number of bins for each dimension

fill_arrays_and_bins(*data*)

Fill arrays and bins for spectrum data.

Parameters
data (*list*) – spectrum results

Current arrays possibly filled are:

- `'default'` (mandatory)
- `'integrated_res'` (over energy, splitted in time for the moment)

_check_bins(*vals*, *ienergy*)

Check bins validity.

_add_last_energy_bin(*data*)

Add last bin in energy from spectrum.

Parameters
data (*list*) – spectrum results

fill_score_units(*data*)

Fill score units if available in data, else leave to unknown.

valjean.eponine.tripoli4.common.**_get_number_of_bins**(*spectrum*)

Get number of bins (time, mu and phi angles and energy).

Parameters
spectrum – input spectrum (full as various levels of list or dictionary may be needed.

Returns

4 integers in following order

nphibins
number of bins in phi angle, default = 1

nmubins
number of bins in mu angle, default = 1

ntbins
number of bins in time, default = 1

nebins
number of bins in energy, no default

Mu and phi angle are angles relative to the direction of the particle.

valjean.eponine.tripoli4.common.**convert_spectrum**(*spectrum*, *colnames=('score', 'sigma', 'score/lethargy')*)

Convert spectrum results in 7D NumPy structured array.

Parameters

- **spectrum** (*list*) – list of spectra. Accepts time and (direction) angular grids.
- **colnames** (*list(str)*) – list of the names of the columns. Default = `['score', 'sigma', 'score/lethargy']`

Returns

dictionary with keys and elements

- `'array'`: 7 dimensions NumPy structured array with related binnings as NumPy arrays `v[u, v, w, E, t, mu, phi] = ('score', 'sigma', 'score/lethargy')`
- `'bins'`: `collections.OrderedDict` of the available bins
- `'units'`: dict containing units of dimensions (bins), score and sigma
- `'eintegrated_array'`: 7 dimensions NumPy structured array `v[u, v, w, E, t, mu, phi] = ('score', 'sigma')`; facultative, seen when time required alone and sometimes when neither time nor mu nor phi are required

valjean.eponine.tripoli4.common.**_get_number_of_space_bins**(*meshvals*)

Get number of space bins used in meshes.

This function is mainly used when meshes are not entirely saved (tests, or useless in the considered case). The limit on the number of lines of mesh in the listing does not necessarly match a completed mesh dimension.

Parameters

meshvals (*list*) – list of meshes, with mesh `[[u, v, w] score sigma]` u, v and w being the space coordinates

Returns

3 integers in following order

nubins
number of bins in the u dimension

nvbins
number of bins in the v dimension

nwbins
number of bins in the w dimension

valjean.eponine.tripoli4.common.**get_energy_bins**(*meshes*)

Get the number of energy bins for mesh.

Parameters

meshes (*list*) – mesh, list of dictionaries, at least one should have the key `'mesh_energyrange'`

valjean.eponine.tripoli4.common.**convert_mesh**(*meshres*)

Convert mesh in 7-dimensions NumPy array.

Parameters

meshres (*list*) – Mesh result constructed as: `[{'time_step': [], 'meshes': [], 'integrated_res': {}}, {}]`, see *Filling arrays and bins* for more details.

Returns

python dictonary with keys

- `'array'`: NumPy structured array of dimension 7 `v[u, v ,w, E, t, mu, phi] = ('score', 'sigma')`
- `'bins'`: `collections.OrderedDict` of the available bins
- `'units'`: dict containing units of dimensions (bins), score and sigma
- `'eintegrated_array'`: 7-dimensions NumPy structured array `v[u,v,w,E,t,mu,phi] = ('score', 'sigma')` corresponding to mesh integrated on energy (facultative)
- `'integrated'`: 7 dimensions NumPy structured array `v[u,v,w,E,t,mu,phi] = (score, sigma)` corresponding to mesh integrated over energy and space; *facultative*, available when time grid is required (so corresponds to integrated results splitted in time)
- `'used_batches'`: number of used batchs (only if integrated result)

class valjean.eponine.tripoli4.common.**NuSpectrumDictBuilder**(*colnames*, *lnbins*)

Class specific to spectrum dictionary -> mainly filling of bins and arrays.

This class inherites from DictBuilder, see *DictBuilder* for initialization and common methods.

__init__(*colnames*, *lnbins*)

Initialization of DictBuilder.

Parameters

- **colnames** (*list(str)*) – name of the columns/results (e.g. `'score'` and `'sigma'` for mesh, or `'score'`, `'sigma'`, `'score/lethargy'` for spectrum)
- **lnbins** (*list(int)*) – number of bins for each dimension

fill_arrays_and_bins(*data*)

Fill arrays and bins for spectrum data.

Parameters
data (*list*) – spectrum results

Current arrays possibly filled are:

- `'default'` (mandatory)
- `'integrated_res'` (over nu)

_check_bins(*vals*, *inu*)

Check bins validity.

add_last_bins(*data*)

Add last bin in nu from spectrum.

Parameters
data (*list*) – spectrum results

fill_score_units(*data*)

Fill score units if available in data, else leave to unknown.

valjean.eponine.tripoli4.common.**convert_nu_spectrum**(*spectrum*, *colnames=('score', 'sigma')*)

Convert nu spectrum results in 1D NumPy structured array.

Parameters

- **spectrum** (*list*) – list of spectra. Accepts time and (direction) angular grids.
- **colnames** (*list(str)*) – list of the names of the columns. Default = `['score', 'sigma']`

Returns

dictionary with keys and elements

- `'array'`: 1 dimension NumPy structured array with related binnings as NumPy arrays `v[nu] = ('score', 'sigma')`
- `'bins'`: `collections.OrderedDict`, nu binning
- `'units'`: dict containing units of dimensions (bins), score and sigma

- `'integrated_array'`: 1 dimension NumPy structured array `v[nu] = ('score', 'sigma')`

class valjean.eponine.tripoli4.common.**ZASpectrumDictBuilder**(*colnames*, *bins*)

Class specific to spectrum dictionary to parse arrays indexed by Z and A numbers (isotopes).

This class inherites from DictBuilder, see *DictBuilder* for initialization and common methods.

__init__(*colnames*, *bins*)

Initialization of ZASpectrumDictBuilder.

Parameters

- **colnames** (*list(str)*) – name of the columns/results (`'score'` and `'sigma'` in the current case)
- **bins** (`collections.OrderedDict` of (str, `numpy.ndarray` (int))) – Z, A bins

fill_arrays_and_bins(*data*)

Fill arrays and bins for spectrum data.

Parameters
data (*list*) – spectrum results

Current arrays possibly filled are:

- `'default'` (mandatory)
- `'integrated_res'` (over Z and A, i.e. all isotopes)

add_last_bins(*data*)

Add last bin in Z,A from spectrum, not applicable in this case.

Parameters
data (*list*) – spectrum results

fill_score_units(*data*)

Fill score units if available in data, else leave to unknown.

valjean.eponine.tripoli4.common.**_get_za_bins**(*values*)

Determine the bins in Z, A before filling the array from the values.

Parameters
values (*list*) – spectrum results

Returns
bins: Z, A bins as a `collections.OrderedDict` of (str, `numpy.ndarray` (int))

valjean.eponine.tripoli4.common.**convert_za_spectrum**(*spectrum*, *colnames=('score', 'sigma')*)

Convert nu spectrum results in 1D NumPy structured array.

Parameters

- **spectrum** (*list*) – list of spectra. Accepts time and (direction) angular grids.
- **colnames** (*list(str)*) – list of the names of the columns. Default = `['score', 'sigma']`

Returns

dictionary with keys and elements

- `'array'`: 1 dimension NumPy structured array with related binnings as NumPy arrays `v[Z, A] = ('score', 'sigma')`
- `'bins'`: `collections.OrderedDict`, Z and A binnings
- `'units'`: dict containing units of dimensions (bins), score and sigma
- `'integrated_array'`: 1 dimension NumPy structured array `v[Z, A] = ('score', 'sigma')`

Remark: no call to add_last_bins or convert_bins_to_increasing_arrays is done here as no energy, time or space bins are given for the moment.

`valjean.eponine.tripoli4.common.`**`convert_keff_with_matrix`**(*res*)

Convert k_{eff} results in NumPy matrices.

Parameters
res (*dict*) – k_{eff} results

Returns
dict filled as:

```
{'used_batches': int,
 'keff_per_estimator': {'estimators': [str],
                        'keff_matrix': numpy.array,
                        'correlation_matrix': numpy.array,
                        'sigma_matrix': numpy.array},
 'keff_combination': numpy.array}
```

`valjean.eponine.tripoli4.common.`**`convert_keff`**(*res*)

Convert k_{eff} results in dictionary containing NumPy objects.

Parameters
res (*dict*) – keff results

Returns
dict containing

```
{'used_batches': int, 'estimators': [str, ],
 'full_comb_estimation': numpy.array,
 'res_per_estimator': {'estimator': numpy.array, },
 'correlation_matrix': {('estimator1', 'estimator2'): numpy.array, }}
```

See *Conversion to standard arrays* for more details.

`class` `valjean.eponine.tripoli4.common.`**`GreenBandsDictBuilder`**(*colnames*, *lnbins*)

Class to build Green bands spectrum results.

`__init__`(*colnames*, *lnbins*)

Initialisation of GreenBandsDictBuilder.

`add_last_bins`(*data*)

Add last bins from source energy bins and energy bins. Also remove duplicates in source number and source tabulations.

Parameters
data – Green bands results

fill_arrays_and_bins(*data*)

Fill arrays and bins from Green bands result.

Parameters
data – Green bands results

valjean.eponine.tripoli4.common.**_get_gb_nbins**(*gbres*)

Get the number of bins for Green bands results.

Parameters
gbres – Green bands results

Returns
tuple(int)

valjean.eponine.tripoli4.common.**convert_green_bands**(*gbs*)

Convert Green bands contribution results in arrays, using same schema as spectrum or mesh.

Parameters
gbs – Green bands results

Returns
dict

valjean.eponine.tripoli4.common.**convert_generic_adjoint**(*res*)

Convert adjoint results in association of dictionaries and NumPy array.

Parameters
res (*list*) – Adjoint result got thanks to IFP or Wielandt method to be converted

Returns

list(dict) each dictionary containing

- metadata like nucleus name, family number or cycle length
- data saved as `'integrated_res'`

Units, if available, and used batch are also saved under the integrated result.

This structure is compatible with the browser and the index. Selections are done like in the default score cases.

valjean.eponine.tripoli4.common.**convert_generic_kinetic**(*res*)

Convert kinetic results into association of dictionaries and NumPy array.

Parameters
res (*list*) – parsed tokens

Returns

list(dict) each dictionary containing

- metadata like nucleus name, family number or cycle length
- data saved as `'integrated_res'`

Units, if available, and used batch are also saved under the integrated result.

This structure is compatible with the browser and the index. Selections are done like in the default score cases.

exception valjean.eponine.tripoli4.common.**AdjointCritEdDictBuilderException**
Exception to adjoint criticality edition array builder (bad bins)

class valjean.eponine.tripoli4.common.**AdjointCritEdDictBuilder**(*colnames*, *bins*)
Specific class to build IFP adjoint criticality edition results as a KinematicDictBuilder.

__init__(*colnames*, *bins*)
Initialisation of AdjointCritEdDictBuilder.

Caution: in that case bins are direclty sent, not 'only' their number as done per default in *DictBuilder*.

Order of the kinematic variables is also different from the usual spectra or mesh. Time is a 'fake' dimension (only one bin available, no splitting allowed there).

Dimensions are then, in bins order and array order (seen in shape): `('X', 'Y', 'Z', 'Phi', 'Theta', 'E', 't')`.

Dimensions not present in the screened result are set to empty array in the bins `collections.OrderedDict` and to 1 in the array shape.

_fill_array(*data*)
Fill array of results from IFP adjoint criticality edition.

Results are looping in the following order: X, Y, Z, Phi, Theta, E. This is then the order to fill the array (indices).

fill_arrays_and_bins(*data*)
Only fill array in IFP adjoint criticality edition case.

_add_last_energy_bin(*data*)
Add last bin in energy from spectrum or mesh.

Parameters
data (*list*) – mesh or spectrum results

class valjean.eponine.tripoli4.common.**VolAdjCritEdDictBuilder**(*colnames*, *bins*)
Class to build spectrum per volume instead of per kinematic variables. E is still present.

__init__(*colnames*, *bins*)
Initialisation of VolAdjCritEdDictBuilder.

Caution: in that case bins are direclty sent, not 'only' their number as done per default in *DictBuilder*.

Two kinds of bins are expected: Vol (volume id in geometry) and E (energy).

_fill_array(*data*)
Fill array of results from IFP adjoint criticality edition when spectrum is given by volume.

Results are looping in the following order: Vol, E. This is then the order to fill the array (indices).

fill_arrays_and_bins(*data*)
Only fill array in IFP adjoint criticality edition case.

add_last_bins(*data*)

Add last bins based on keywords presence in data.

Parameters
data (*list*) – mesh or spectrum results

valjean.eponine.tripoli4.common.**_get_ace_kin_bins**(*columns, values*)

Initialize bins for IFP adjoint criticality edition.

valjean.eponine.tripoli4.common.**_get_ace_vol_bins**(*values*)

Initialize bins for IFP adjoint criticality edition.

valjean.eponine.tripoli4.common.**_crit_edition_dict_builder**(*columns, values*)

Return the needed DictBuilder for IFP adjoint criticality edition according to columns names.

Parameters

- **columns** (*list(str)*) – columns names
- **values** (*list(int,float)*) – bins edges or centers

Returns
AdjointCritEdDictBuilder or *VolAdjCritEdDictBuilder*.

valjean.eponine.tripoli4.common.**convert_crit_edition**(*res*)

Convert IFP adjpint criticality edition results in standard kinematic result.

valjean.eponine.tripoli4.common.**convert_kij_sources**(*res*)

Convert k_{ij} sources result in python dictionary in which k_{ij} sources values are converted in a NumPy array.

Parameters
res (*dict*) – k_{ij} sources

Returns
same dictionary with `numpy.ndarray` for k_{ij} sources values

valjean.eponine.tripoli4.common.**convert_kij_result**(*res*)

Convert k_{ij} result in NumPy objects and return a dictionary.

Parameters
res (*dict*) – k_{ij} result with keys `'used_batches'`, `'kij_eigenval'`, `'kij_eigenvec'`, `'kij_matrix'`

Returns
dictionary containing the same keys but with different types:

```
{'used_batches_res': int, 'kij_mkeff_res': float,
 'kij_domratio_res': float, 'kij_reigenval_res': numpy.array,
 'kij_reigenvec_res': numpy.array, 'kij_matrix_res': numpy.array}
```

For more details see *kij matrix (result)*.

This result returns **right** eigenvalues and **right** eigenvectors (meaning of the 'r' in the key).

valjean.eponine.tripoli4.common.**convert_kij_keff**(*res*)

Convert matrices in NumPy array or matrix when estimating k_{eff} from k_{ij}

Parameters
res (*dict*) – k_{ij} result from k_{eff} result block

Returns
dictionary containing *NumPy* arrays:

```
{'keff_estimator': str,
 'results': {'used_batches_res': int,
             'kij_mkeff': float (kij result - keff),
             'space_bins_res': numpy.array of int with shape (nbins,) or
               (nbins, 3), the latter case corresponding to space mesh,
             'kij_leigenvec_res': numpy.array,
             'kij_matrix_res': numpy.array,
             'kij_stddev_matrix_res': numpy.array,
             'kij_sensibility_matrix_res': numpy.array}}
```

Key `'space_bins'` is facultative.

For more details see *kij in keff block*.

The eigenvector is here the dominant **left** eigenvector.

class valjean.eponine.tripoli4.common.**SensitivityDictBuilder**(*colnames*, *lnbins*)

Class to build sensitivity results dictionary.

__init__(*colnames*, *lnbins*)

Initialization of DictBuilder.

Parameters

- **colnames** (*list(str)*) – name of the columns/results (e.g. `'score'` and `'sigma'` for mesh, or `'score'`, `'sigma'`, `'score/lethargy'` for spectrum)
- **lnbins** (*list(int)*) – number of bins for each dimension

fill_arrays_and_bins(*data*)

Fill arrays and bins for sensitivities.

Fill integrated result (written as energy integrated but integrated over all dimensions) in the `'integrated_res'` array, that will be in parallel of the default sensitivity result (a priori always here).

add_last_bins(*data*)

Add last bins in incident energy (E), energy (E') and direction cosine (μ). All orders are possible.

Parameters
data (*list*) – results as a list of dictionaries

valjean.eponine.tripoli4.common.**convert_sensitivities**(*res*)

Convert sensitivities to dictionary containing `numpy.ndarray`.

Parameters
res – result

Returns
list(dict) containing the results and the associated metadata.

The dictionary contains a structured array of 3 dimensions: incident energy `'einc'`, exiting energy `'e'` and direction cosine `'mu'`. The dtype is `('score', 'sigma')`.

Bins are filled in an `collections.OrderedDict` always containing the 3 keys `'einc'`, `'e'`, `'mu'` in the order of the bins in the `numpy.ndarray`.

class valjean.eponine.tripoli4.common.**SphericalHarmonicsDictBuilder**(*colnames*, *bins*, *corr_names*)

Class specific to results on spherical harmonics.

This class inherites from DictBuilder, see *DictBuilder* for initialization and common methods.

Arrays are indexed by (u, v, w) for space, ie for incident energy, e for energy, l for moment and m for sub_moment with $L+1$ values of l and $2L+1$ values of m, L being the maximum value of l.

__init__(*colnames*, *bins*, *corr_names*)

Initialization of SphericalHarmonicsDictBuilder.

Parameters

- **colnames** (*list(str)*) – name of the columns/results (`'score'` and `'sigma'` in the current case)
- **bins** (*collections.OrderedDict*) – (u, v, w, ie, e, l, m)
- **corr_names** (*dict*) – correspondence table for score names

fill_arrays_and_bins(*data*)

Fill arrays and bins for spherical harmonics results.

Parameters
data – parsed result from *pyparsing*

add_last_bins(*data*)

Add last bins in incident energy and energy.

Parameters
data (*dict*) – last result

fill_space_bins()

Fill spaces bins based on array shape.

fill_moments_bins()

Fill moments bins.

- l goes from 0 to L
- m goes from $-L$ to L

reduced_bins(*score*)

Reduce bins according to score.

Bins are initialized with highest dimensions possible for each. This method makes them match with the array shape. For example, remove incident energy bins keeping first and last edges in most case. A special case is done for fission_spectrum score that only has one l value, so $l = 0$, $m = 0$.

Parameters
score (*str*) – score name

Returns
bins

Return type
collections.OrderedDict

valjean.eponine.tripoli4.common.**_build_shr_table**(*scores*)

Build correspondance table of valjean names for spherical harmonics results and Tripoli4 ones.

Parameters
scores (*list*) – score names

Returns
correspondance table between valjean names and Tripoli-4 ones

Return type
dict

valjean.eponine.tripoli4.common.**_get_nb_shr_bins**(*res*)

Extract number of bins.

Parameters
res – parsed result from *pyparsing*

Return type
list(int)

valjean.eponine.tripoli4.common.**convert_spherical_harmonics**(*res*, *colnames=('score', 'sigma')*)

Convert results on spherical harmonics to dictionary containing `numpy.ndarray`.

Parameters

- **res** – result
- **colnames** (*list(str)*) – name of the columns/results

Returns
dict containing the results and the associated metadata.

valjean.eponine.tripoli4.common.**convert_list_to_tuple**(*liste*)

Convert nested list to nested tuple (to get imutable object).

If list is not nested just convert it to tuple.

Parameters
liste – result as a liste

Returns
(nested) tuple

parse_debug – Debug parsing of Tripoli-4 outputs

Extension of *parse* module for debugging.

Main difference is the possibility of using the `end_flag` parameter in the *scan*.

class valjean.eponine.tripoli4.parse_debug.**ParserDebug**(*jddname*, ***, *end_flag=''*, *ofile=''*)

Scan up to the end flag then parse. For parsing debugging.

__init__(*jddname*, ***, *end_flag=''*, *ofile=''*)

Initialize the *ParserDebug* object.

Parameters

- **jddname** (*str*) – path to the Tripoli-4 output
- **batch** (*int*) – batch to read (-1 = last, 0 = all, then X)
- **end_flag** (*str*) – optional end flag to stop scanning and parsing (empty string per default)

It also initalize the result of *scan.Scanner* to `None` and the parsing result, from *pyparsing*, to `None`.

If the path contains the `"PARA"` string, the checks will be done for parallel mode.

parse_from_number(*batch_number*, *name*='')

Parse from batch index or batch number.

Per default the last batch is parsed (index = -1).

Parameters
batch_number (*int*) – the number of the batch to parse

Returns
list(dict)

check_parsing(*parsed_res*)

Check if parsing went to the end:

- if the end flag was not precised, a time should appear in the last result;
- if not, no check can be performed as the end flag can be anywhere, even "transformed" during parsing.

Print a logger message if not found but don't block access to the parsing result (this can help to find the issue).

5.4.4 apollo3 – Data reading for Apollo3

Introduction

This sub-package transforms some results from Apollo3 in the standard data format of *valjean*. For the moment only results stored in the HDF5 format are concerned.

Two possibilities are given to the user, available in two different modules:

- Reading and storing all results in the HDF5 file in a *Browser* with *hdf5_reader*. All results are converted in *Dataset*, metadata allow selection in *Browser*. Further selections are possible with the *Browser* as usual.
- Picking only results of interest for the user (normally largely quicker) with *hdf5_picker*. All results are returned as *Dataset*. No *Browser* is built.

Main modules

`hdf5_reader` – Read an Apollo3 HDF5 and store the results

This module is designed to read all results contained an standard HDF5 output file from Apollo3 (with keff, fluxes and rates) and to store them in memory as a *Browser*. This makes it possible to explore the content of the file using the facilities provided by *Browser*.

Note: The main limitation is on speed, especially if the HDF5 file is big or contains a lot of nested groups. In that case, if you do not need to explore the file and you know exactly which results will be used, use *Picker* instead.

Data models of Apollo3 HDF5 files

The Apollo3 HDF5 output files mainly follow the standard data model, initially designed for reaction rates, which is structured as follows:

```
┌─ info ─┬─ (COMMENT)
│        ├─ NOUT
│        ├─ output_id1 ─┬─ GEOMID
│        │              ├─ NG
│        │              └─ COMMENT
│        ├─ output_id2 ⋯
│        ┆
│        ┆
│        └─ output_idNOUT ⋯
│
├─ geometry ─┬─ NGEO
│            ├─ geometry_id1 ─┬─ NZONE
│            │                ├─ VOLUME[NZONE]
│            │                └─ ZONENAME[NZONE]
│            ├─ geometry_id2 ⋯
│            ┆
│            ┆
│            └─ geometry_idNGEO ⋯
│
├─ output_id1 ─┬─ totaloutput ─┬─ KEFF (float)
│              │               ├─ (KINF (float))
│              │               ├─ (ABSORPTION[NG])
│              │               ├─ (CURRENT[NG])
│              │               ├─ (FLUX[NG])
│              │               ├─ (MIGRATIONAREA (float))
│              │               ├─ (PRODUCTION[NG])
│              │               ├─ (NVAL)
│              │               ├─ (LOCALVALUE[NVAL])
│              │               ├─ (LOCALNAME[NVAL])
│              │               └─ (localvalue) ─┬─ LOCALNAME[NVAL]
│              │                                ├─ value_id1
│              │                                ├─ value_id2
│              │                                ┆
│              │                                ┆
```

(continues on next page)

(continued from previous page)

```
│               │                                   └─ value_idNVAL
│               ├─ zone_id_1 ─┬─ NISOT
│               │              ├─ ISOTOPE[NISOT]
│               │              ├─ CONCEN[NISOT]
│               │              ├─ FLUX[NG]
│               │              ├─ macro ─┬─ reaction_id[NG]
│               │              │         ├┄
│               │              │         ┆
│               │              │
│               │              ├─ isotope_id1 ─┬─ reaction_id[NG]
│               │              │                ├┄
│               │              │                ┆
│               │              │
│               │              ├─ isotope_id2 ┄
│               │              ┆
│               │              ┆
│               │              └─ isotope_idNISOT ┄
│               ├─ zone_id2 ┄
│               ┆
│               ┆
│               └─ zone_idNZONE ┄
├─ output_id2 ┄
┆
┆
└─ output_idNOUT ┄
```

Elements in brackets are optional. If `NISOT == 0`, the `ISOTOPE`, `CONCEN` and `isotope_id` groups will be absent. The `macro` group stores the macroscopic reaction rates (= reaction rates for a medium). The `zone_id` folder names appear as values in the ZONENAME dataset in the geometry group, and they are arbitrary strings (chosen by the user in the Apollo3 case). `VOLUME` stores the volumes (in cm^3) of the zones, if needed later for normalisation for example. Most physics quantities are given on `NG` energy groups but the group bounds are not stored in the file. Some quantities are given as a function of energy groups and anisotropy. The order of the anisotropy development is available in the `'info'` group, possibly in the `'isotope_id` one. Current and surfacic flux (under `'totaloutput'`) have a `surface id` component in addition to energy groups.

Some files can also contain custom values or user values, especially in some cases the standard structure does not appear at all (no `zone_id` or `totaloutput` folder) but directly local values:

```
┌─ info ─┬─ (COMMENT)
│        ├─ (FORMAT)
│        ├─ (VERSION)
│
└─ output ─┬─ (LOCALVALUE[NVAL])
           ├─ (LOCALNAME[NVAL])
           └─ (localvalue) ─┬─ LOCALNAME[NVAL]
                            ├─ value_id1
                            ┆
                            ┆
                            └─ value_idNVAL
```

Use of Reader

```
from valjean.eponine.apollo3.hdf5_reader import Reader
ap3r = Reader('MON_FICHIER.hdf')
ap3b = ap3r.to_browser()
```

See *browser* to get more details on how to use the *Browser* objects.

In the resulting *Browser*, the metadata are taken from the h5py.Group values, while the actual data are taken from the h5py.Dataset ones.

Data are directly stored as *Dataset*. Errors are set to numpy.nan by default but this can be changed and be set to 0 (for example) using the error_value argument to the constructor. Bins are set to the group indices, or to group indices and number of anisotropies when available. Two specific cases are added: * surfacic flux: bins are given in group indices and surface numbers * current: bins are given in group indices, surface numbers and direction

Some metadata keys in the browser are slightly different from the corresponding names of the HDF5 file:

- **in the global variables (Browser.globals):**
  - **under the 'info' key:**
    * 'geom_id' `` for the ``'GEOMID' metadata
    * 'ngroups' for 'NG'
  - under the 'geometry' key: a dictonary associating the 'geom_id' names to dictionaries of {'ZONENAME': 'VOLUME'}.
- **in the results (Browser.content):**
  - zone names: kept as they are given in the HDF5 file (also true for 'totaloutput')
  - reaction names, flux, keff are set in lower case instead of upper
  - 'CONCEN' is renamed in 'concentration' and appears as a standard result (like reaction rates)
  - isotopes names are kept as they are but spaces surrounding the name are stripped. The 'macro' key is kept unchanged for macroscopic results (integrated over isotopes)
  - local value names are left inchanged (under the 'result_name' key).

The most common keys in the *Browser* are ['result_name', 'output', 'zone', 'isotope'].

An inspect function is also available (no class:*Reader* object needed) to have a quick look at the content of the file.

```
from valjean.eponine.apollo3.hdf5_reader import inspect
inspect('MON_FICHIER.hdf')
```

valjean.eponine.apollo3.hdf5_reader.**hdf_to_browser**(**args*, ***kwargs*)

Build a *Browser* from an Apollo3 HDF5 output.

Parameters

- **args** – other arguments to be passed to *Reader* constructor

- **kargs** – keyword arguments to be passed to *Reader* constructor

Returns
Browser

class valjean.eponine.apollo3.hdf5_reader.**Reader**(*fname*, *error_value=nan*)

Read an HDF5 file from Apollo3.

__init__(*fname*, *error_value=nan*)

Initialize the *Reader* object and read the HDF5 file.

Parameters

- **fname** (*str*) – path to the Apollo3 HDF5 ouput
- **error_value** – value to give to the error (default: `numpy.nan`)

read_file(*hfile*)

Read the HDF5 file.

The information read depends on the type of file. In the case of a *standard* output file, we read `'info'`, `'geometry'` and all results. In a *user* output, only results are read.

Parameters
hfile (*str*) – path to the HDF5 file

Raises
ReaderException – if unexpected keys are found.

process_standard_values(*hfile*)

Process standard values (keff, flux, rates).

process_user_values(*hfile*)

Process user values stored in files (named local values in file).

to_browser()

Build a *Browser* from the results in the HDF5 file.

Return type
Browser

valjean.eponine.apollo3.hdf5_reader.**extract_geometry**(*geometry*)

Extract geometry in dict to go to *Browser* globals.

Parameters
geometry (*h5py.Group*) – geometry stored in HDF5

valjean.eponine.apollo3.hdf5_reader.**extract_info**(*info*)

Extract info in dict to go to *Browser* globals.

Parameters
info (*h5py.Group*) – info stored in HDF5

valjean.eponine.apollo3.hdf5_reader.**extract_standard_values**(*val*, *ngroups*, *error*)

Extract results from Apollo3 HDF5 file.

Parameters

- **out_id** (*str*) – name of the output folder in HDF5
- **val** (*h5py.Group*) – results stored in the folder

- **error** (*int*, *float*) – value to give to the error (default: `numpy.nan`)

Returns
list of all results stored with metadata and data shaped as *Dataset*

Return type
list(dict)

`valjean.eponine.apollo3.hdf5_reader.`**`extract_user_values`**(*val*, *error*)

Extract user results (or 'local' values).

Parameters

- **val** (*h5py.Group*) – results stored under `'LOCALVALUES'`
- **error** (*int*, *float*) – value to give to the error (default: `numpy.nan`)

Returns
list of results including metadata

Return type
list(dict)

`valjean.eponine.apollo3.hdf5_reader.`**`extract_localvalues`**(*lnames*, *lvals*, *error*)

Extract local values from a list of local names.

Parameters

- **lnames** (*list(str)*) – list of the names of the local values
- **lvals** (*h5py.Dataset*) – local (scalar) values
- **error** (*int*, *float*) – value to give to the error (default: `numpy.nan`)

Returns
list of results including metadata

Return type
list(dict)

`valjean.eponine.apollo3.hdf5_reader.`**`loop_over_std_values`**(*lres*, *ngroups*, *error*)

Loop over results stored in the `'output_*'` groups of the HDF5 file.

Also retrieve number of anisotropies from the *info* `h5py.Group`.

Parameters

- **lres** (*h5py.Group*) – list of results
- **ngroups** (*int*) – number of energy groups
- **error** (*int*, *float*) – value to give to the error (default: `numpy.nan`)

Returns
list of results including metadata

Return type
list(dict)

`valjean.eponine.apollo3.hdf5_reader.`**`hdfdataset_to_dataset`**(*hdf_data*, *what*, *error*, *bins=None*)

Build *Dataset* from HDF5 dataset (in *rates* cases).

Returns a scalar or an array depending on data shape.

Parameters

- **hdf_data** (*h5py.Dataset*) – dataset containing the data
- **what** (*str*) – quantity name
- **error** (*int, float*) – value to give to the error (default: `numpy.nan`)
- **bins** (*collections.OrderedDict*) – bins corresponding to data, default *None*

Return type
Dataset

`valjean.eponine.apollo3.hdf5_reader.`**`build_dataset`**(*data*, *error*, *what*, *bins=None*)

Build *Dataset* from HDF5 result.

Parameters

- **data** (*numpy.generic, numpy.ndarray*) – data
- **what** (*str*) – quantity name
- **error** (numpy.generic, float, int, default = `numpy.nan`) – value of error
- **what** – quantity name
- **bins** (*collections.OrderedDict*) – bins corresponding to data, default *None*

Return type
Dataset

`valjean.eponine.apollo3.hdf5_reader.`**`make_bins`**(*nres*, *vres*, *ngroups*, *anisotropies=None*, *nsurfaces=None*)

Build bins `collections.OrderedDict`.

The bins currently available are * number of energy groups * number of anisotropies (for isotopes reaction rates) * number of surfaces (for current and surface flux)

Parameters
data (*h5py.Group*) – *info* block from *output_**

Return type
dict

Returns
dictionary of all possible number of bins

`valjean.eponine.apollo3.hdf5_reader.`**`extract_output_info`**(*data*, *name='anisotropy'*)

Extract info from output group containing number of anisotripies.

Parameters

- **data** (*h5py.Group*) – *info* block from *output_**
- **name** (*str*) – anisotropy name (‘*anisotropy*’ per default, result name if one

Return type
dict

Returns
dictionary indexed by anisotropy name

valjean.eponine.apollo3.hdf5_reader.**extract_surfaces_number**(*data*)

Extract number of surfaces if available.

valjean.eponine.apollo3.hdf5_reader.**extract_concentrations**(*zone*, *error*)

Store isotopes list and concentration from the zone group.

Parameters

- **zone** (*h5py.Group*) – HDF5 group corresponding to the zone
- **error** (*int*, *float*) – value to give to the error (default: `numpy.nan`)

Returns

list of dictionaries with isotope name in metadata, `'concentration'` as `result_name` and concentration values stored in a *Dataset* under the `'results'` key

Return type

list(dict)

valjean.eponine.apollo3.hdf5_reader.**extract_zone_values**(*val*, *ngroups*, *error*)

Store results by zone.

Results are stored depending on their nature (flux, concentrations, etc.).

Parameters

- **val** (*h5py.Group*) – HDF5 group containing results from a given zone
- **error** (*int*, *float*) – value to give to the error (default: `numpy.nan`)

Returns

list of results including metadata

Return type

list(dict)

valjean.eponine.apollo3.hdf5_reader.**dict_to_list**(*key_name*, *tdict*)

Transform a dictionary in a list of dictionaries including the key as a new key, value member of each dictionary.

Parameters

- **key_name** (*str*) – key of the new dictionary
- **tdict** (*dict*) – dictionary whose keys will become values of key_name

Returns

list of results including metadata

Return type

list(dict)

```
>>> tdict = OrderedDict()
>>> tdict['Graham'] = [{'day': 'Monday', 'meal': 'spam'},
...                    {'day': 'Tuesday', 'meal': 'egg'}]
>>> tdict['Terry'] =  {'day': 'Monday', 'meal': 'bacon'}
>>> tlist = dict_to_list('consumer', tdict)
>>> len(tlist)
3
>>> from pprint import pprint
>>> pprint(tlist)
[{'consumer': 'Graham', 'day': 'Monday', 'meal': 'spam'},
```

(continues on next page)

(continued from previous page)

```
 {'consumer': 'Graham', 'day': 'Tuesday', 'meal': 'egg'},
 {'consumer': 'Terry', 'day': 'Monday', 'meal': 'bacon'}]
```

Todo: Simplify this example by changing back OrderedDict in dict when python 3.5 won't be anymore supported.

valjean.eponine.apollo3.hdf5_reader.**inspect**(*hfile*, *name_spaces=50*)

Loop recursively over content of the file and print it.

Parameters

- **hfile** (*str*) – path to the Apollo3 HDF5 ouput
- **name_spaces** (*int*) – number of spaces reserved for the folder's names, default = 50

exception valjean.eponine.apollo3.hdf5_reader.**ReaderException**

An error that may be raised by the *Reader* class.

hdf5_picker – Pick results in HDF5 from Apollo3

This module is designed to pick the required results from an Apollo3 standard HDF5 output file. This means that the users should already know which result they want to extract. Some navigation methods exist, but they are not optimised for speed of access.

One possibility is to first explore the output file interactively with a *Reader*, and then to write the efficient code using *Picker*.

Picking results with Picker

```
from valjean.eponine.apollo3.hdf5_picker import Picker
ap3p = Picker('MON_FICHIER.hdf')
rate_ds = ap3p.pick_standard_value(output='OUTPUT_FOLDER',
                                   zone='ZONE_FOLDER',
                                   result_name='REQUIRED_RESULT')
```

For example to get the k_{eff}, in `output_1` under `totaloutput`:

```
keff = ap3p.pick_standard_value(output='output_1', zone='totaloutput',
                                result_name='KEFF')
```

The result will be a *Dataset*. The error is not available and it is set to `numpy.nan` by default. It can be changed using the `error_value` to the *Picker* constructor.

In the case of a *rates* file, bins are set to groups index in most cases. When the quantity is also calculated on anisotropies (e.g. diffusion) the bins are `('anisotropies', 'groups')`. The surfacic flux `('SURFFLUX')` bins are `('groups', 'surfaces')` and the current `('CURRENT')` ones are `('groups', 'surfaces', 'direction')`.

Results per isotopes can be obtained with an additional keyword argument, for example for ^{10}B absorption in `output_0` from zone 5:

```
b10abs = ap3p.pick_standard_value(output='output_0', zone='5',
                                  result_name='Absorption', isotope='B10')
```

Macroscopic rates are obtained with `isotope='macro'`.

Local values can also be retrived using the *Picker.pick_user_value* method:

- if `'LOCALNAME'` and `'LOCALVALUE'` belong to zone names:

```
locv = ap3p.pick_user_value(output='output', zone=None,
                            result_name='LOCAL_VALUE_NAME')
```

- if `'LOCALNAME'` and `'LOCALVALUE'` belong to result names in a zone:

```
locv = ap3p.pick_user_value(output='output', zone='totaloutput',
                            result_name='LOCAL_VALUE_NAME')
```

- if `'localvalue'` belongs to zone names:

```
locv = ap3p.pick_user_value(output='output', zone='localvalue',
                            result_name='LOCAL_VALUE_NAME')
```

In both cases, it might be useful to check the zones and the local names first.

Quick exploration with `Picker`

Some helpers exist if needed:

- *Picker.outputs*: list of output folders
- *Picker.geometry*: list of geometries stored in a given output folder
- *Picker.nb_groups*: number of energy groups available in a given output folder
- *Picker.zones*: list of zones in a given output folder
- *Picker.isotopes*: list of isotopes in a given zone of a given output folder
- *Picker.results*: list of available results in a given zone of a given output folder, isotope can be precised if needed
- *Picker.nb_anisotropies*: number of anisotropies available for a given result for the considered isotope in its zone and output
- *Picker.local_names*: list of names of the local values stored in a given zone from a given output folder

class valjean.eponine.apollo3.hdf5_picker.**Picker**(*fname*, *error_value=nan*)

Pick selected values from an Apollo3 HDF5 file.

__init__(*fname*, *error_value=nan*)

Initialize the *Picker* object and read the HDF5 file.

Parameters

- **fname** (*str*) – path to the Apollo3 HDF5 ouput file
- **error_value** – value to give to the error (default: `numpy.nan`)

close()

Close HDF5 file.

geometry_from_geomid(*, *geometry*)

Return geometry as a dictionary of zone names and volumes.

Parameters
geometry (*str*) – geometry name

geometry(*, *output*)

Return geometry characteristics (zones and their volumes) from the output name.

Parameters
output (*str*) – output in which zones will be found

Return type
dict

nb_groups(*, *output*)

Return the number of groups for the *output*.

Parameters
output (*str*) – output in which zones will be found

Return type
int

outputs()

Return list of available outputs.

Return type
list(str)

zones(*, *output*)

Return list of available zones in *output*.

Parameters
output (*str*) – output in which zones will be found

Return type
list(str)

isotopes(*, *output*, *zone*)

Return list of available isotopes in *zone* from *output*.

Parameters

- **output** (*str*) – output folder considered
- **zone** (*str*) – zone from output folder

Return type
list(str)

results(*, *output*, *zone*, *isotope=None*)

Return available results for the given configuration.

The *'info'* group is not considered as a result (group containing the number of anisotropies), so it is omitted from of the results list.

Parameters

- **output** (*str*) – output folder considered

- **zone** (*str*) – zone from the output folder
- **isotope** (*str or None*) – isotope considered for results (‘*macro*’ is counted as an isotope, if no isotope is given global results are given like flux or keff)

Return type
list(str)

nb_anisotropies(*, *output*, *zone*, *isotope*, *result*)

Return number of anisotropies for the given configuration.

Parameters

- **output** (*str*) – output folder considered
- **zone** (*str*) – zone from the output folder
- **isotope** (*str or None*) – isotope considered for results (‘*macro*’ is counted as an isotope, if no isotope is given global results are given like flux or keff)
- **result** (*str*) – result considered

Return type
list(str)

pick_standard_value(*, *output*, *zone*, *result_name*, *isotope=None*)

Pick a result according to required parameters from a standard file (like keff, flux or rate).

local_names(*, *output*, *zone*)

Return the list of available local value names.

Note: The most common zones seem to be in this case: *‘totaloutput’* (especially in rates files), *None* or *‘localvalue’* in core files.

Parameters

- **output** (*str*) – output folder considered
- **zone** (*str or None*) – zone from the output folder

Return type
list(str)

pick_value_from_index(*, *output*, *result_index*, *zone*, *name*)

Pick a user value result from required parameters.

Parameters

- **output** (*str*) – output folder considered
- **result_index** (*int*) – index of the required result
- **zone** (*str*) – zone from the output folder

Return type
Dataset

pick_user_value(*, *output*, *result_name*, *zone*)

Pick a user value result from required parameters (under local value in the file).

Parameters

- **output** (*str*) – output folder considered
- **result_name** (*str*) – name of the required result
- **zone** (*str*) – zone from the output folder

Return type

Dataset

exception valjean.eponine.apollo3.hdf5_picker.**PickerException**

An error that may be raised by the *Picker* class.

5.5 gavroche — Gestion AVancée de Routines et Oracles en CHaînE (nom temporaire)

Fig. 5: Gavroche Thénardier, illustré par Émile Bayard (1862).

5.5.1 test – Domain-specific language for writing tests

Domain-specific language for writing numeric tests.

This module provides a few classes and functions to write numeric tests.

Let us import the relevant modules first:

```
>>> from collections import OrderedDict
>>> from valjean.eponine.dataset import Dataset
>>> import numpy as np
```

Now we create a toy data set:

```
>>> x = np.linspace(-5., 5., num=100)
>>> y = x**2
>>> error = np.zeros_like(y)
>>> bins = OrderedDict()
>>> bins['x'] = x
>>> parabola = Dataset(y, error, bins=bins)
```

We perturb the data by applying some small amount of noise:

```
>>> eps = 1e-8
>>> noise = np.random.uniform(-eps, eps, parabola.shape)
>>> y2 = y + noise
>>> parabola2 = Dataset(y2, error, bins=bins)
```

Now we can test if the new dataset is equal to the original one:

```
>>> from valjean.gavroche.test import TestEqual
>>> test_equality = TestEqual(parabola, parabola2, name="parabola",
...                           description="equality test")
>>> test_equality_res = test_equality.evaluate()
>>> print(bool(test_equality_res))
False
```

However, they are approximately equal:

```
>>> from valjean.gavroche.test import TestApproxEqual
>>> test_approx = TestApproxEqual(parabola, parabola2, name="parabola",
...                               description="approx equal test")
>>> test_approx_res = test_approx.evaluate()
>>> print(bool(test_approx_res))
True
```

exception valjean.gavroche.test.**CheckBinsException**

An error is raised when check bins fails.

valjean.gavroche.test.**same_arrays**(*arr1*, *arr2*)

Return *True* if *arr1* and *arr2* are equal.

Parameters

- **arr1** – the first array.
- **arr2** – the second array.

valjean.gavroche.test.**same_bins**(*bins1*, *bins2*)

Return *True* if all the coordinate arrays are compatible.

Parameters

- **bins1** – the first dictionary of coordinate arrays.
- **bins2** – the second dictionary of coordinate arrays.

valjean.gavroche.test.**same_bins_datasets**(**datasets*)

Return *True* if all datasets have the same coordinates.

Parameters

datasets (*Dataset*) – any number of datasets.

valjean.gavroche.test.**check_bins**(**datasets*)

Check if the datasets have compatible coordinates, raise if not.

Raises

ValueError – if the datasets do not have compatible coordinates.

class valjean.gavroche.test.**Test**(*, *name*, *description=''*, *labels=None*)

Generic class for comparing any kind of results.

Base class for tests.

__init__(*, *name*, *description=''*, *labels=None*)

Initialize the *Test* object with a name, a description of the test (may be long) and labels if needed.

The test is actually performed in the *evaluate* method, which is abstract in the base class and must be implemented by sub-classes.

Parameters

- **name** (*str*) – name of the test, this string will typically end up in the test report as a section name.
- **description** (*str*) – description of the test exepcted with context, this string will typically end up in the test report.
- **labels** (*dict*) – labels to be used for test classification in reports (for example category, input file name, type of result, …)

abstract evaluate()

Evaluate the test on the given inputs.

Must return a subclass of *TestResult*.

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

class valjean.gavroche.test.**TestDataset**(*dsref*, **datasets*, *name*, *description=''*, *labels=None*)

Generic class for comparing datasets.

__init__(*dsref*, **datasets*, *name*, *description=''*, *labels=None*)

Initialisation of *TestEqual*:

Parameters

- **name** (*str*) – name of the test (in analysis)

- **description** (*str*) – specific description of the test
- **labels** (*dict*) – labels to be used for test classification in reports (for example category, input file name, type of result, ...)
- **dsref** (Dataset) – reference dataset
- **datasets** (*list(*Dataset*)*) – list of datasets to be compared to reference dataset

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

abstract evaluate()

Evaluate the test on the given datasets.

Must return a subclass of *TestResult*.

class valjean.gavroche.test.**TestResult**(*test*)

Base class for test results.

This result should be filled by *Test* daughter classes.

__init__(*test*)

Initialisation of *TestResult*.

Parameters
test (*Test* used) – the used test

class valjean.gavroche.test.**TestResultFailed**(*test*, *msg*)

Class for failed TestResults when an exception was raised during the evaluation.

__init__(*test*, *msg*)

Initialisation of *TestResult*.

Parameters
test (*Test* used) – the used test

class valjean.gavroche.test.**TestResultEqual**(*test*, *equal*)

Result from *TestEqual*.

__init__(*test*, *equal*)

Initialisation of the result from *TestEqual*:

Parameters

- **test** (*TestEqual*) – the used test
- **equal** (list (numpy.bool_) if datasets are numpy.generic, list (numpy.ndarray) if datasets are numpy.ndarray with dtype == bool.) – result from the test

__bool__()

Return the result of the test: True or False or raises an exception when it is not suitable.

class valjean.gavroche.test.**TestEqual**(*dsref*, **datasets*, *name*, *description=''*, *labels=None*)

Test if the datasets values are equal. Errors are ignored.

evaluate()

Evaluation of *TestEqual*.

Returns
TestResultEqual

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

class valjean.gavroche.test.**TestResultApproxEqual**(*test*, *approx_equal*)

Result from *TestApproxEqual*.

__init__(*test*, *approx_equal*)

Initialisation of the result from *TestApproxEqual*:

Parameters

- **test** (*TestApproxEqual*) – the used test
- **approx_equal** (numpy.generic if datasets are numpy.generic, numpy.ndarray if datasets are numpy.ndarray. In both cases `dtype == bool`.) – result from the test

class valjean.gavroche.test.**TestApproxEqual**(*dsref*, **datasets*, *name*, *description=''*, *labels=None*, *rtol=1e-05*, *atol=1e-08*)

Test if the datasets values are equal within the given tolerances. Errors are ignored.

__init__(*dsref*, **datasets*, *name*, *description=''*, *labels=None*, *rtol=1e-05*, *atol=1e-08*)

Initialisation of *TestApproxEqual*:

Parameters

- **name** (*str*) – local name of the test
- **description** (*str*) – specific description of the test
- **labels** (*dict*) – labels to be used for test classification in reports (for example category, input file name, type of result, ...)
- **dsref** (Dataset) – reference dataset
- **datasets** (*list(*Dataset*)*) – list of datasets to be compared to reference dataset
- **rtol** (*float*) – relative tolerance, default = 10^{-5}
- **atol** (*float*) – absolute tolerance, default = 10^{-8}

To get more details on *rtol* and *atol* parameters, see numpy.isclose.

evaluate()

Evaluation of *TestApproxEqual*.

Returns
TestResultApproxEqual

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

5.5.2 `stat_tests` – Statistical tests for datasets comparisons

Introduction

The available statistical tests are:

- Student's t-test, to compare two datasets, in the *`student`* module;
- The χ^2 test, for comparing distributions (spectra or meshes), in the *`chi2`* module;
- The Bonferroni and Holm-Bonferroni tests, to solve the problem of multiple hypothesis tests, in the *`bonferroni`* module.

Modules

`student` — Student's t-test

In Student's t-test, we consider 2 different datasets that can have different means and different variances. The main problem is to test if the means are equal. One way to do that is to use Student's t-test, although technically this name should be reserved for the case where the variances are unknown but they are known to be equal (the generalized version to different variances is called Welch's test).

In Tripoli-4, like in most Monte Carlo codes, the estimated statistical uncertainty is the standard error, i.e. the standard deviation of the mean of the samples. It is typically assumed that the distribution of the mean is normal, as a consequence of the central limit theorem. The variance of the population is estimated as

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

and the variance of the mean is estimated as

$$\bar{s}^2 = \frac{s^2}{n}$$

The standard error $\bar{s}$ is the quantity that typically enters Student's t-test.

The Student test statistic is:

$$t_{\text{Student}} = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\bar{s}_1^2 + \bar{s}_2^2}}$$

Note that, with our *`Dataset`*, this is equivalent to

```
diff = dataset_1 - dataset_2
t(Student) = diff.value / diff.error
```

as value is difference of values and error is equal to the quadratic sum of the errors.

Test interpretation

The tested hypothesis in the Student's t-test, i.e. the null hypothesis is:

- equal means, so $m_1 = m_2$
- Gaussian errors (or error following a normal distribution)
- independent datasets.

The test interpretation is based on the comparison of the p-value with a given significance level, called α. The (two-sided) p-value is defined as the probability to observe results more extreme than those that were actually observed, assuming that the null hypothesis is true. Mathematically, $p = P(|X| \geq |t_{\text{Student}}||N)$, X being the Student test statistic for samples of size N; the variable N is also called the number of degrees of freedom. The significativity level α is given by the user, usually 1% or 5%. Student's test fails if $P(|X| \geq |t_{\text{Student}}|) \leq \alpha$. If this is the case the null hypothesis is rejected.

To evaluate the test we thus need an estimate of the number of degrees of freedom, which is typically approximated as $n_1 + n_2 - 2$, with n_1 and n_2 being the number of batches. For large values of n_i (i.e. for a large number of degrees of freedom), the distribution of the Student statistic approaches a standard normal distribution.

It is sometimes easier to evaluate if the test succeeds by directly comparing the value of t_{Student} with a threshold given by the significativity level α and the number of degrees of freedom.

For example, here are some thresholds for some values of α and of the number of degrees of freedom:

	α = 1 %	α = 5 %
N = 10	2.7638	1.8125
N = 1000	2.5807	1.9623
N = ∞	2.5758	1.9600

Results for N = ∞ come from the normal distribution.

Note: This test can also be used on histograms (spectrum, mesh). In this case, an independent test is made for each bin. The test for the whole histogram succeeds if *all* the individual tests succeed. A better option is to use a Bonferroni or a Holm-Bonferroni test (see the *bonferroni* module).

Code implementation

If the number of degrees of freedom is given, p-values are estimated using Student's t-distribution. If the number is not given or it is *None*, it is assumed that it is very large and the normal approximation is used.

The p-value is only calculated if the number of degrees of freedom is given; otherwise, the Student result is just compared to the threshold. To test the p-value instead of the value of the t-statistic, use *TestResultStudent.test_pvalue*.

All these methods use the **SciPy** package (see scipy). The Student distribution comes from `scipy.stats.t`, where the pdf is explicitely written.

The method `sf` of `scipy.stats.t` corresponds to $1 - cdf$, so this gives the one-sided p-value; we need to multiply it by 2.0 to get the two-sided p-value. The `ppf` (percent point) function returns the threshold corresponding to the required one-sided significativity, so we need to divide α by 2.0 in our case.

The methods where the test is really applied are static methods, they can be used outside of the *TestStudent* class. Examples are given in the class docstrings.

Examples

Let's get an example: one simulation gave 5.3 ± 0.2, the other one 5.25 ± 0.08. Are these numbers in agreement ?

```
>>> from valjean.eponine.dataset import Dataset
>>> from valjean.gavroche.stat_tests.student import TestStudent
>>> import numpy as np
>>> ds1 = Dataset(5.3, 0.2)
>>> ds2 = Dataset(5.25, 0.08)
>>> tstudent = TestStudent(ds1, ds2, name="comp",
...                        description="Comparison using Student's t-test")
>>> tstudent_res = tstudent.evaluate()
>>> print(f'{tstudent_res.tstud[0]:.7f}')
0.2321192
>>> print(f'{tstudent_res.test.threshold:.7f}')
2.5758293
>>> bool(tstudent_res)
True
```

To obtain the p-value, a number of degrees of freedom should be given:

```
>>> tstudent = TestStudent(ds1, ds2, ndf=1000, name="comp",
...                        description="Comparison using Student's t- test")
>>> tstudent_res = tstudent.evaluate()
>>> print(f'{tstudent_res.tstud[0]:.7f}')
0.2321192
>>> print(f'{tstudent_res.test.threshold:.7f}')
2.5807547
>>> bool(tstudent_res)
True
>>> print(f'{tstudent_res.pvalue[0]:.7f}')
0.8164929
>>> tstudent_res.test_pvalue()
[True]
```

This is also possible if we want to compare a bin to bin different spectra. Let's define 2 small datasets containing arrays:

```
>>> ds3 = Dataset(np.array([5.2, 5.3, 5.25, 5.4, 5.5]),
...               np.array([0.2, 0.25, 0.1, 0.2, 0.3]))
>>> ds4 = Dataset(np.array([5.1, 5.6, 5.2, 5.3, 5.2]),
...               np.array([0.1, 0.3, 0.05, 0.4, 0.3]))
>>> tstudent = TestStudent(ds3, ds4, name="comp",
...                        description="Comparison using Student's t-test")
>>> tstudent_res = tstudent.evaluate()
>>> print(np.array2string(tstudent_res.oracles()))
[[ True  True  True  True  True]]
```

(continues on next page)

(continued from previous page)

```
>>> bool(tstudent_res)
True
```

In the case of multiple comparisons, the test will return *True* if *all* the individual comparisons succeed. This is the case here.

It is possible to get a detailed bin-by-bin comparisons, including bad bins:

```
>>> ds5 = Dataset(np.array([5.1, 5.9, 5.8, 5.3, 4.5]),
...               np.array([0.1, 0.1, 0.05, 0.4, 0.1]))
>>> tstudent = TestStudent(ds3, ds5, name="comp",
...                        description="Comparison using Student's t-test")
>>> tstudent_res = tstudent.evaluate()
>>> print(f'{tstudent_res.test.threshold:.7f}')
2.5758293
>>> print(np.array2string(tstudent_res.oracles()))
[[ True  True False  True False]]
```

Since some of the comparisons failed here, the whole tests fails as well:

```
>>> bool(tstudent_res)
False
```

As in the scalar case, it is possible to ask for the p-values. In both cases the significativity can be changed via the `alpha` argument, as shown here:

```
>>> tstudent = TestStudent(ds3, ds5, name="comp", alpha=0.05, ndf=1000,
...                        description="Comparison using Student's t-test")
>>> tstudent_res = tstudent.evaluate()
>>> print(f'{tstudent_res.test.threshold:.7f}')
1.9623391
>>> print(np.array2string(tstudent_res.oracles()))
[[ True False False  True False]]
>>> print(f'{tstudent_res.tstud[0][1]:.7f}')
-2.2283441
```

This last value is in-between the thresholds at 1% and 5%, in magnitude, and therefore the test succeeds in the former case and fails in the latter.

Obtaining p-values and comparing them is also possible in the spectrum case:

```
>>> print(np.array2string(tstudent_res.pvalue[0],
...                       formatter={'float_kind':'{:.7e}'.format}))
[6.5481769e-01 2.6079281e-02 1.0147751e-06 8.2310894e-01 1.6125524e-03]
>>> print(np.array2string(tstudent_res.test_pvalue()[0]))
[ True False False  True False]
```

Finally, for an array of size 1 (and dimension 1), the test will do the same as for datasets containing `numpy.generic`:

```
>>> ds6 = Dataset(np.array([5.3]), np.array([0.2]))
>>> ds7 = Dataset(np.array([5.25]), np.array([0.08]))
>>> tstudent = TestStudent(ds6, ds7, name="comp",
...                        description="Comparison using Student's t-test")
>>> tstudent_res = tstudent.evaluate()
>>> print(np.array2string(tstudent_res.tstud[0],
...                       formatter={'float_kind':'{:.7f}'.format}))
```

(continues on next page)

(continued from previous page)

```
[0.2321192]
>>> print(f'{tstudent_res.test.threshold:.7f}')
2.5758293
>>> bool(tstudent_res)
True
```

Like for usual operations on datasets, only tests between datasets of the same shape are possible:

```
>>> tstudent = TestStudent(ds1, ds7, name="comp",
...                        description="Comparison using Student's t-test")
>>> tstudent_res = tstudent.evaluate()
Traceback (most recent call last):
    ...
ValueError: Datasets to subtract do not have same shape
```

Multi-dimensional datasets are also allowed:

```
>>> ds8 = Dataset(np.array([[5.2, 5.3, 5.25], [5.4, 5.5, 5.2]]),
...               np.array([[0.2, 0.25, 0.1], [0.2, 0.3, 0.1]]))
>>> ds9 = Dataset(np.array([[5.1, 5.6, 5.2], [5.3, 5.2, 5.3]]),
...               np.array([[0.1, 0.3, 0.05], [0.4, 0.3, 0.2]]))
>>> ds8.shape
(2, 3)
>>> ds8.size
6
>>> tstudent = TestStudent(ds8, ds9, alpha=0.05, ndf=1000, name="comp",
...                        description="Comparison using Student's t-test")
>>> tstudent_res = tstudent.evaluate()
>>> bool(tstudent_res)
True
>>> print(np.array2string(tstudent_res.tstud[0],
...                       formatter={'float_kind':'{:.7f}'.format}))
[[0.4472136 -0.7682213 0.4472136]
 [0.2236068 0.7071068 -0.4472136]]
>>> print(np.array2string(tstudent_res.oracles()))
[[[ True  True  True]
  [ True  True  True]]]
```

Warning: If the errors are equal to 0, so the Student denominator is 0, AND the values are equal, so the numerator is also 0, the Student value is set to 0. In that case the test is assumed to succeed (boolean returning *True*).

```
>>> ds10 = Dataset(np.array([3.2, 0, 5]), np.array([0, 0.5, 0]))
>>> ds11 = Dataset(np.array([3.2, 0, 5.2]), np.array([0, 0.5, 0]))
>>> tstudent = TestStudent(ds10, ds11, name='zeros')
>>> tstudent_res = tstudent.evaluate()
>>> bool(tstudent_res)
False
>>> print(np.array2string(tstudent_res.tstud[0]))
[  0.   0. -inf]
>>> print(np.array2string(tstudent_res.oracles()))
[[ True  True False]]
```

class valjean.gavroche.stat_tests.student.**TestResultStudent**(*test*, *tstud*, *pvalue=None*)

Result from Student's t-test.

__init__(*test*, *tstud*, *pvalue=None*)

Initialisation of *TestResultStudent*

Members are lists whose length corresponds to the number of datasets compared to the reference dataset.

Parameters

- **test** (*TestStudent*) – the TestStudent object
- **tstud** (list (numpy.ndarray)) – Student's t-test results
- **pvalue** (list (numpy.generic) or list (list (numpy.generic))) – p-value or p-values depending on datasets, default is None

test_alpha(*tstud*)

Test p-value or first kind error.

Parameters
tstud (numpy.generic) – value of Student's t-statistic to be used for comparison

Returns
bool

oracles()

Final test (if spectrum)

Returns
list(bool)

__bool__()

Has this test succeeded?

The policy for *TestResultStudent* is that the test is considered to fail if any of the p-values are below the given significance level. This means that a test on a *Dataset* having multiple bins will fail if *any* of the bins exhibits large fluctuations. You might want to use *TestBonferroni* or *TestHolmBonferroni* to deal with such cases.

If this test concerns multiple datasets, the test is considered to succeed if all the dataset individually pass the test against the reference.

test_pvalue()

Result of the test by testing p-value.

Returns
bool or numpy.ndarray of bool if ndf was given to *TestStudent* and datasets are numpy.ndarray

class valjean.gavroche.stat_tests.student.**TestStudent**(*dsref*, **datasets*, *name*, *description=''*, *labels=None*, *alpha=0.01*, *ndf=None*)

Class to build the Student's t-test.

__init__(*dsref*, **datasets*, *name*, *description=''*, *labels=None*, *alpha=0.01*, *ndf=None*)

Initialisation of *TestStudent*

Parameters

- **name** (*str*) – local name of the test
- **description** (*str*) – specific description of the test
- **labels** (*dict*) – labels to be used for test classification in reports (for example category, input file name, type of result, ...)
- **dsref** (*Dataset*) – reference dataset
- **datasets** (list (*Dataset*)) – list of datasets to be compared to reference dataset
- **alpha** (*float*) – significance level, i.e. limit on the p-value (expected values are typically 0.01, 0.05), default is `0.01`
- **ndf** (*int*) – default is None, if given p-value will be calculated for ndf degrees of freedom (should correspond to number of batches), otherwise ndf assumed infinite, normal approximation

evaluate()

Evaluate Student's t-test method.

Returns
TestResultStudent

static student_test(*ds1*, *ds2*)

Compute Student's t-test or Student distribution for the given datasets (**static method**).

Parameters

- **ds1** (*Dataset*) – dataset 1
- **ds2** (*Dataset*) – dataset 2

Returns
numpy.generic or numpy.ndarray depending on *Dataset* type

static pvalue(*tstud*, *ndf*)

Calculation of the p-value (**static method**).

Parameters

- **tstud** (numpy.generic (float) or numpy.ndarray (float)) – Student's t-test statistic
- **ndf** (int or numpy.generic (int)) – number of degrees of freedom

Returns
numpy.generic (float) or numpy.ndarray (float)

static student_threshold(*alpha*, *ndf=None*)

Get threshold from probability (**static method**).

Parameters

- **alpha** (float or numpy.generic (float)) – required significance level (α)

- **ndf** (int or numpy.generic (int), default is None) – number of degrees of freedom

Returns
numpy.generic (float)

The threshold is two-sided.

If number of degrees of freedom is given, the distribution used is the Student distribution corresponding to ndf. If not, we consider an infinite number of degrees of freedom. In that case, the Student distribution tends to a normal distribution.

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

chi2 – χ^2 test

In our case we'll use the modified χ^2-test widely used in physics for comparing histograms. This test is also available in the **ROOT** software. For further explanations a good desciption is provided by the documentation of ROOT's Chi2Test() method on TH1 page or in this article by N. Gagunashvili.

The used formula is then:

$$\chi^2_{obs} = \sum_{i=1}^{p} \frac{(x_{1i} - x_{2i})^2}{\sigma_{1i}^2 + \sigma_{2i}^2}$$

where datasets (or distributions) 1 and 2 are compared, x_i being the values and σ_i being the error in each bin.

In our case only the 'weighted' case is implemented.

Statistical interpretation

The null hypothesis for the χ^2-test is:

- equal means in each bin
- gaussian means in each bin (so normal distribution in each bin)
- bins are independent

The interpretation of the test is based on the χ^2 value and on the p-value. Usually it is expected that the χ^2/ndf should be close to 1 (ndf = number of degrees of freedom), see PDG, statistics chapter. The p-value can also be calculated from the probability density function of the χ^2 distribution, which is given by:

$$pdf(x,k) = \frac{1}{2^{k/2}\Gamma(k/2)} x^{k/2-1} e^{-x/2}$$

for $x > 0$, k the number of degrees of freedom and Γ is Euler's Gamma function[1].

[1] The Γ function is given by

$$\Gamma(s) = \int_0^\infty t^{s-1} e^{-t} dt = (s-1)!$$

The cumulative distribution function, cdf, used to calculated the p-value is then:

$$cdf(x,k) = \frac{\gamma(k/2, x/2)}{\Gamma(k/2)} = \int_{-\infty}^{x} pdf(t,k)dt$$

where γ is the lower incomplete gamma function[2].

It is equivalent to calculate the probability $P(x < \chi^2_{obs})$ which is equal to $1 - cdf(x,k)$, so the integral from x to $+\infty$. This probability is the p-value.

If the p-value is lower than the define level (1%, 5% for example), the null hypothesis is rejected.

Possible biases can come from wrong estimation of uncertainties (underestimated for example or only statistical while a systematic one should be taken into account).

Code implementation

SciPy is used to calculate the p-value. The distribution `scipy.stats.chi2` is imported as *schi2`* in order to name conflicts. The p-value is obtained with the `sf` method. It is directly compared with the significance level required by the user.

> **Warning:** If some of your bins are not filled, if they are initialized to zero or if their error is zero, the χ^2 calculation will be wrong and probably give ∞.
>
> One option is proposed to remove these bins from the calculation, `ignore_empty`. The use of this option implies a reduction of the number of degrees of freedom by the number of removed bins (ndf = # bins). It has to be considered with caution, this does not make necessarly sense, depending on the user case.
>
> Another option is to apply some rebinning in order to have no empty bins (what usually means re-running to get consistent uncertainties).

Examples of the test

Let's consider a spectrum of 5 bins with their error and apply the χ^2-test.

```
>>> from valjean.eponine.dataset import Dataset
>>> from valjean.gavroche.stat_tests.chi2 import TestChi2
>>> import numpy as np
>>> ds1 = Dataset(np.array([5.2, 5.3, 5.25, 5.4, 5.5]),
...               np.array([0.2, 0.25, 0.1, 0.2, 0.3]))
>>> ds2 = Dataset(np.array([5.1, 5.6, 5.2, 5.3, 5.2]),
...               np.array([0.1, 0.3, 0.05, 0.4, 0.3]))
```

(continues on next page)

It also has an incomplete form (not starting at 0):

$$\Gamma(s,x) = \int_{x}^{\infty} t^{s-1}e^{-t}dt$$

[2] The γ function is given by:

$$\gamma(s,x) = \int_{0}^{x} t^{s-1}e^{-t}dt$$

(continued from previous page)

```
>>> tchi2 = TestChi2(ds1, ds2, alpha=0.05, name="comp",
...                  description="Comparison using Chi2 test")
>>> tchi2_res = tchi2.evaluate()
>>> print(f'{tchi2_res.chi2_per_ndf[0]:.7f}')
0.3080328
>>> bool(tchi2_res)
True
>>> print(f'{tchi2_res.pvalue[0]:.7f}')
0.9083889
```

Test with empty bins

One “empty bin” at the same position in the 2 compared datasets prevents the test to work. An empty bin is defined as a bin with zero error. In Monte Carlo codes a zero error can happen for example when only one batch has been run, no variance can be calculated in that case (`n-1` usually used with `n` the number of batches).

```
>>> ds3 = Dataset(np.array([5.2, 5.3, 5.25, 5.4, 5.5]),
...               np.array([0.2, 0.25, 0., 0.2, 0.3]))
>>> ds4 = Dataset(np.array([5.1, 5.6, 5.2, 5.3, 5.2]),
...               np.array([0.1, 0.3, 0., 0.4, 0.3]))
>>> tchi2 = TestChi2(ds3, ds4, alpha=0.05, name="comp",
...                  description="Comparison using Chi2 test")
>>> tchi2_res = tchi2.evaluate()
>>> tchi2_res.chi2
[inf]
>>> bool(tchi2_res)
False
>>> print(np.array2string(tchi2_res.test.nonzero_bins[0]))
[ True  True  True  True  True]
>>> tchi2_res.test.ndf[0]
5
>>> np.count_nonzero(ds4.error)
4
```

If the values are also at zero, we get a `nan` instead of an `inf`.

To get a χ^2 evaluation based on the non-empty bins, use the `ignore_empty` argument of the *`TestChi2`*. It recalculates the number of degrees of freedom removing the empty bins and only use the non-zero ones in the χ^2 calculation.

```
>>> tchi2 = TestChi2(ds3, ds4, alpha=0.05, ignore_empty=True,
...                  name="comp", description="Comparison using Chi2 test")
>>> tchi2_res = tchi2.evaluate()
>>> print(f'{tchi2_res.chi2[0]:.7f}')
1.3401639
>>> tchi2_res.test.ndf[0]
4
>>> tchi2_res.test.dsref.size
5
>>> print(np.array2string(tchi2_res.test.nonzero_bins[0]))
[ True  True False  True  True]
>>> print(f'{tchi2_res.chi2_per_ndf[0]:.7f}')
0.3350410
```

(continues on next page)

(continued from previous page)

```
>>> bool(tchi2_res)
True
```

Test with multiple dimensions datasets

```
>>> ds5 = Dataset(np.array([[5.2, 5.3, 5.25], [5.4, 5.5, 5.2]]),
...               np.array([[0.2, 0.25, 0.1], [0.2, 0.3, 0.1]]))
>>> ds6 = Dataset(np.array([[5.1, 5.6, 5.2], [5.3, 5.2, 5.3]]),
...               np.array([[0.1, 0.3, 0.05], [0.4, 0.3, 0.2]]))
>>> ds5.shape
(2, 3)
>>> ds5.size
6
>>> tchi2 = TestChi2(ds5, ds6, alpha=0.05, name="comp",
...                  description="Comparison using Chi2 test")
>>> tchi2_res = tchi2.evaluate()
>>> print(f'{tchi2_res.chi2_per_ndf[0]:.7f}')
0.2900273
>>> bool(tchi2_res)
True
>>> print(f'{tchi2_res.pvalue[0]:.7f}')
0.9419786
```

And when we have at least one empty bin, using the `ignore_empty` argument:

```
>>> ds7 = Dataset(np.array([[5.2, 5.3, 5.25], [5.4, 5.5, 5.2]]),
...               np.array([[0.2, 0.25, 0.], [0.2, 0.3, 0.1]]))
>>> ds8 = Dataset(np.array([[5.1, 5.6, 5.2], [5.3, 5.2, 5.4]]),
...               np.array([[0.1, 0.3, 0.], [0.4, 0.3, 0.]]))
>>> ds7.shape
(2, 3)
>>> ds7.size
6
>>> tchi2 = TestChi2(ds7, ds8, alpha=0.05, ignore_empty=True,
...                  name="comp", description="Comparison using Chi2 test")
>>> tchi2_res = tchi2.evaluate()
>>> print(f'{tchi2_res.chi2[0]:.7f}')
5.3401639
>>> tchi2_res.test.ndf[0]
5
>>> tchi2_res.test.dsref.size
6
>>> print(np.array2string(tchi2_res.test.nonzero_bins[0]))
[[ True  True False]
 [ True  True  True]]
>>> print(f'{tchi2_res.chi2_per_ndf[0]:.7f}')
1.0680328
>>> bool(tchi2_res)
True
```

Note: A RuntimeWarning is emitted if zero bins are used during calculation.

class valjean.gavroche.stat_tests.chi2.**TestResultChi2**(*test*, *chi2*, *pvalue*)

Result from the χ^2-test.

__init__(*test*, *chi2*, *pvalue*)

Construct a TestResultChi2 object.

Members are lists which length corresponds to the number of datasets compared to the reference dataset.

Parameters

- **test** (*TestChi2*) – the χ^2 test object
- **chi2** (list (numpy.generic (float))) – the value of the χ^2 statistic
- **pvalue** (list (numpy.generic (float))) – the test p-value

oracles()

Final test for the list of compared datasets.

Returns
numpy.ndarray

property chi2_per_ndf

Calculate the χ^2 per number of degrees of freedom.

Returns
χ^2/ndf

Return type
float

No parenthesis needed: this is a property.

class valjean.gavroche.stat_tests.chi2.**TestChi2**(*dsref*, **datasets*, *name*, *description=''*, *labels=None*, *alpha=0.01*, *ignore_empty=False*)

Test class for χ^2, inheritate from *Test*.

__init__(*dsref*, **datasets*, *name*, *description=''*, *labels=None*, *alpha=0.01*, *ignore_empty=False*)

Initialisation of *TestChi2*

Parameters

- **name** (*str*) – local name of the test
- **description** (*str*) – specific description of the test
- **labels** (*dict*) – labels to be used for test classification in reports (for example category, input file name, type of result, ...)
- **dsref** (*Dataset*) – reference dataset
- **datasets** (list (*Dataset*)) – list of datasets to be compared to reference dataset
- **alpha** (*float*) – probability to accept of not the test (p-value is expected greater), i.e. significance level
- **ignore_empty** (*bool*) – if the array contains zero bins, the test can done not consider them if this option is True; otherwise it will probably fail as the array to sum will contain infinite terms. Default is False.

nonzero_bins
nonzero bins identification by True, False if zero (list (numpy.ndarray (bool)))

ndf
number of degrees of freedom (list (int))

static pvalue(*chi2*, *ndf*)
Calculation of the p-value of the test.

Parameters
- **chi2** (numpy.generic (float)) – observed χ^2
- **ndf** (int or numpy.generic (int)) – number of degrees of freedom

Returns
numpy.generic (float)

static chi2_test(*ds1*, *ds2*, *nonzero_bins=None*)
Compute the χ^2 value for the given datasets.

Parameters
- **ds1** (*Dataset*) – first dataset
- **ds2** (*Dataset*) – second dataset
- **nonzero_bins** (numpy.ndarray (bool) or None (default)) – optional argument, booleans array of the same size as ds1 and ds2 to identify zero bins and possibly avoid them (see below)

Returns
χ^2 value

Return type
numpy.generic (float)

It has to be noted that if in a bin the sum of the variances of the test and the reference dataset is zero (so both datasets bin has a zero error) the bin cannot be taken into account in the calculation of the sum as the ratio will be infinite. This is why they can be removed via the nonzero_bins argument at that step. This has another consequence: the number of degrees of freedom is consequently reduced by the number of bins removed.

evaluate()
Evaluate χ^2 test method.

Return type
TestResultChi2

data()
Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

`bonferroni` – Multiple hypothesis tests

In some cases we would like to interpret not only one test (like on a generic score in Tripoli-4) but multiple ones (in a spectrum case for example). Each individual test is supposed to be successful but in most of the case we can accept that some of them fail (in a given acceptance of course).

The Bonferroni correction and the Holm-Bonferroni method allow to treat such cases. See for example family-wise error rate Wikipedia webpage for more details on this kind of tests.

Here are implemented two tests based on comparison of the p-value from the chosen test and a significance level:

- Bonferroni correction: all individual p-values are compared to the same significance level, depending on the number of tests;
- Holm-Bonferroni method: p-values are first sorted then compared to different significance levels depending on number of tests and rank.

In both cases a first test is chosen, like Student test, to evaluate the individual hypothesis. The p-value of the test is calculated and used in the following tests. Like in other tests, we consider two-sided significance levels here.

The null hypothesis is that the p-value is higher than a given significance level, that depends on the overall required significance level, the number of tests considered and the chosen method (Bonferroni or Holm-Bonferroni).

Bonferroni correction

The Bonferroni correction is used for multiple comparisons. It is based on a weighted comparison to the significance level α.

If we have m different tests (for example a spectrum with m bins), the significance level α will be weighted by m. This p-value of the m tests will then be compared to that value. As a consequence, the null hypothesis will be rejected if

$$p_k \leq \frac{\alpha}{m}$$

The individual significance level is usually largely smaller than the required one and the sum of them obvioulsy give α.

Conclusion of the test depends on what we want:

- it can be a simple `True` if all hypothesis are accepted;
- it can be `False` if at least one null hypothesis was rejected;
- we can also choose a level of acceptance, for example “it is fine if 1 % of the null hypothesis are rejected”

Per default, in our case the test is rejected if there is at least one null hypothesis rejected.

Holm-Bonferroni method

The Holm-Bonferroni method is a variation of the Bonferroni correction.

The p-value from the chosen test are first sorted from lower to higher value. Then each of them are compared to the significance level weighted by $m-k+1$ with m the number of tests and k the rank (starting at 1). The null hypothesis is rejected if

$$p_k \leq \frac{\alpha}{m-k+1}.$$

The test can be stopped at the first k accepting the null hypothesis.

This test is quite conservative. It can be possible to get the proportion of accepted and rejected hypothesis, like in the Bonferroni correction case.

Code implementation

In both cases, the test is initiliazed with another test, a Student test for example. This test can be evaluated inside the new test *evaluate* method or outside of it. Only examples with the full structure will be shown here.

TestResults are returned as `False` if one null hypothesis is rejected. Access to the initial test is still provided.

Examples

Let's do the usual imports then use the quick example from Student test, with the p-value calculation:

```
>>> import numpy as np
>>> from valjean.eponine.dataset import Dataset
>>> from valjean.gavroche.stat_tests.student import TestStudent
>>> from valjean.gavroche.stat_tests.bonferroni import TestBonferroni
>>> from valjean.gavroche.stat_tests.bonferroni import TestHolmBonferroni
```

Successful test with Bonferroni correction and Holm-Bonferroni method

```
>>> ds1 = Dataset(np.array([5.2, 5.3, 5.25, 5.4, 5.5]),
...               np.array([0.2, 0.25, 0.1, 0.2, 0.3]))
>>> ds2 = Dataset(np.array([5.1, 5.6, 5.2, 5.3, 5.2]),
...               np.array([0.1, 0.3, 0.05, 0.4, 0.3]))
>>> tstudent_12 = TestStudent(ds1, ds2, name="comp",
...                           description="Comparison using Student test",
...                           alpha=0.05, ndf=1000)
```

```
>>> tbonf = TestBonferroni(name="bonf", description="Bonferroni correction",
...                        test=tstudent_12, alpha=0.05)
>>> tbonf.ntests
5
>>> tbonf.ntests == ds1.size
True
```

(continues on next page)

(continued from previous page)

```
>>> tbonf.bonf_signi_level
0.005
>>> tbonf_res = tbonf.evaluate()
>>> bool(tbonf_res)
True
>>> tbonf_res.nb_rejected
[0]
```

```
>>> tholmbonf = TestHolmBonferroni(name="holm-bonf",
...                                description="Holm-Bonferroni method",
...                                test=tstudent_12, alpha=0.05)
>>> tholmbonf.alpha  # alpha is two-sided
0.025
>>> thb_res = tholmbonf.evaluate()
>>> bool(thb_res)
True
>>> thb_res.nb_rejected
[0]
```

Failing test with Bonferroni correction and Holm-Bonferroni method

```
>>> ds3 = Dataset(np.array([5.1, 5.9, 5.8, 5.3, 4.6]),
...               np.array([0.1, 0.1, 0.05, 0.4, 0.12]))
>>> tstudent_13 = TestStudent(ds1, ds3, name="comp",
...                           description="Comparison using Student test",
...                           alpha=0.05, ndf=1000)
```

```
>>> tbonf = TestBonferroni(name="bonf", description="Bonferroni correction",
...                        test=tstudent_13, alpha=0.05)
>>> tbonf.ntests
5
>>> tbonf.bonf_signi_level
0.005
>>> tbonf_res = tbonf.evaluate()
>>> bool(tbonf_res)
False
>>> tbonf_res.nb_rejected
[1]
>>> tbonf_res.rejected_proportion  # in percentage
[20.0]
```

```
>>> tholmbonf = TestHolmBonferroni(name="holm-bonf",
...                                description="Holm-Bonferroni method",
...                                test=tstudent_13, alpha=0.05)
>>> tholmbonf.alpha  # alpha is two-sided
0.025
>>> thb_res = tholmbonf.evaluate()
>>> bool(thb_res)
False
>>> thb_res.nb_rejected
[2]
>>> thb_res.rejected_proportion  # in percentage
[40.0]
```

For an easier comparison, p-values, associated significance levels and the corresponding result from the test (rejection or not) can be printed. The sorting order is also given for the Holm-Bonferroni method.

```
>>> it = np.nditer([thb_res.first_test_res.pvalue,
...                 tbonf.bonf_signi_level, tbonf_res.rejected_null_hyp,
...                 thb_res.sort_ordering, thb_res.alphas_i,
...                 thb_res.rejected_null_hyp],
...                flags=['multi_index'])
>>> while not it.finished:
...     if it.iterindex == 0:
...         print("index  | pvalue(B) |   α(B)    |  reject B | order(HB) |"
...               f"   α(HB)   | reject HB\n{'-':->77}")
...     print(f"{it.multi_index} | {it[0]:.3e} | {it[1]:.3e} | {it[2]!s:^9} | "
...           f"{it[3]:^7}   | {it[4]:.3e} |   {it[5]!s:<}")
...     _ = it.iternext()
index  | pvalue(B) |   α(B)    |  reject B | order(HB) |   α(HB)   | reject HB
-----------------------------------------------------------------------------
(0, 0) | 6.548e-01 | 5.000e-03 |   False   |    3      | 1.250e-02 |   False
(0, 1) | 2.608e-02 | 5.000e-03 |   False   |    2      | 8.333e-03 |   False
(0, 2) | 1.015e-06 | 5.000e-03 |   True    |    0      | 5.000e-03 |   True
(0, 3) | 8.231e-01 | 5.000e-03 |   False   |    4      | 2.500e-02 |   False
(0, 4) | 5.447e-03 | 5.000e-03 |   False   |    1      | 6.250e-03 |   True
```

The first index corresponds to the index of the compared dataset in the `datasets` container in the input test (compared to `dsref`).

The Holm-Bonferroni test is uniformly more powerful than the Bonferroni test. In this particular case, Holm-Bonferroni is able to reject an additional null hypothesis with respect to plain Bonferroni.

Tests with multi-dimension datasets

```
>>> ds4 = Dataset(np.array([[5.2, 5.3, 5.25], [5.4, 5.5, 5.2]]),
...               np.array([[0.2, 0.25, 0.1], [0.2, 0.3, 0.25]]))
>>> ds5 = Dataset(np.array([[5.1, 5.9, 5.8], [5.3, 4.7, 5.5]]),
...               np.array([[0.1, 0.1, 0.05], [0.4, 0.05, 0.2]]))
>>> ds4.shape
(2, 3)
>>> ds4.size
6
>>> tstudent_45 = TestStudent(ds4, ds5, name="comp",
...                           description="Comparison using Student test",
...                           alpha=0.05, ndf=1000)
```

```
>>> tbonf = TestBonferroni(name="bonf", description="Bonferroni correction",
...                        test=tstudent_45, alpha=0.05)
>>> tbonf.ntests
6
>>> print(f'{tbonf.bonf_signi_level:.6f}')
0.004167
>>> tbonf_res = tbonf.evaluate()
>>> bool(tbonf_res)
False
>>> tbonf_res.nb_rejected
```

(continues on next page)

(continued from previous page)

```
[1]
>>> print(f'{tbonf_res.rejected_proportion[0]:.3f} %')
16.667 %
```

```
>>> tholmbonf = TestHolmBonferroni(name="holm-bonf",
...                                description="Holm-Bonferroni method",
...                                test=tstudent_45, alpha=0.05)
>>> thb_res = tholmbonf.evaluate()
>>> tholmbonf.alpha
0.025
>>> bool(thb_res)
False
>>> thb_res.nb_rejected
[1]
>>> print(f'{thb_res.rejected_proportion[0]:.3f} %')
16.667 %
```

The detailed comparison also works for multi-dimensions arrays:

```
>>> it = np.nditer([thb_res.first_test_res.pvalue, tbonf.bonf_signi_level,
...                 tbonf_res.rejected_null_hyp, thb_res.sort_ordering,
...                 thb_res.alphas_i, thb_res.rejected_null_hyp],
...                flags=['multi_index'])
>>> while not it.finished:
...     if it.iterindex == 0:
...         print("index     | pvalue(B) |   α(B)   | reject B | order(HB) |"
...               f"   α(HB)  | reject HB\n{'-':->79}")
...     print(f"{it.multi_index} |  {it[0]:.2e} | {it[1]:.2e} | {it[2]!s:^7}  "
...           f"| {it[3]:^7}   | {it[4]:.2e} |   {it[5]!s:<}")
...     _ = it.iternext()
index     | pvalue(B) |   α(B)   | reject B | order(HB) |   α(HB)  | reject HB
--------------------------------------------------------------------------------
(0, 0, 0) |  6.55e-01 | 4.17e-03 |  False   |    4      | 1.25e-02 |   False
(0, 0, 1) |  2.61e-02 | 4.17e-03 |  False   |    2      | 6.25e-03 |   False
(0, 0, 2) |  1.01e-06 | 4.17e-03 |  True    |    0      | 4.17e-03 |   True
(0, 1, 0) |  8.23e-01 | 4.17e-03 |  False   |    5      | 2.50e-02 |   False
(0, 1, 1) |  8.66e-03 | 4.17e-03 |  False   |    1      | 5.00e-03 |   False
(0, 1, 2) |  3.49e-01 | 4.17e-03 |  False   |    3      | 8.33e-03 |   False
```

The first index corresponds to the index of the compared dataset in the `datasets` container in the input test (compared to `dsref`). The next ones correspond to the shape of the arrays.

The sorting order for Holm-Bonferroni considers all elements as equivalent (not done per dimension).

class valjean.gavroche.stat_tests.bonferroni.**TestResultBonferroni**(*test*, *first_test_res*, *rejected_null_hyp*)

Test result from Bonferroni correction.

The test returns `True` if the test is successful (null hypothesis was accepted) and `False` if it is rejected.

__init__(*test*, *first_test_res*, *rejected_null_hyp*)

Initialisation of *TestResultBonferroni*

Parameters

- **test** (*TestBonferroni*) – the used Bonferroni test
- **first_test_res** (*TestResult* child class) – the test result used to obtain the p-values
- **reject_hyp** (numpy.ndarray (bool)) – rejection of the null hypothesis for all bins

oracles()

Final test (on list of tests, espacially in case of mulitple tests on common reference).

Returns

list(bool)

property nb_rejected

Number of rejected null hypothesis according Holm-Bonferroni test.

Returns

list(int) or list (numpy.generic (int))

property rejected_proportion

Rejected proportion in percentage.

Returns

list (numpy.generic (float))

class valjean.gavroche.stat_tests.bonferroni.**TestBonferroni**(**, name, description='', labels=None, test, alpha=0.01*)

Bonferroni correction for multiple tests of the same hypothesis.

__init__(**, name, description='', labels=None, test, alpha=0.01*)

Initialisation of *TestBonferroni*.

Parameters

- **name** (*str*) – local name of the test
- **description** (*str*) – specific description of the test
- **labels** (*dict*) – labels to be used for test classification in reports (for example category, input file name, type of result, ...)
- **test** (*valjean.gavroche.test.Test* child class) – test from which pvalues will be extracted
- **alpha** (*float*) – required significance level

The significance level is considered two-sided here, so divide by 2. It is the significance level for the whole test.

property ntests

Returns the number of hypotheses to test, i.e. number of bins here.

static bonferroni_correction(*pvalues, bonf_signi_level*)

Bonferroni correction.

Parameters

- **pvalues** (numpy.ndarray (float)) – p-values from the original test

- **bonf_signi_level** (*float*) – significance level for each test (α weighted by the number of tests)

Returns
numpy.ndarray (bool)

Compares the p-value to the "Bonferroni significance level":

- if lower, null hypothesis is rejected for the bin
- if higher, null hypothesis is accepted for the bin.

evaluate()

Evaluate the Bonferroni correction.

Returns
TestResultBonferroni

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

class valjean.gavroche.stat_tests.bonferroni.**TestResultHolmBonferroni**(*test*, *first_test_res*, *alphas_i*, *rejected_null_hyp*)

Result from the Holm-Bonferroni method.

__init__(*test*, *first_test_res*, *alphas_i*, *rejected_null_hyp*)

Initialisation of *TestResultHolmBonferroni*

Parameters

- **test** (*TestHolmBonferroni*) – the used Holm-Bonferroni test
- **first_test_res** (*TestResult* child class) – the test result used to obtain the p-values
- **sorted_indices** (numpy.generic (int)) – indices of the p-values sorted to get p-values in increasing order
- **alphas_i** (numpy.ndarray (float)) – significance levels for sorted pvalues
- **rejected_hyp** (numpy.ndarray (bool)) – rejection of the null hypothesis for all bins

oracles()

Final test (on list of tests, espacially in case of mulitple tests on common reference).

Returns
list(bool)

property nb_rejected

Number of rejected null hypothesis according Holm-Bonferroni test.

Returns
list(int) or list (numpy.generic (int))

property rejected_proportion

Rejected proportion in percentage.

Returns

list (numpy.generic (float))

property sort_ordering

Get the sorted pvalues.

Returns

list (numpy.generic (int))

class valjean.gavroche.stat_tests.bonferroni.**TestHolmBonferroni**(*, *name*, *description=''*, *labels=None*, *test*, *alpha=0.01*)

Holm-Bonferroni method for multiple tests of the same hypothesis.

__init__(*, *name*, *description=''*, *labels=None*, *test*, *alpha=0.01*)

Initialisation of *TestHolmBonferroni*.

Parameters

- **name** (*str*) – local name of the test
- **description** (*str*) – specific description of the test
- **labels** (*dict*) – labels to be used for test classification in reports (for example category, jdd name, type of result, ...)
- **test** (*valjean.gavroche.test.Test* child class) – test from which pvalues will be extracted
- **alpha** (*float*) – significance level

The significance level is also considered two-sided here, so divide by 2. This is the overall significance level.

property ntests

Returns the number of hypotheses to test, i.e. number of bins here.

static holm_bonferroni_method(*pvalues*, *alpha*)

Holm-Bonferroni method.

Parameters

- **pvalues** (numpy.ndarray) – array of pvalues
- **alpha** (*float*) – significance level chosen for the Holm-Bonferroni method

Returns

sorted indices, array of the bins significance level, array of rejection of the null hypothesis

evaluate()

Evaluation of the used test and of the Holm-Bonferroni method.

Returns

TestResultHolmBonferroni

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

5.5.3 eval_test_task – One task to evaluate them all

The purpose of this module is to define the *EvalTestTask* class, a task that evaluates a collection of *Test* objects and transforms them into *TestResult* objects, which can be subsequently processed for inclusion in a test report.

valjean.gavroche.eval_test_task.**actually_eval_test**(*test*)

Actually perform test evaluation.

Returns
the result of test evaluation.

Return type
TestResult

Raises
TestResultFailed – if the underlying test raises any exception.

class valjean.gavroche.eval_test_task.**EvalTestTask**(*name*, *test_task_name*, ***, *deps=None*, *soft_deps=None*)

Class that evaluates a list of tests and stores the resulting *TestResult* objects in the environment.

classmethod from_test_task(*test_task*, *name=None*)

This method instantiates an *EvalTestTask* that will evaluate all the tests generated by a given task.

Parameters
test_task (Task) – a task that is expected to generate a list of tests as a result.

__init__(*name*, *test_task_name*, ***, *deps=None*, *soft_deps=None*)

Direct instantiation of an *EvalTestTask*.

Parameters

- **name** (*str*) – the name of this task.
- **test_task_name** (*str*) – the name of the task that generated the tests.
- **deps** (*list(*Task*) or None*) – the list of dependencies for this task.
- **soft_deps** (*list(*Task*) or None*) – the list of soft dependencies for this task.

valjean.gavroche.eval_test_task.**evaluate_tests**(*test_fn*, *name=None*)

Create an *EvalTestTask* objects for the given test.

Parameters

- **test_fn** (valjean.cosette.use.Use) – a function that produces tests, wrapped in a *Use* decorator.
- **name** (*str*) – the name of the *EvalTestTask* object.

Returns
the task that evaluates your tests.

Return type
EvalTestTask

5.5.4 diagnostics – Diagnostic tests

Test and task statistics

This module defines a few tests that report about the success/failure status of other tests/tasks.

Three different tests are available:

- statistics over all the tasks done, generated with *task_stats*
- statistics over all the tests performed with *test_stats*
- statistics over the tests performed based on labels given at test initialization with *test_stats_by_labels*

The two tests over the test results are performed using the *TestResult*, so the statistics only includes the tests that were actually run.

valjean.gavroche.diagnostics.stats.**stats_worker**(*test_fn*, *name*, *description*, *tasks*, ***kwargs*)

Function creating the test for all the required tasks (summary tests of test tasks or summary tests on all tasks for example).

Parameters

- **test_fn** – function generating the test to apply to tasks
- **name** (*str*) – the name of the stat task
- **description** (*str*) – its description
- **tasks** (*list(Task)*) – the list of tasks to be tested.

Returns
an *EvalTestTask* that evaluates the diagnostic test.

Return type
EvalTestTask

valjean.gavroche.diagnostics.stats.**task_stats**(*, *name*, *description=''*, *labels=None*, *tasks*)

Create a *TestStatsTasks* from a list of tasks.

The *TestStatsTasks* class must be instantiated with the list of task results, which are not available to the user when the tests are specified in the job file. Therefore, the creation of the *TestStatsTasks* must be delayed until the other tasks have finished and their results are available in the environment. For this purpose it is necessary to wrap the instantiation of *TestStatsTasks* in a *Use* wrapper, and evaluate the resulting test using a *EvalTestTask*.

This function hides this bit of complexity from the user. Assume you have a list of tasks that you would like to produce statistics about (we will use *DelayTask* objects for our example):

```
>>> from valjean.cosette.task import DelayTask
>>> my_tasks = [DelayTask(1), DelayTask(3), DelayTask(0.2)]
```

Here is how you make a *TestStatsTasks*:

```
>>> stats = task_stats(name='delays', tasks=my_tasks)
>>> from valjean.gavroche.eval_test_task import EvalTestTask
>>> isinstance(stats, EvalTestTask)
True
>>> print(stats.depends_on)
{Task('delays.stats')}
>>> create_stats = next(task for task in stats.depends_on)
```

Here *create_stats* is the task that actually creates the *TestStatsTasks*. It soft-depends on the tasks in *my_tasks*:

```
>>> [task in create_stats.soft_depends_on for task in my_tasks]
[True, True, True]
```

The reason why the dependency is soft is that we want to collect statistics about the task outcome in any case, even (especially!) if some of the tasks failed.

Parameters

- **name** (*str*) – the name of the task to create.
- **description** (*str*) – its description.
- **tasks** (*list(Task)*) – the list of tasks to be tested.

Returns
an *EvalTestTask* that evaluates the diagnostic test.

Return type
EvalTestTask

class valjean.gavroche.diagnostics.stats.**NameFingerprint**(*name*, *fingerprint=None*)

A small helper class to store a name and an optional fingerprint for the referenced item.

__init__(*name*, *fingerprint=None*)

__str__()
Return str(self).

__repr__()
Return repr(self).

__eq__(*other*)
Return self==value.

__lt__(*other*)
Return self<value.

__ge__(*other*, *NotImplemented=NotImplemented*)
Return a >= b. Computed by @total_ordering from (not a < b).

__gt__(*other*, *NotImplemented=NotImplemented*)
Return a > b. Computed by @total_ordering from (not a < b) and (a != b).

__hash__ = None

__le__(*other*, *NotImplemented=NotImplemented*)
Return a <= b. Computed by @total_ordering from (a < b) or (a == b).

class valjean.gavroche.diagnostics.stats.**TestStatsTasks**(*, *name*, *description=''*, *labels=None*, *task_results*)
A test that evaluates statistics about the success/failure status of the given tasks.

__init__(*, *name*, *description=''*, *labels=None*, *task_results*)
Instantiate a *TestStatsTasks*.

Parameters

- **name** (*str*) – the test name.
- **description** (*str*) – the test description.
- **task_results** (list(dict(str, *stuff*))) – a list of task results, intended as the contents of the environment sections associated with the executed tasks. This test notably inspects the `'status'` key to see if the task succeeded.

evaluate()
Evaluate this test and turn it into a *TestResultStatsTasks*.

data()
Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

class valjean.gavroche.diagnostics.stats.**TestResultStatsTasks**(*, *test*, *classify*)
The result of the evaluation of a *TestStatsTasks*. The test is considered successful if all the observed tasks have successfully completed (`TaskStatus.DONE`).

__init__(*, *test*, *classify*)
Instantiate a *TestResultStatsTasks*.

Parameters

- **test** (TestStatsTasks) – the test producing this result.
- **classify** (*dict(*TaskStatus, *list(str))*) – a dictionary mapping the task status to the list of task names with the given status.

__bool__()
Returns *True* if all the observed tests have succeeded.

class valjean.gavroche.diagnostics.stats.**TestOutcome**(*value*)
An enumeration that represents the possible outcomes of a test:

SUCCESS
represents tests that have been evaluated and have succeeded;

FAILURE
represents tests that have been evaluated and have failed;

MISSING
represents tasks that did not generate any `'result'` key;

NOT_A_TEST
represents tasks that did not generate a *TestResult* object as a result;

valjean.gavroche.diagnostics.stats.**test_stats**(*, *name*, *description=''*, *labels=None*, *tasks*)

Create a *TestStatsTests* from a list of tests.

The *TestStatsTests* class must be instantiated with the list of test results, which are not available to the user when the tests are specified in the job file. Therefore, the creation of the *TestStatsTests* must be delayed until the test tasks have finished and their results are available in the environment. For this purpose it is necessary to wrap the instantiation of *TestStatsTests* in a *Use* wrapper, and evaluate the resulting test using a *EvalTestTask*.

This function hides this bit of complexity from the user. Assume you have a list of tasks that evaluate some tests and that you would like to produce statistics about the tests results. Let us construct a toy dataset first:

```
>>> from collections import OrderedDict
>>> import numpy as np
>>> from valjean.eponine.dataset import Dataset
>>> x = np.linspace(-5., 5., num=100)
>>> y = x**2
>>> error = np.zeros_like(y)
>>> bins = OrderedDict([('x', x)])
>>> parabola = Dataset(y, error, bins=bins, name='parabola')
>>> parabola2 = Dataset(y*(1+1e-6), error, bins=bins, name='parabola2')
```

Now we write a function that generates dummy tests for the *parabola* dataset:

```
>>> from valjean.gavroche.test import TestEqual, TestApproxEqual
>>> def test_generator():
...     result = [TestEqual(parabola, parabola2, name='equal?').evaluate(),
...               TestApproxEqual(parabola, parabola2,
...                               name='approx_equal?').evaluate()]
...     return {'test_generator': {'result': result}}, TaskStatus.DONE
```

We need to wrap this function in a PythonTask so that it can be executed as a part of the dependency graph:

```
>>> from valjean.cosette.pythontask import PythonTask
>>> create_tests_task = PythonTask('test_generator', test_generator)
```

Here is how you make a *TestStatsTests* to collect statistics about the results of the generated tests:

```
>>> stats = test_stats(name='equal', tasks=[create_tests_task])
```

```
>>> from valjean.gavroche.eval_test_task import EvalTestTask
>>> isinstance(stats, EvalTestTask)
True
```

Here *stats* evaluates the test that gathers the statistics, and it depends on a special task that generates the *TestStatsTests* instance:

```
>>> print(stats.depends_on)
{Task('equal.stats')}
>>> create_stats = next(task for task in stats.depends_on)
```

In turn, *create_stats* has a soft dependency on the task that generates our test, *create_tests_task*:

```
>>> create_tests_task in create_stats.soft_depends_on
True
```

The reason why the dependency is soft is that we want to collect statistics about the test outcome in any case, even (especially!) if some of the tests failed or threw exceptions.

Let's run the tests:

```
>>> from valjean.config import Config
>>> config = Config()
>>> from valjean.cosette.env import Env
>>> env = Env()
>>> for task in [create_tests_task, create_stats, stats]:
...     env_up, status = task.do(env=env, config=config)
...     env.apply(env_up)
>>> print(status)
TaskStatus.DONE
```

The results are stored in a `list` under the key `'result'`:

```
>>> print(len(env[stats.name]['result']))
1
>>> stats_res = env[stats.name]['result'][0]
>>> print("SUCCESS:", stats_res.classify[TestOutcome.SUCCESS])
SUCCESS: ['approx_equal?']
>>> print("FAILURE:", stats_res.classify[TestOutcome.FAILURE])
FAILURE: ['equal?']
```

Parameters

- **name** (*str*) – the name of the task to create.
- **description** (*str*) – its description.
- **tasks** (*list(Task)*) – the list of tasks that generate the tests to observe.

Returns
an *EvalTestTask* that evaluates the diagnostic test.

Return type
EvalTestTask

class valjean.gavroche.diagnostics.stats.**TestStatsTests**(*, *name*, *description*='', *labels=None*, *task_results*)

A test that evaluates statistics about the success/failure of the given tests.

__init__(*, *name*, *description*='', *labels=None*, *task_results*)

Instantiate a *TestStatsTests* from a collection of task results. The tasks are expected to generate *TestResult* objects, which must appear in the `'result'` key of the task result.

evaluate()

Evaluate this test and turn it into a *TestResultStatsTests*.

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

class valjean.gavroche.diagnostics.stats.**TestResultStatsTests**(*, *test*, *classify*)

The result of the evaluation of a *TestStatsTests*. The test is considered successful if all the observed tests have been successfully evaluated and have succeeded.

__bool__()

Returns *True* if all the observed tests have succeeded.

valjean.gavroche.diagnostics.stats.**test_stats_by_labels**(*, *name*, *description*='', *labels=None*, *tasks*, *by_labels*)

Create a *TestStatsTestsByLabels* from a list of tests.

See *test_stats* for the generalities about this function.

Compared to *test_stats* it takes one argument more: `'by_labels'` to classify then build statistics based on these labels. **The order of the labels matters**, as they are successively selected.

Let's define three menus:

```
>>> menu1 = {'food': 'egg + spam', 'drink': 'beer'}
>>> menu2 = {'food': 'egg + bacon', 'drink': 'beer'}
>>> menu3 = {'food': 'lobster thermidor', 'drink': 'brandy'}
```

These three menus are ordered by pairs. Statistics on meals are kept in the restaurant, using *TestMetadata*. The goal of the tests is to know if both persons of a pair order the same menu and when they do it.

```
orders = [TestMetadata(
    {'Graham': menu1, 'Terry': menu1}, name='gt_wday_lunch',
    labels={'day': 'Wednesday', 'meal': 'lunch'}),
          TestMetadata(
    {'Michael': menu1, 'Eric': menu2}, name='me_wday_dinner',
    labels={'day': 'Wednesday', 'meal': 'dinner'}),
          TestMetadata(
    {'John': menu2, 'Terry': menu2}, name='jt_wday',
    labels={'day': 'Wednesday'}),
          TestMetadata(
    {'Terry': menu3, 'John': menu3}, name='Xmasday',
    labels={'day': "Christmas Eve"})]
```

The restaurant owner uses *test_stats_by_labels* to build statistics on his menus and the habits of his consumers.

For example, the menus filtered on `day` will give:

```
=============  ============  =============
      day       % success      % failure
=============  ============  =============
Christmas Eve      1/1            0/1
    Wednesday      2/3            1/3
=============  ============  =============
```

These results means, considering the tests requested, both consumers have the same meal on Christmas Eve. On Wednesday, one pair of customers out of three did not order the same menu.

The same kind of statistics can be done based on the meal:

```
==========  ============  =============
   meal       % success     % failure
==========  ============  =============
  dinner         0/1            1/1
   lunch         1/1            0/1
==========  ============  =============
```

In that case two tests were not taken into account as they did not have any `'meal'` label.

It is also possible to make selections on multiple labels. In that case the order matters: the classification is performed following the order of the labels requested. For example, `'meal'` then `'day'` :

```
==========  =========  ============  =============
   meal        day       % success     % failure
==========  =========  ============  =============
  dinner    Wednesday       0/1            1/1
   lunch    Wednesday       1/1            0/1
==========  =========  ============  =============
```

- Only two tests are filtered due to the meal selection
- Requesting `'day'` then `'meal'` would only inverse the two first columns in that case and emit a **warning**: a preselection on `'day'` is done and in Christmas Eve case the `'meal'` label is not provided, the selection cannot be done. In the Wednesday one, no problem `'meal'` appears at least in one case (two in our cases).

Finally if the request involves a label that does not exist in any test an exception will be raised, mentioning the failing label.

Parameters

- **name** (*str*) – the name of the task to create.
- **description** (*str*) – its description.
- **tasks** (*list(Task)*) – the list of tasks that generate the tests to observe.
- **by_labels** (*tuple(str)*) – labels from the tests on which the classification will be based

Returns

an *EvalTestTask* that evaluates the diagnostic test.

Return type

EvalTestTask

exception valjean.gavroche.diagnostics.stats.**TestStatsTestsByLabelsException**

Exception raised during the diagnostic test on *TestResult* when a classification by labels is required.

class valjean.gavroche.diagnostics.stats.**TestStatsTestsByLabels**(*, *name*, *description=''*, *labels=None*, *task_results*, *by_labels*)

A test that evaluates statistics about the success/failure of the given tests using their labels to classify them.

Usually more than one test is performed for each tested case. This test summarize tests done on a given category defined by the user in the usual tests (*TestStudent*, *TestMetadata*, etc.).

During the evaluation a list of dictionaries of labels is built for each test. These labels are given by the user at the initialization of the test. Each dictionary also contains the name of the test (name of the task) and its result (sucess or failure). From this list of dictionaries an *Index* is built.

The result of the evaluation is given a a list of dictionaries containing the strings corresponding to the chosen labels under the `'labels'` key and the number of results OK, KO and total.

__init__(*, *name*, *description=''*, *labels=None*, *task_results*, *by_labels*)

Instantiate a *TestStatsTestsByLabels* from a collection of task results. The tasks are expected to generate *TestResult* objects, which must appear in the `'result'` key of the task result.

Parameters

- **name** (*str*) – the test name
- **description** (*str*) – the test description
- **task_result** – a list of task results, each task normally contains a *TestResult*, used in this test.
- **by_labels** (*tuple*) – ordered labels to sort the test results. These labels are the test labels.

evaluate()

Evaluate this test and turn it into a *TestResultStatsTestsByLabels*.

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

class valjean.gavroche.diagnostics.stats.**TestResultStatsTestsByLabels**(*, *test*, *classify*, *n_labels*)

The result of the evaluation of a *TestStatsTestsByLabels*. The test is considered successful if all the observed tests have been successfully evaluated and have succeeded.

An oracle is available for each individual test (usually what is required here).

`self.classify` is here a list of dictionaries with the following keys: `['labels', 'OK', 'KO', 'total']`.

__init__(*, *test*, *classify*, *n_labels*)

Instantiate a *TestResultStatsTasks*.

Parameters

- **test** (TestStatsTasks) – the test producing this result.
- **classify** (*dict(TaskStatus, list(str))*) – a dictionary mapping the task status to the list of task names with the given status.

__bool__()
Test is successful if all tests are.

oracles()
Test if each test is successful.

Return type
list(bool)

nb_missing_labels()
Return the number of tests where at least one of the labels required were missing.

Return type
int

valjean.gavroche.diagnostics.stats.**classification_counts**(*classify*, *status_first*)
Count the occurrences of different statuses in the *classify* dictionary.

Parameters

- **classify** (*dict*) – a dictionary associating *things* to statuses. The statuses must have the same type as *status_first*
- **status_first** – the status that is considered as success. This must be an enum class

Returns
a pair of lists of equal length. The first element of the pair is the list of statuses appearing in *classify* (*status_first* is guaranteed to come first in this list); the second element is the number of times the corresponding status appears in *classify*.

Test of metadata

This module define tests for metadata.

class valjean.gavroche.diagnostics.metadata.**Missing**
Class for missing metadata.

__str__()
Return str(self).

class valjean.gavroche.diagnostics.metadata.**TestResultMetadata**(*test*, *dict_res*)
Results of metadata comparisons.

__init__(*test*, *dict_res*)
Initialisation of TestResultMetadata.

per_key()
Test result sorted by key.

only_failed_comparisons()
Return only the failed comparisons. Structure is the same as the dict_res.

class valjean.gavroche.diagnostics.metadata.**TestMetadata**(*dict_md*, *name*, *description=''*, *labels=None*, *exclude=('results', 'index', 'score_index', 'response_index', 'response_type')*)

A test that compares metadata.

Todo: Document the parameters...

__init__(*dict_md*, *name*, *description=''*, *labels=None*, *exclude=('results', 'index', 'score_index', 'response_index', 'response_type')*)

Initialisation of *TestMetadata*.

Parameters

- **name** (*str*) – local name of the test
- **description** (*str*) – specific description of the test
- **labels** (*dict*) – labels to be used for test classification in reports, for example category, input file name, type of result, ...
- **exclude** (*tuple*) – a tuple of keys that will not be considered as metadata. Default: `('results', 'index', 'score_index', 'response_index', 'response_type')`

build_metadata_dict()

Build the dictionary of metadata.

Contains all the metadata for all samples.

compare_metadata()

Metadata are compared with respect to the first one.

evaluate()

Evaluate this test and turn it into a *TestResultMetadata*.

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

5.6 javert — Journal Automatique de VÉRificaTion

5.6.1 templates – Things that can be put in the report

This module contains classes that are supposed to act as containers of all the information that is necessary to represent a test in a given format. For instance, in the case of tables this includes the column contents, the headers, etc. It does **not** include any formatting information, such as column widths, floating-point precision, colours, etc. Decisions about the formatting are handled by suitable formatting classes, such as *Rst*.

Fig. 6: L’inspecteur Javert, illustré par Émile Bayard (1862).

class valjean.javert.templates.**TableTemplate**(**columns*, *headers=None*, *units=None*, *highlights=None*)

A container class that encapsulates all the necessary information to represent a table.

Examples of use of mainly show in context of concatentation of *TableTemplate*, obtained with the *join* method.

```
>>> import numpy as np
>>> tit1 = TableTemplate(np.float_(1.5), np.float_(1.4),
...                      headers=['egg', 'spam'])
>>> tit2 = TableTemplate(np.float_(1.2), np.float_(0.9),
...                      headers=['egg', 'spam'])
>>> stab12 = join(tit1, tit2)
>>> print(len(tit1.columns), len(tit2.columns))
2 2
>>> print(tit1.columns[0].size, tit2.columns[0].size)
1 1
>>> print(len(stab12.columns))
2
>>> print(stab12.columns[0].size)
2
>>> print(f"{stab12!r}")
class: <class 'valjean.javert.templates.TableTemplate'>
headers: ['egg', 'spam']
egg: [1.5 1.2]
spam: [1.4 0.9]
highlights: [array([0., 0.]), array([0., 0.])]
```

`stab12` contained both `tit1` and `tit2` as expected. Headers of the columns are the same, length of the columns is the sum of the two.

```
>>> tit3 = TableTemplate(np.float_(0.8), np.float_(1.1),
...                      headers=['knight', 'parrot'])
>>> stab13 = join(tit1, tit3)
Traceback (most recent call last):
    ...
ValueError: TableTemplates to add should have same headers
```

An error is raised as the two *TableTemplate* don't contain the same headers, so not the same kind of columns, thus they cannot be concatenated.

It is also possible to join tables with same headers but different 'types' (scalars and arrays):

```
>>> tit4 = TableTemplate(np.arange(4), np.arange(4)*0.5,
...                      headers=['egg', 'spam'])
>>> print(len(tit4.columns), tit4.columns[0].size)
2 4
>>> stab14 = join(tit1, tit4)
>>> print(len(stab14.columns), stab14.columns[0].size)
2 5
>>> stab14.columns[0].size == tit1.columns[0].size + tit4.columns[0].size
True
>>> print(f"{stab14!r}")
class: <class 'valjean.javert.templates.TableTemplate'>
headers: ['egg', 'spam']
egg: [1.5 0.  1.  2.  3. ]
```

(continues on next page)

(continued from previous page)

```
spam: [1.4 0.  0.5 1.  1.5]
highlights: [array([0., 0., 0., 0., 0.]), array([0., 0., 0., 0., 0.])]
```

It is also possible to join arrays, a bigger array is obtained, without separation between the initial *TableTemplate*:

```
>>> tit5 = TableTemplate(np.arange(3)*0.1, np.arange(3)*0.05,
...                      headers=['egg', 'spam'])
>>> stab45 = join(tit4, tit5)
>>> print(len(stab45.columns), len(stab45.columns[0]))
2 7
>>> print(f"{stab45!r}")
class: <class 'valjean.javert.templates.TableTemplate'>
headers: ['egg', 'spam']
egg: [0.  1.  2.  3.  0.  0.1 0.2]
spam: [0.   0.5  1.   1.5  0.   0.05 0.1 ]
highlights: [array([0., 0., 0., 0., 0., 0., 0.]), array([0., 0., 0., 0., 0., 0.,␣
↪0.])]
```

Any number of *TableTemplate* can be joined (if fulfilling the requirements).

```
>>> stab145 = join(tit1, tit4, tit5)
>>> print(f"{stab145!r}")
class: <class 'valjean.javert.templates.TableTemplate'>
headers: ['egg', 'spam']
egg: [1.5 0.  1.  2.  3.  0.  0.1 0.2]
spam: [1.4  0.   0.5  1.   1.5  0.   0.05 0.1 ]
highlights: [array([0., 0., 0., 0., 0., 0., 0., 0.]), array([0., 0., 0., 0., 0.,␣
↪0., 0., 0.])]
```

The *TableTemplate.join* method updates the left *TableTemplate* as expected:

```
>>> tit1.join(tit4, tit5)
>>> print(f"{tit1!r}")
class: <class 'valjean.javert.templates.TableTemplate'>
headers: ['egg', 'spam']
egg: [1.5 0.  1.  2.  3.  0.  0.1 0.2]
spam: [1.4  0.   0.5  1.   1.5  0.   0.05 0.1 ]
highlights: [array([0., 0., 0., 0., 0., 0., 0., 0.]), array([0., 0., 0., 0., 0.,␣
↪0., 0., 0.])]
```

__init__(**columns*, *headers=None*, *units=None*, *highlights=None*)

Construct a table from a set of columns. The columns must be `numpy.ndarray` objects, and they must all contain the same number of elements (same array *size*).

Column headers may be specified using the *headers* argument; in this case, the number of headers must be equal to the number of columns.

Column units can also be specified using the *units* argument. Again, you must pass as many units as there are columns.

Finally, it is possible to specify which table elements should be highlighted. This is done by passing a list of lists (or `numpy.ndarray`) to the *highlights* argument. Each element of the *highlights* (outer) list represents a table column and therefore must have the same shape as all the other columns; also, the length of *highlights* must be equal to the number of columns. Elements of the inner lists (or `numpy.ndarray`)

must be booleans and indicate whether the corresponding table element must be highlighted.

Parameters

- **columns** (*list(numpy.ndarray)*) – a list of columns.
- **headers** (*list(str)*) – a list of headers.
- **units** (*list(str)*) – a list of measurement units.
- **highlights** (*list(list(bool)) or list(numpy.ndarray(bool))*) – a list describing which table elements should be highlighted.

copy()

Copy a *TableTemplate* object.

Return type
TableTemplate

Note: the highlignt function is not really copied, it has the same address as the self one. I don't know how to change that.

---

join(**others*)

Join a given number a *TableTemplate* to the current one.

Only *TableTemplate* with the same number of columns and same headers can be joined. The method returns the updated current one.

Parameters
others (*list(TableTemplate)*) – list of TableTemplates to be joined to the current TableTemplate

__repr__()

Print TableTemplate details.

__getitem__(*index*)

Build a sliced *TableTemplate* from the current *TableTemplate*.

Slicing is done like in the usual *NumPy* arrays, see: numpy indexing for more informations. No treatment like in *dataset* is done.

Parameters
index (*int, slice, tuple(slice)*) – index, slice or tuple of slices

Return type
TableTemplate

data()

Yield bytes representing *self*. Two *TableTemplate* objects containing equal data yield the same data.

__eq__(*other*)

Test for equality of *self* and another *TableTemplate*.

__ne__(*other*)

Test for inequality of *self* and another *TableTemplate*.

__hash__ = None

class valjean.javert.templates.**CurveElements**(*values*, *bins*, *legend*, *, *index=0*, *errors=None*)

Define the characteristics of a curve to plot.

__init__(*values*, *bins*, *legend*, *, *index=0*, *errors=None*)

Construction of *CurveElements*: curve details (values, bins, etc).

Values and errors (if given) should be numpy.ndarray of same shape (they must have only non-trivial dimension).

Bins are stored as a list of numpy.ndarray. This list should have the same length as the dimension of the values.

The index is used to share the plotting style between curves that should. For example, if on a plot there are the reference and two curves representing different data, let's say 'egg' and 'spam', if we also want to draw the ratio of these data with the reference, the same style will be applied to 'egg vs reference' and 'egg' and to 'spam vs reference' and 'spam'. In that case to ensure the same style 'egg vs reference' and 'egg' should have the same index (same for the 'spam' case).

Parameters

- **values** (*numpy.ndarray*) – array to be represented on the plot, **mandatory**
- **bins** (*list(numpy.ndarray)*) – bins to be used to represent the values
- **legend** (*str*) – string to be used in the legend to characterize the curve, **mandatory**
- **index** (*int*) – index of the curve (used for style for example)
- **errors** (*numpy.ndarray*) – errors associated to values (per default only on 1D plots and y-axis)

copy()

Copy a *CurveElements* object.

Return type
CurveElements

__repr__()

Printing of *CurveElements*.

__str__()

Printing of *CurveElements*.

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

__eq__(*other*)

Test for equality of *self* and another *CurveElements*.

__ne__(*other*)

Test for inequality of *self* and another *CurveElements*.

__hash__ = None

class valjean.javert.templates.**SubPlotAttributes**(*dim*)

Container to store sub-plots attributes:

- axis limits
- axis scale: linear (default) or logatithmic
- additional horizontal or vertical lines

Theses attributes are independent of the used backend (examples: **matplotlib**, **Root**, **gnuplot**, **D3**). The backend then gets the attributes and apply them with its own features.

__init__(*dim*)

Initialisation of PlotAttributes.

The attributes of the instance are private.

Parameters
dim (*int*) – dimension of the data on the sub-plot (used to check consistency of limits)

copy()

Copy a *SubPlotAttributes* object.

Return type
SubPlotAttributes

property limits

Return limits.

property lines

Return lines to be plotted.

exception valjean.javert.templates.**SubPlotElementsException**

Error raised if the sub plot looks inconsistent.

class valjean.javert.templates.**SubPlotElements**(**, curves, axnames=('', ''), ptype='1D'*)

Container to store a given sub-plot.

__init__(**, curves, axnames=('', ''), ptype='1D'*)

Initialisation of *SubPlotElements*.

A subplot is defined as data (`curves`) sharing the same plotting properties:

- axis names
- type of the plot (e.g. `'1D'`, `'2D'`, ...), see the chosen backend to get the list of possibilities (example: *MplPlot*)
- axis scales
- axis limits

The last axis name corresponds to the quantity to be drawn, the first ones to the bins.

Axis scales can be linear (default) or logarithmic (if set to `True`).

Vertical or horizontal lines can also be added.

Parameters

- **curves** (*list(CurveElements)*) – list of curves to go on the sub-plot
- **axnames** (*tuple*) – name of the axes of the sub-plot
- **type** (*str*) – type of the sub-plot, default: `'1D'`

copy()

Copy a *SubPlotElements* object.

Return type
SubPlotElements

__repr__()

Printing of *SubPlotElements*

__str__()

Printing of *SubPlotElements*

data()

Generator yielding objects supporting the buffer protocol that (as a whole) represent a serialized version of *self*.

__eq__(*other*)

Test of equality of *self* and another *SubPlotElements*.

__ne__(*other*)

Test for inequality of *self* and another *SubPlotElements*.

__hash__ = None

class valjean.javert.templates.**PlotTemplate**(*, *subplots*, *small_subplots=True*, *suppress_xaxes=False*, *suppress_legends=False*, *backend_kw=None*)

A container for full test result to be represented as a plot. This includes all the datasets and the test result. This can also include p-values or other test parameters, depending on what is given to the *PlotTemplate*.

Examples mainly present the *join* method, used to concatentate *PlotTemplate*.

```
>>> bins1, d11, d12 = np.arange(4), np.arange(4), np.arange(4)*10
>>> d13 = d11 + d12
>>> bins2, d2 = np.arange(5), np.arange(5)*0.5
>>> pit1 = PlotTemplate(subplots=[SubPlotElements(
...     curves=[CurveElements(d11, [bins1], 'd11', index=0)],
...     axnames=['egg', 'brandy'])])
>>> pit2 = PlotTemplate(subplots=[SubPlotElements(
...     curves=[CurveElements(d12, bins=[bins1], legend='d12', index=1)],
...     axnames=['egg', 'beer'])])
>>> pit3 = PlotTemplate(subplots=[SubPlotElements(
...     curves=[CurveElements(d13, legend='d13', bins=[bins1], index=2)],
...     axnames=['egg', 'wine'])])
>>> splt123 = join(pit1, pit2, pit3)
>>> print(f"{splt123!r}")
class:   <class 'valjean.javert.templates.PlotTemplate'>
N subplots: 3
Subplot 0
 axnames: ['egg', 'brandy'], plot type: 1D, N curves: 1
 Curve 0
```

(continues on next page)

(continued from previous page)

```
  legend:  d11
  index:   0
  bins:    [array([0, 1, 2, 3])]
  values:  [0 1 2 3]
  errors:  None
Subplot 1
 axnames: ['egg', 'beer'], plot type: 1D, N curves: 1
 Curve 0
  legend:  d12
  index:   1
  bins:    [array([0, 1, 2, 3])]
  values:  [ 0 10 20 30]
  errors:  None
Subplot 2
 axnames: ['egg', 'wine'], plot type: 1D, N curves: 1
 Curve 0
  legend:  d13
  index:   2
  bins:    [array([0, 1, 2, 3])]
  values:  [ 0 11 22 33]
  errors:  None
```

As expected a new *PlotTemplate* is obtained, containing three subplots, each one containing one curve.

Like in the TableTemplate case, the *PlotTemplate.join* method updates the left *PlotTemplate* as expected:

```
>>> pit1.join(pit2, pit3)
>>> pit1 == splt123
True
```

A new curve with the same axes will also create a new suplot:

```
>>> d14 = d11*2
>>> pit4 = PlotTemplate(subplots=[SubPlotElements(
...     curves=[CurveElements(d14, legend='d14', index=3, bins=[bins1])],
...     axnames=['egg', 'beer'])])
>>> split24 = join(pit2, pit4)
>>> print(split24)
class:   <class 'valjean.javert.templates.PlotTemplate'>
Subplot 0
 axnames: ['egg', 'beer'], plot type: 1D
 Curve 0
  legend:  d12
  index:   1
  bins:    [array([0, 1, 2, 3])]
Subplot 1
 axnames: ['egg', 'beer'], plot type: 1D
 Curve 0
  legend:  d14
  index:   3
  bins:    [array([0, 1, 2, 3])]
```

To get it in the same subplot it has to be done at creation.

```
>>> pit24 = PlotTemplate(subplots=[SubPlotElements(
...     curves=[CurveElements(d12, legend='d12', index=1, bins=[bins1]),
...             CurveElements(d14, legend='d14', index=3, bins=[bins1])],
...     axnames=['egg', 'beer'])])
>>> print(pit24)
class:   <class 'valjean.javert.templates.PlotTemplate'>
Subplot 0
 axnames: ['egg', 'beer'], plot type: 1D
 Curve 0
  legend:  d12
  index:   1
  bins:    [array([0, 1, 2, 3])]
 Curve 1
  legend:  d14
  index:   3
  bins:    [array([0, 1, 2, 3])]

>>> split24 == pit24
False
```

N-dimensional plot templates can be built, but the plotting engine may not be able to convert multi-dimensional templates into plots.

The same behavior is expected for multi-dimensions plots: `join` will had a new subplot. To be noted: only one curve can be plotted on a subplot in multi-dimensional case, so any additional curve will throw a warning. Plot representation may not be as expected.

```
>>> d31 = np.arange(bins1.size*bins2.size).reshape(bins1.size, bins2.size)
>>> d32 = np.arange(bins1.size*bins2.size).reshape(
...     bins1.size, bins2.size)*0.01
>>> pit7 = PlotTemplate(subplots=[SubPlotElements(
...     curves=[CurveElements(d31, bins=[bins1, bins2],
...                           legend='d31', index=0)],
...     axnames=['egg', 'spam', 'bacon'], ptype='2D')])
>>> pit8 = PlotTemplate(subplots=[SubPlotElements(
...     curves=[CurveElements(d32, bins=[bins1, bins2],
...                           legend='d32', index=1)],
...     axnames=['egg', 'spam', 'lobster'], ptype='2D')])
>>> splt78 = join(pit7, pit8)
>>> print(f"{splt78!s}")
class:   <class 'valjean.javert.templates.PlotTemplate'>
Subplot 0
 axnames: ['egg', 'spam', 'bacon'], plot type: 2D
 Curve 0
  legend:  d31
  index:   0
  bins:    [array([0, 1, 2, 3]), array([0, 1, 2, 3, 4])]
Subplot 1
 axnames: ['egg', 'spam', 'lobster'], plot type: 2D
 Curve 0
  legend:  d32
  index:   1
  bins:    [array([0, 1, 2, 3]), array([0, 1, 2, 3, 4])]
```

It is also possible to mix 1D and 2D plots:

```
>>> splt27 = join(pit2, pit7)
```

(continues on next page)

(continued from previous page)

```
>>> print(f"{splt27!s}")
class:   <class 'valjean.javert.templates.PlotTemplate'>
Subplot 0
 axnames: ['egg', 'beer'], plot type: 1D
 Curve 0
  legend:  d12
  index:   1
  bins:    [array([0, 1, 2, 3])]
Subplot 1
 axnames: ['egg', 'spam', 'bacon'], plot type: 2D
 Curve 0
  legend:  d31
  index:   0
  bins:    [array([0, 1, 2, 3]), array([0, 1, 2, 3, 4])]
```

__init__(*, *subplots*, *small_subplots=True*, *suppress_xaxes=False*, *suppress_legends=False*, *backend_kw=None*)

Construction of the PlotTemplate from a list of *SubPlotElements*.

Parameters

- **subplots** (*list(SubPlotElements)*) – list of sub-plots
- **small_subplots** (*bool*) – draw additional subplots in smaller size than the first one, default = `True`
- **suppress_xaxes** (*bool*) – suppress label and ticks labels of the x-axis of subplots except the last one, default = `False`
- **suppress_legends** (*bool*) – suppress legend on all subplots except the first one, default = `False`
- **backend_kw** (*dict*) – dictionary with backend-specific options

copy()

Copy a *PlotTemplate* object.

Return type
PlotTemplate

join(**others*)

Join a given number a *PlotTemplate* to the current one.

Only *PlotTemplate* with the same number of columns and same headers can be joined. The method returns the updated current one.

Parameters
others (*list(PlotTemplate)*) – list of PlotTemplates to be join with the current PlotTemplate

__repr__()

Printing of *PlotTemplate*.

__str__()

Printing of *PlotTemplate*.

data()

Yield bytes representing *self*. Two *TableTemplate* objects containing equal data yield the same data.

__eq__(*other*)
Test for equality of *self* and another *PlotTemplate*.

__ne__(*other*)
Test for inequality of *self* and another *PlotTemplate*.

curves_index()
Return a sorted list of unique index of the curves.

__hash__ = None

class valjean.javert.templates.**TextTemplate**(*text*)
A container class that encapsulates text for the report.

The user has to write the text as a string. ReST markdown can be used as compilation is expected to be done by sphinx.

Note: Titles might not be well represented in a ReSt formatted TextTemplate: no knowledge of the current level of title, nor the associated symbol. Lists or enumerations might be more suitable.

As in the other templates, examples will focus on the concatenation (join) of different TextTemplate.

```
>>> ttplt1 = TextTemplate('Spam egg bacon')
>>> print(f"{ttplt1!r}")
<class 'valjean.javert.templates.TextTemplate'>(text='Spam egg bacon')
>>> ttplt2 = TextTemplate('**Spam** egg bacon')
>>> print(f"{ttplt2!r}")
<class 'valjean.javert.templates.TextTemplate'>(text='**Spam** egg bacon')
>>> ttplt3 = TextTemplate(r".. role:: hl\n\nsausage :hl:`tomato`")
>>> ttplt1.join(ttplt3)
>>> print(f"{ttplt1!r}")
<class 'valjean.javert.templates.TextTemplate'>(text='Spam egg bacon.. role:: hl\\
↪n\\nsausage :hl:`tomato`')
```

The sphinx compilation will fail there, as the are no empty line between the first and the second string. If you know some text will follow, think about the \n.

```
>>> ttplt1 = TextTemplate('Spam egg bacon\n\n')
>>> ttplt3 = TextTemplate('.. role:: hl\n\nsausage :hl:`tomato`\n\n')
>>> ttplt1.join(ttplt3)
>>> print(f"{ttplt1!r}")
<class 'valjean.javert.templates.TextTemplate'>(text='Spam egg bacon\n\n.. role::␣
↪hl\n\nsausage :hl:`tomato`\n\n')
```

Test of the external function *join*:

```
>>> ttplt4 = join(ttplt3, ttplt3)
>>> print(f"{ttplt4!r}")
<class 'valjean.javert.templates.TextTemplate'>(text='.. role:: hl\n\nsausage :hl:
↪`tomato`\n\n.. role:: hl\n\nsausage :hl:`tomato`\n\n')
```

The copy doesn't affect the original:

```
>>> ttplt5 = ttplt2.copy()
>>> ttplt5.text += ' sausage'
>>> print(f"{ttplt2!r}")
<class 'valjean.javert.templates.TextTemplate'>(text='**Spam** egg bacon')
>>> print(f"{ttplt5!r}")
<class 'valjean.javert.templates.TextTemplate'>(text='**Spam** egg bacon sausage')
```

__init__(*text*)

Construct the text to be sent to the report.

Parameters
text (*str*) – text to be written in the report

__repr__()

Print *TextTemplate* details.

copy()

Copy a *TextTemplate* object.

Return type
TextTemplate

join(**others*)

Join a given number of *TextTemplate* to the current one.

data()

Yield bytes representing *self*. Two *TableTemplate* objects containing equal data yield the same data.

__eq__(*other*)

Test for equality of *self* and another *TextTemplate*.

__hash__ = None

__ne__(*other*)

Test for inequality of *self* and another *TextTemplate*.

valjean.javert.templates.**join**(**templates*)

Join a "list" of templates of same kind, *TableTemplate* or *PlotTemplate* using the related `join` methods.

It returns a new templates (*TableTemplate* or *PlotTemplate*).

Parameters
templates (*list(TableTemplate) or list(PlotTemplate)*) – list of templates

Return type
TableTemplate, *PlotTemplate*, *TextTemplate*

See *TableTemplate* and *PlotTemplate* for examples of use. Only few error cases will be shown here:

```
>>> bins1, data11, data12 = np.arange(4), np.arange(4), np.arange(4)*10
>>> bins2, data2 = np.arange(5), np.arange(5)*0.5
>>> tablit = TableTemplate(bins1, data11, headers=['egg', 'spam'])
>>> plotit = PlotTemplate(subplots=[SubPlotElements(
...     curves=[CurveElements(data11, bins=[bins1], legend='d11')],
...     axnames=['egg', 'spam'])])
```

(continues on next page)

(continued from previous page)

```
>>> tit = join(tablit, plotit)
Traceback (most recent call last):
    ...
TypeError: Only a TableTemplate can be joined to another TableTemplate
>>> tit = join(plotit, tablit)
Traceback (most recent call last):
    ...
TypeError: Only a PlotTemplate can be joined to another PlotTemplate
```

5.6.2 representation – Transform test results into items

This module contains code that converts a test result into some kind of human-readable representation (table, plots, etc.).

Todo: Possible improvement: turn *Representer* into an *ABC*; loop over the classes in *eponine* that inherit from *:class:`~.TestResult* and add *@abstractmethod* methods in *Representer*. This way, if a new *TestResult* is added to *eponine*, it will no longer be possible to instantiate any of the classes that derive from *Representer*, pointing to the fact that the code in this module needs to be extended to handle the new class. This is better than silently falling back to some default do-nothing implementation, which may lead to bugs.

This module uses the *Strategy* design pattern. The *Representation* class plays the role of *Context*, *Representer* plays the role of *Strategy* and the classes derived from *Representer* (such as *TableRepresenter*, *PlotRepresenter*, etc.) play the role of *ConcreteStrategy*. See E. Gamma et al., "Design Patterns" (1995), ISBN 0-201-63361-2, Addison-Weasley, USA.

Currently we have 3 main `Representer` classes:

- *Representer*: parent class of all others, containing the default *Representer.__call__* method, calling the class method named `'repr_' + class.name` of the test;
- *TableRepresenter*: inherited from *Representer*, designed as a parent class for user's own representations of tables. Its *TableRepresenter.__call__* method first looks for a method called `'repr_' + class.name`; if it does not exist call the default method from the catalogue of table representation accessible in *table_repr*;
- *PlotRepresenter*: inherited from *Representer*, designed as a parent class for user's own representations of plots. Its *PlotRepresenter.__call__* method first looks for a method called `'repr_' + class.name`; if it does not exist call the default method from the catalogue of plot representation accessible in *plot_repr*.

An example of use of the `Representer` objects can be seen in the *FullTableRepresenter*. In this table representation, for the Bonferroni test result, the input test result is first represented in a table, then the Bonferroni itself is represented in a second table.

Thus the use of the `Representer` is foreseen as:

- use the default methods provided in *TableRepresenter* and *PlotRepresenter*;
- if customisation is needed, you can easily call the additional method available in *table_repr* and *plot_repr*;
- you can also write your own representation method provided they follow the naming convention `'repr_' + class.name`;
- *valjean* calls the `Representer` classes through the *Representation* class.

class valjean.javert.representation.**Representation**(*representer*, *verbosity=Verbosity.DEFAULT*)

Class for representing test results as templates calling the available representers (tables or plots).

This class corresponds to the *Context* role in the *Strategy* design pattern.

__init__(*representer*, *verbosity=Verbosity.DEFAULT*)

'Initilialisation of the *Representation* class with the Representer to use.

Parameters
Representer – representer to use (table, plots, both, etc)

__call__(*result*)

Dispatch handling of *result* to the __call__ methods of the representer class.

class valjean.javert.representation.**Representer**

Base class for representing test results as templates (in the sense of the *templates* module).

This class corresponds to the *Strategy* role in the *Strategy* design pattern. Its subclasses play the role of *ConcreteStrategy*.

__call__(*result*, *verbosity=Verbosity.DEFAULT*)

Dispatch handling of *result* to the appropriate subclass method, based on the name of the class of *result*. This methods essentially implements a simplified, run-time version of the Visitor pattern.

class valjean.javert.representation.**ExternalRepresenter**

This class is the default representation class for external tests, i.e. tests defined by the users who already defined the test representation as templates.

__call__(*result*, *_verbosity*)

Call self as a function.

class valjean.javert.representation.**TableRepresenter**

This class is the default representation class for tables. It contains the overridden *Representer.__call__*.

Advice: users willing to customize the behaviour of TableRepresenter for specific test results should subclass TableRepresenter and define the relevant repr_* methods.

__call__(*result*, *verbosity=Verbosity.DEFAULT*)

Dispatch handling of *result* to the appropriate subclass method, based on the name of the class of *result*. This methods essentially implements a simplified, run-time version of the Visitor pattern.

class valjean.javert.representation.**FullTableRepresenter**

Class to define the specific methods for full representation of tables. This only involve few cases needing the *TableRepresenter.__call__* method like in Bonferroni and Holm-Bonferroni test results.

repr_testresultbonferroni(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestBonferroni* test in two tables:

1. First test result (Student, equal, etc)
2. Bonferroni test result

Parameters
result (TestResultBonferroni) – a test result.

Returns
Representation of a *TestResultBonferroni* as two tables (the first test result and the Bonferroni result).

Return type
list(*TableTemplate*)

repr_testresultholmbonferroni(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestHolmBonferroni* test in two tables:

1. First test result (Student, equal, etc)
2. Holm-Bonferroni test result

Parameters
result (TestResultHolmBonferroni) – a test result.

Returns
Representation of a *TestResultHolmBonferroni* as two tables (the first test result and the Holm-Bonferroni result).

Return type
list(*TableTemplate*)

class valjean.javert.representation.**PlotRepresenter**(*post='default'*)

This class is the default representation class for plots. It contains the overridden *Representer.__call__*.

Advice: users willing to customize the behaviour of *PlotRepresenter* for specific test results should subclass *PlotRepresenter* and define the relevant repr_* methods.

__init__(*post='default'*)

__call__(*result*, *verbosity=Verbosity.DEFAULT*)

Dispatch handling of *result* to the appropriate subclass method, based on the name of the class of *result*. This methods essentially implements a simplified, run-time version of the Visitor pattern.

repr_testresultbonferroni(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestBonferroni* test one a plot (only the input test for the moment) (Student, equal, etc)

Parameters
result (TestResultBonferroni) – a test result.

Returns
Representation of a *TestResultBonferroni* as a plot (the first test result).

Return type
list(*PlotTemplate*)

repr_testresultholmbonferroni(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestHolmBonferroni* test as a plot (Student, equal, etc)

Parameters
result (TestResultHolmBonferroni) – a test result.

Returns
Representation of a *TestResultHolmBonferroni* as a plot (the first test result).

Return type
list(*PlotTemplate*)

class valjean.javert.representation.**FullPlotRepresenter**(*post='default'*)

Class to define the specific methods for full representation of plots. This only involve few cases needing the *PlotRepresenter.__call__* method like in Bonferroni and Holm-Bonferroni test results.

repr_testresultbonferroni(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestBonferroni* test one a plot (only the input test for the moment) (Student, equal, etc)

Parameters
result (TestResultBonferroni) – a test result.

Returns
Representation of a *TestResultBonferroni* as a plot (the first test result).

Return type
list(*PlotTemplate*)

repr_testresultholmbonferroni(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestHolmBonferroni* test as a plot (Student, equal, etc)

Parameters
result (TestResultHolmBonferroni) – a test result.

Returns
Representation of a *TestResultHolmBonferroni* as a plot (the first test result).

Return type
list(*PlotTemplate*)

class valjean.javert.representation.**EmptyRepresenter**

Class that does not generate any templates for any test result.

class valjean.javert.representation.**FullRepresenter**(*post=<function post_treatment>*)

This class generates the fullest possible representation for test results. If anything can be represented as an template, *FullRepresenter* will do it.

__init__(*post=<function post_treatment>*)

Initialisation of *FullRepresenter*.

Two instance objects are built: a *FullTableRepresenter* and a *PlotRepresenter*.

__call__(*result*, *verbosity=Verbosity.DEFAULT*)

Dispatch handling of *result* to all the Representer subclass instance attributes of *FullRepresenter*, based on the name of the class of *result*.

If the representer does not exist in *table_repr* or *plot_repr* a None is returned and replaced here by an empty list. If none of them exist the global return will be an empty list (no None returned from this step).

5.6.3 table_repr – Transform test results into TableTemplate

Module containing all available methods to convert a test result in a table to be converted in rst.

valjean.javert.table_repr.**repr_bins**(*dsref*)

Representation of bins in tables.

When bins are given by edges, representation is `min - max`, when they are given at center, representation is `center`.

Trivial dimensions are not represented, i.e. dimensions where there is only one bin.

If there are more than one non-trivial dimensions, some repetition is expected. For example with two non-trivial dimensions of two bins each one point will be defined by its coordinated in the two dimensions and we expect all the bins to possibily be shown in a table. We expected 4 bins and their associated values in that case.

Let's consider the following dataset: >>> from valjean.eponine.dataset import Dataset >>> import numpy as np >>> from collections import OrderedDict

```
>>> vals = np.arange(6).reshape(1, 2, 1, 3, 1)
>>> errs = np.array([0.1]*6).reshape(1, 2, 1, 3, 1)
>>> bins = OrderedDict([('bacon', np.array([0, 1])),
...                     ('egg', np.array([0, 2, 4])),
...                     ('sausage', np.array([10, 20])),
...                     ('spam', np.array([-5, 0, 5])),
...                     ('tomato', np.array([-2, 2]))])
>>> ds = Dataset(vals, errs, bins=bins)
>>> names, rbins = repr_bins(ds)
>>> print(list(ds.bins.keys()))
['bacon', 'egg', 'sausage', 'spam', 'tomato']
>>> print(names)
['egg', 'spam']
>>> print(ds.shape)
(1, 2, 1, 3, 1)
>>> print([rb.shape for rb in rbins])
[(1, 2, 1, 3, 1), (1, 2, 1, 3, 1)]
```

`'bacon'` and `'sausage'` are trivial dimensions, so won't be represented in the table, but we expect 6 values corresponding to 2 bins in `'egg'` and 3 in `'spam'`. Each value corresponds to a line in the table so each columns should have the same size and the same shape, the shape of the given dataset without trivial dimensions. We then have 3 `'spam'` bins in each `'egg'` bins or 2 `'egg'` bins in each `'spam'` bins. Each couple appears only once.

```
>>> for name, rbin in zip(names, rbins):
...     print(name, ':', rbin.flatten())
egg : ['0 - 2' '0 - 2' '0 - 2' '2 - 4' '2 - 4' '2 - 4']
spam : ['-5' '0' '5' '-5' '0' '5']
```

As expected from the bins, `'egg'` bins are given by edges (`min - max`) while `'spam'` bins are given by center (`center`).

Parameters
dsref (Dataset) – dataset

Returns
list of the non-trivial dimensions and a tuple of the bins

Return type
(list(str), tuple(numpy.ndarray))

The tuple must have the same length as the list of dimensions and the bins inside must have the same shape as `dsref.value`.

valjean.javert.table_repr.**repr_testresultequal**(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestEqual* test.

Parameters

- **result** (TestResultEqual) – a test result.
- **verbosity** (Verbosity) – verbosity level

Returns
list of templates representing a *TestResultEqual*

valjean.javert.table_repr.**repr_equal_summary**(*result*)

Function to generate a summary table for the equal test (only tells if the test was successful or not).

Parameters
result (TestResultEqual) – a test result.

Return type
list(*TextTemplate*)

valjean.javert.table_repr.**repr_equal**(*result*)

Representation of equal test.

Parameters
result (TestResultEqual) – a test result.

Returns
Representation of a *TestResultEqual* as a table.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_testresultapproxequal**(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestApproxEqual* test.

Parameters

- **result** (TestResultApproxEqual) – a test result.
- **verbosity** (Verbosity) – verbosity level

Returns
list of templates representing a *TestResultApproxEqual*

valjean.javert.table_repr.**repr_approx_equal_summary**(*result*)

Function to generate a summary table for the approx equal test (only tells if the test was successful or not).

Parameters
result (TestResultApproxEqual) – a test result.

Return type
list(*TextTemplate*)

valjean.javert.table_repr.**repr_approx_equal**(*result*)

Representation of approx equal test.

Parameters
result (TestResultApproxEqual) – a test result.

Returns
Representation of a *TestResultApproxEqual* as a table.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_testresultstudent**(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestStudent* test.

Parameters

- **result** (TestResultStudent) – a test result.
- **verbosity** (Verbosity) – verbosity level

Returns
list of templates representing a *TestResultStudent*

valjean.javert.table_repr.**repr_student**(*result*)

Representation of Student test result.

Parameters
result (TestResultStudent) – a test result.

Returns
Representation of a *TestResultStudent* as a table.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_student_silent**(*_result*)

Function to generate a silent table for the Student test (only tells if the test was successful or not).

Parameters
result (TestResultStudent) – a Student test result.

Returns
empty list

valjean.javert.table_repr.**repr_student_summary**(*result*)

Function to generate a summary table for the Student test (only tells if the test was successful or not).

Parameters
result (TestResultStudent) – a Student test result.

Return type
list(*TextTemplate*)

valjean.javert.table_repr.**repr_student_intermediate**(*result*)

Function to generate an intermediate table for the Student test: print all the failing results.

Parameters
result (TestResultStudent) – a Student test result.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_testresultbonferroni**(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestBonferroni* test.

Only represents the Bonferroni result, not the input test result.

Parameters

- **result** (TestResultBonferroni) – a test result.
- **verbosity** (Verbosity) – verbosity level

Returns
list of templates representing a *TestResultBonferroni*

valjean.javert.table_repr.**repr_bonferroni**(*result*)

Reprensetation of Bonferroni test result.

Only represents the Bonferroni result, not the input test result.

Parameters
result (TestResultBonferroni) – a test result.

Returns
Representation of a *TestResultBonferroni* as a table.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_bonferroni_summary**(*result*)

Represent the result of a *TestBonferroni* test for the SUMMARY level of verbosity.

Parameters
result (TestResultBonferroni) – a test result.

Returns
Representation of a *TestResultBonferroni* as a table.

Return type
list(*TextTemplate*)

valjean.javert.table_repr.**repr_testresultholmbonferroni**(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestHolmBonferroni* test.

Parameters

- **result** (TestResultHolmBonferroni) – a test result.
- **verbosity** (Verbosity) – verbosity level

Returns
list of templates representing a *TestResultHolmBonferroni*

valjean.javert.table_repr.**repr_holm_bonferroni**(*result*)

Reprensetation of Holm-Bonferroni test result.

Only represents the Holm-Bonferroni result, not the input test result.

Parameters
result (TestResultHolmBonferroni) – a test result.

Returns
Representation of a *TestResultHolmBonferroni* as a table.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_holm_bonferroni_summary**(*result*)

Represent the result of a *TestHolmBonferroni* test for the SUMMARY level of verbosity.

Parameters
result (TestResultHolmBonferroni) – a test result.

Returns
Representation of a *TestResultHolmBonferroni* as a table.

Return type
list(*TextTemplate*)

valjean.javert.table_repr.**percent_fmt**(*num*, *den*)

Format a fraction as a percentage. Example:

```
>>> percent_fmt(2, 4)
'2/4 (50.0%)'
>>> percent_fmt(0, 3)
'0/3 (0.0%)'
>>> percent_fmt(7, 7)
'7/7 (100.0%)'
>>> percent_fmt(0, 0)
'0/0 (???%)'
```

Parameters
- **num** (*int*) – the numerator.
- **den** (*int*) – the denominator.

Return type
str

valjean.javert.table_repr.**repr_testresultstatstasks**(*result*, *verbosity=Verbosity.DEFAULT*)

Represent a *TestResultStatsTasks* as a table. The table breaks down the tasks by status.

Parameters
- **result** (TestResultStatsTasks) – the test result to represent.
- **verbosity** (Verbosity) – verbosity level

Returns
list of templates representing the test result.

valjean.javert.table_repr.**repr_testresultstatstests**(*result*, *verbosity=Verbosity.DEFAULT*)

Represent a *TestResultStatsTests* as a table. The table breaks down the tests by success status.

Parameters
- **result** (TestResultStatsTests) – the test result to represent.

- **verbosity** (Verbosity) – verbosity level

Returns
the tables representing the test result.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_testresultstats**(*result*, *status_ok*, *label*)

Helper function for *repr_testresultstatstests* and *repr_testresultstatstasks*. It generates a table with the *status_ok* value in the first row. Non-null results in other rows are considered as failures, and are highlighted if the count is non-zero. Null results are omitted from the table.

Parameters

- **result** (TestResultStatsTasks *or* TestResultStatsTests) – the test result to represent.
- **status_ok** – the status value that must be considered as a success.
- **label** (*str*) – the type of things that we are testing (`'tests'` or `'tasks'`)

Returns
the tables representing the test result.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_testresultstatstestsbylabels**(*result*, *verbosity=Verbosity.DEFAULT*)

Represent a *TestResultStatsTestsByLabels* as tables. Shape of the table may change according to the number of flags required.

Parameters

- **result** (TestResultStatsTestsByLabels) – the test result to represent.
- **verbosity** (Verbosity) – verbosity level

Returns
the tables representing the test result.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_testresultstatsbylabels**(*result*)

Function to print detailed statistics on tests, per category and sample run.

Parameters
result (TestResultStatsTestsByLabels) – the test result to represent.

Returns
the tables representing the test result.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_testresultstatsbylabels_summary**(*result*)

Function to print detailed statistics on tests, per category and sample run in summary case: only print failed cases.

Parameters
result (TestResultStatsTestsByLabels) – the test result to represent.

Returns
the tables representing the test result.

Return type
list(*TableTemplate*), list(*TextTemplate*)

valjean.javert.table_repr.**repr_testresultmetadata**(*result*, *verbosity=Verbosity.DEFAULT*)

Represent the result of a *TestMetadata* test.

Parameters

- **result** (TestResultMetadata) – a test result.
- **verbosity** (Verbosity) – verbosity level

Returns
list of templates representing a *TestResultMetadata*

valjean.javert.table_repr.**repr_metadata**(*result*)

Function to generate a table from the metadata test results.

Parameters
result (TestResultMetadata) – a test result.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_metadata_full_details**(*result*)

Function to generate a table from the metadata test results.

Parameters
result (TestResultMetadata) – a test result.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_metadata_intermediate**(*result*)

Function to generate a table from the metadata test results.

Parameters
result (TestResultMetadata) – a test result.

Return type
list(*TableTemplate*)

valjean.javert.table_repr.**repr_metadata_summary**(*result*)

Function to generate a table from the metadata test results.

Parameters
result (TestResultMetadata) – a test result.

Return type
list(*TextTemplate*)

valjean.javert.table_repr.**repr_metadata_silent**(*_result*)

Function to generate a table from the metadata test results.

Parameters
result (TestResultMetadata) – a test result.

Returns
empty list

valjean.javert.table_repr.**repr_testresultexternal**(*_result*, *_verbosity=Verbosity.DEFAULT*)

Represent external test as tables -> no table done.

If tables are required they are already done. Their representation in the report is done by *ExternalRepresenter*.

Returns
empty list

valjean.javert.table_repr.**repr_testresultfailed**(*result*, *_verbosity=Verbosity.DEFAULT*)

Represent a failed result as rst text.

Parameters
result (TestResultFailed) – a failed test result.

Return type
list(*TextTemplate*)

5.6.4 plot_repr – Transform test results into PlotTemplate

Module containing all available methods to convert a test result in a table to be converted in rst.

valjean.javert.plot_repr.**dimensions_from_array**(*array_shape*)

Check if array is consistent with 1D plot.

Checks are done on presence of non-trivial and trivial dimensions, trivial being a dimension with size equal to one, non-trivial being for a dimension with at least two values.

Parameters
array_shape (*tuple(int)*) – shape of the **values** array

Returns
indices of the non-trivial dimensions in the shape tuple

Return type
tuple(int)

valjean.javert.plot_repr.**dimensions_and_bins**(*bins*, *array_shape*)

Determine the dimensions of the result from the collections.OrderedDict of bins. It is expected to be the "only non-trivial" dimensions.

Parameters

- **bins** (*collections.OrderedDict(str, numpy.ndarray)*) – bins coming from the results dataset
- **array_shape** (*tuple(int)*) – shape of the **values** array from the results dataset

Returns
dimensions and bins to be used (non-trivial ones), or None if all the dimensions are trivial

Return type
collections.OrderedDict

valjean.javert.plot_repr.**trim_range**(*bins*, *max_ratio=1000*)

Adapt bins range when extreme bins are very large.

This function suggests reasonable ranges for the given bin axes by trimming the extreme bins if they are too wide. For each axis, the extreme bins are trimmed if their width is at least *max_ratio* times larger than the width of the neighbouring bin (the first bin is compared to the second one, and the last bin is compared to the second-last one). If there is no need to change the previous limits a tuple with initial limits is returned for the considered dimension. A boolean is associated to the tuple to precise if the limits have be changed.

The trimmed ranges extend a bit over the clipped bins, so that their content is still visible in the plots.

Parameters

- **bins** (*list(numpy.ndarray)*) – bins of the *PlotTemplate*.
- **max_ratio** (*float*) – the bin size ratio above which bins will be truncated.

Returns
new limits for all dimensions. The list has one item per dimension; the bool indicates whether the axis was actually trimmed.

Return type
list(tuple(float, float, bool))

Examples:

```
>>> bins = [np.array([-1e4, 0, 2, 4, 1e4]),
...         np.array([-1e2, 0, 2, 4, 1e4]),
...         np.array([-1e2, 0, 2, 4, 1e2])]
>>> trim_range(bins)
[(-0.2, 4.2, True), (-100.0, 9.2, True), (-100.0, 100.0, False)]
```

valjean.javert.plot_repr.**fit_curve_ranges**(*curves*, *threshold=0.0*)

Return a set of best-fit limits for the given curves.

This function suggests axis limits for the given curves that make all curves fill the plot area. The limits are determined by looking at the region where any of the curves exceed the given threshold.

Parameters

- **curves** (*list(CurveElements)*) – list of curves
- **threshold** (*float*) – threshold for the detection

Returns
suggested limits for all dimensions

Return type
list((float, float))

Example:

```
>>> bins = [np.array([-1.0, 0.0, 1.0]),
...         np.array([0.0, 5.0, 10.0, 15.0])]
>>> curve1 = CurveElements(np.array([[1.0, 2.0, 3.0],
```

(continues on next page)

(continued from previous page)

```
...                                     [2.0, 3.0, 4.0]]),
...                          bins=bins, legend='curve1')
>>> curve2 = CurveElements(np.array([[1.0, 4.0, 3.0],
...                                     [2.0, 3.0, 2.0]]),
...                          bins=bins, legend='curve2')
>>> fit_curve_ranges([curve1, curve2], threshold = 3.5)
[(-1.0, 1.0), (5.0, 15.0)]
```

valjean.javert.plot_repr.**ranges_union**(*ranges*, *union_axis=0*, *minmax_axis=-1*)

Return the union of the given ranges, axis by axis.

Parameters

- **ranges** (*list or numpy.ndarray*) – an n-dimensional array of range bounds. The array shape can be anything, as long as one of the dimensions has length 2; this axis is assumed to contain the `(min_range, max_range)` pair.
- **union_axis** (anything that *NumPy* understands as an axis.) – the axis along which the union will be computed. If unspecified, the first axis will be assumed.
- **minmax_axis** (*int*) – the length-2 axis along which the range minimum and maximum are stored. If unspecified, the last axis will be assumed.

Returns

an array of union bounds, with the same shape as the input array except for the suppression of *union_axis*.

Return type

numpy.ndarray

Examples:

```
>>> ranges_union([[(-1, 1)]])
array([[-1,  1]])
```

```
>>> ranges_union([[(-5, 5)], [(-3, 8)]])
array([[-5,  8]])
```

```
>>> ranges_union([[(-5, 5), (-1, 1)],
...               [(-3, 8), (-2, 2)]])
array([[-5,  8],
       [-2,  2]])
```

```
>>> ranges_union([[(-5, 5), (-1, 1)],
...               [(-3, 8), (-2, 2)],
...               [(-7, 1), (0, 6)]])
array([[-7,  8],
       [-2,  6]])
```

valjean.javert.plot_repr.**curve_limits**(*curve*, *threshold=0.0*)

Return the set of bounds over each axis where the given curve exceeds the given threshold.

For example, the following curve exceeds the threshold value of 50 in the central bins, which corresponds to *2.0<x<4.0* and *-0.5<y<0.5*:

```
>>> bins = [np.linspace(0.0, 6.0, num=7),
...         np.linspace(-2.5, 2.5, num=6)]
>>> curve = CurveElements(np.array([[  1,   1,   1,   1,   1],
...                                 [  1,  10,  30,  10,   1],
...                                 [  1,  10, 100,  10,   1],
...                                 [  1,  10, 100,  10,   1],
...                                 [  1,  10,  30,  10,   1],
...                                 [  1,   1,   1,   1,   1]]),
...                        bins=bins, legend='curve')
>>> curve_limits(curve, threshold=50)
[(2.0, 4.0), (-0.5, 0.5)]
```

Changing the threshold to 20 modifies the range for the first axis, but not for the second one.

```
>>> curve_limits(curve, threshold=20)
[(1.0, 5.0), (-0.5, 0.5)]
```

The function also works if the number of bin edges is equal to the number of values along a given direction (as opposed to the number of values plus one):

```
>>> bins = [np.array([-1.0, 1.0]), np.array([0.0, 5.0, 10.0, 15.0])]
>>> curve = CurveElements(np.array([[1.0, 2.0, 3.0],
...                                 [2.0, 3.0, 4.0]]),
...                        bins=bins, legend='curve')
>>> curve_limits(curve, threshold = 3.5)
[(1.0, 1.0), (10.0, 15.0)]
```

valjean.javert.plot_repr.**pad_range**(*limits*, *log*, *padding=0.05*)

Pad the given limits.

This function adds a bit of padding to the input limit range. If *log* is *False*, the new limits will be `(limits[0] - padding * delta, limits[1] + padding * delta)`, where `delta = limits[1] - limits[0]`. In log scale (*log=True*), the padded limits are `(limits[0] * exp(-padding * log_delta), limits[1] * exp(padding * log_delta)`, with `log_delta = log(limits[1] / limits[0])`.

Parameters

- **limits** (*(float, float)*) – a pair of floats.
- **log** (*bool*) – whether we are in log scale.
- **padding** (*float*) – the amount of padding to insert.

Returns
new limits

Return type
(float, float)

valjean.javert.plot_repr.**post_treatment**(*templates*, *result*)

Post-treatment of plots after template generate.

For example add names from test result if not done before, suppress zero bins at range edges, etc.

valjean.javert.plot_repr.**build_plot_template_with_dim**(*curves*, *axnames*)

Organise curves in *SubPlotElements*.

If curves are 1D they will all go in the same *SubPlotElements*, if there are in 2 dimensions each curve will have its *SubPlotElements*.

Parameters

- **curves** (*list(CurveElements)*) – list of curves
- **axnames** (*list(str)*) – list of axis names

Return type
list(*SubPlotElements*)

valjean.javert.plot_repr.**repr_datasets_values**(*result*)

Representation of the datasets values from test results obtained from a child test of *TestDataset*.

Examples:

- *TestResultEqual*;
- *TestResultApproxEqual*;
- *TestResultStudent*.

Parameters
result (TestResult) – test result from test on datasets

Return type
list(*PlotTemplate*)

If the dimension cannot be determined or if the dimension is greater than 2 an empty list is returned.

valjean.javert.plot_repr.**repr_testresultequal**(*result*, *_verbosity=Verbosity.DEFAULT*)

Represent the equal test result as a plot.

Parameters
result (TestResultEqual) – a test result.

Return type
list(*PlotTemplate*)

valjean.javert.plot_repr.**repr_testresultapproxequal**(*result*, *_verbosity=Verbosity.DEFAULT*)

Represent the approx equal test result as a plot.

Parameters
result (TestResultApproxEqual) – a test result.

Return type
list(*PlotTemplate*)

valjean.javert.plot_repr.**repr_testresultstudent**(*result*, *verbosity=Verbosity.DEFAULT*)

Plot the Student results according to verbosity.

Parameters
result (TestResultStudent) – a test result.

Returns
Representation of a *TestResultStudent* as a plot.

Return type
list(*PlotTemplate*)

valjean.javert.plot_repr.**repr_student_intermediate**(*result*)

Represent the Student test result as a plot.

By default two *PlotTemplate* are returned in order to get a top plot representing the two series of values and the bottom plot representing Student's t-statistic.

Parameters
result (TestResultStudent) – a test result.

Return type
list(*PlotTemplate*)

valjean.javert.plot_repr.**repr_student_full_details**(*result*)

Represent the Student test result as a plot.

By default two *PlotTemplate* are returned in order to get a top plot representing the two series of values and the bottom plot representing Student's t-statistic.

Parameters
result (TestResultStudent) – a test result.

Return type
list(*PlotTemplate*)

valjean.javert.plot_repr.**repr_student_tstud**(*result*)

Represent the t distribution from a Student test result as a plot.

Parameters
result (TestResultStudent) – a test result.

Return type
list(*PlotTemplate*)

Note: if we have a member `units` in dataset the axis names would be constructed like `name + units['name']`.

If the dimension cannot be determined an empty list is returned.

valjean.javert.plot_repr.**repr_student_pvalues**(*result*)

Representation of p-values from Student test result as a plot.

Parameters
result (TestResultStudent) – a test result.

Return type
list(*PlotTemplate*)

If p-values were not calculated, no *PlotTemplate* is built, so an empty list is returned.

valjean.javert.plot_repr.**repr_testresultmetadata**(*_result*, *_verbosity=Verbosity.DEFAULT*)

Plot metadata test -> no plot done.

Returns
empty list

valjean.javert.plot_repr.**repr_testresultexternal**(*_result*, *_verbosity=Verbosity.DEFAULT*)

Plot external test -> no plot done.

If plots are required they are already done. Their representation in the report is done by *ExternalRepresenter*.

Returns
empty list

valjean.javert.plot_repr.**repr_testresultfailed**(*_result*, *_verbosity=Verbosity.DEFAULT*)

Plot failed test -> no plot done.

Returns
empty list

valjean.javert.plot_repr.**repr_testresultstats**(*result*, *status_ok*, *label*)

Represent result from statistical test as a plot (pie chart probably).

Null results are omitted from the table.

Parameters
result (TestResult) – result from a statistical test

Return type
list(*PlotTemplate*)

valjean.javert.plot_repr.**repr_testresultstatstasks**(*result*, *_verbosity=Verbosity.DEFAULT*)

Represent result from statistical test on tasks as a plot (pie chart).

Parameters
result (TestResultStatsTasks) – result from tasks statistics

Return type
list(*PlotTemplate*)

valjean.javert.plot_repr.**repr_testresultstatstests**(*result*, *_verbosity=Verbosity.DEFAULT*)

Represent result from statistical test on tests as a plot (pie chart).

Parameters
result (TestResultStatsTests) – result from tasks statistics

Return type
list(*PlotTemplate*)

valjean.javert.plot_repr.**repr_statstestsby2labels**(*result*)

Plot statistics on the tests classified by 2 labels.

Caution: the tests are summed over the second label, only the first one will be written on the plot. This is for summaries.

Return type
list(*PlotTemplate*)

valjean.javert.plot_repr.**repr_testresultstatstestsbylabels**(*result*, *_verbosity=Verbosity.DEFAULT*)

Plot statistics on the tests classified by labels if 1 or labels.

Return type
list(*PlotTemplate*)

5.6.5 `formatter` – Abstract class for formatters

This module contains the *Formatter* abstract base class. All other formatters must derive from it.

class valjean.javert.formatter.**Formatter**

Abstract base class for any formatter.

abstract header(*name*, *depth*)

Format a section header.

text(*text*)

Format some text.

template(*item*)

Convert an item to the relevant format.

5.6.6 `mpl` – Convert templates to plots

This module provides the classes to convert test results to plots using `matplotlib.pyplot`.

MplPlot objects take as input *PlotTemplate* containing curves (*CurveElements*) classified by sub-plots (*SubPlotElements*).

The format, or rendering, of the plot can be set using the rcParams but also some predefined parameters on which the class cycle like colors, markers shape and filling.

By default the first color is black and is used only once: it is excluded from the cycle on colors. It is typically reserved for the reference but can be reused if the first index is used for another curve.

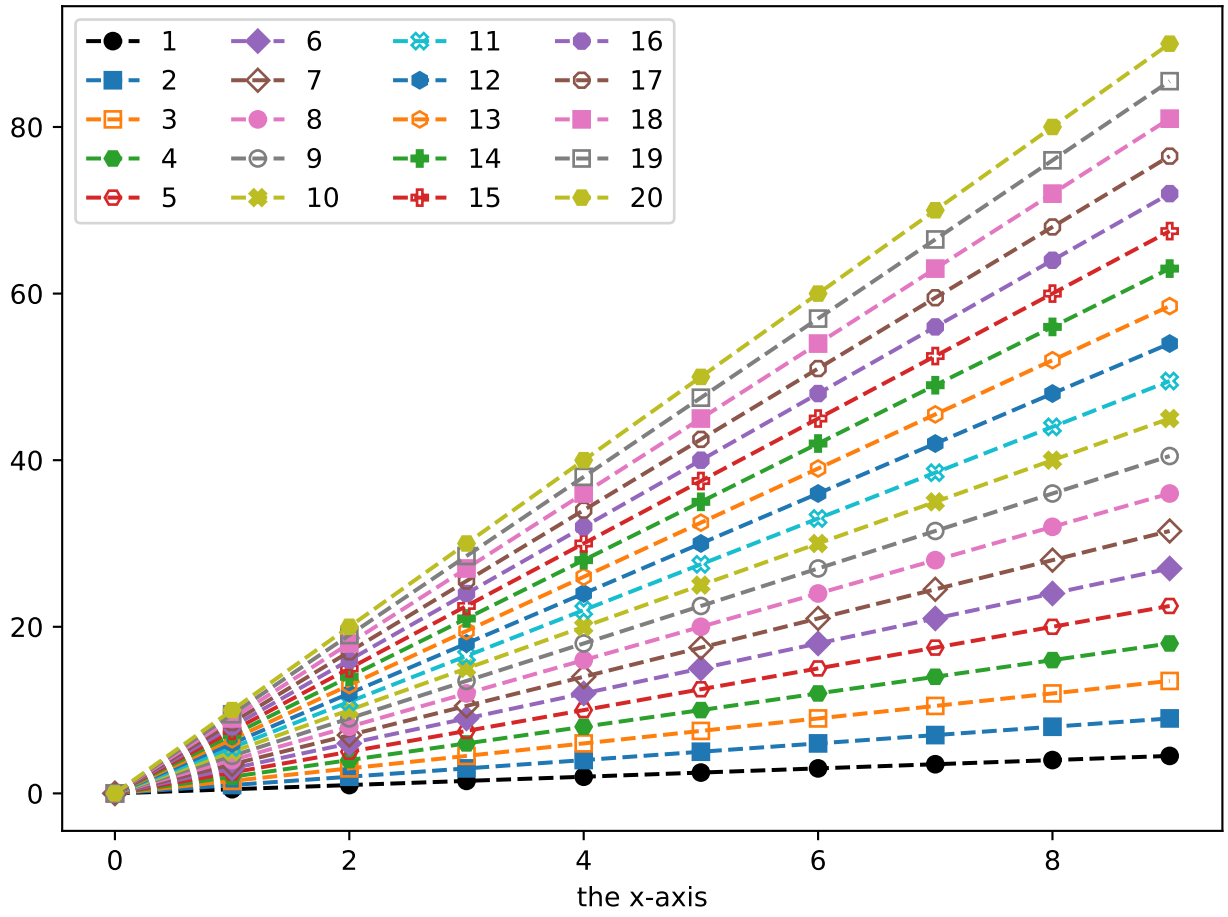
Plots can be obtained with the following for example:

```
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = (np.array(np.arange(10)),)
>>> lcurves = [CurveElements(values=bins[0]*0.5*(icurve+1), bins=bins,
...                          legend=str(icurve+1), index=icurve)
...            for icurve in range(20)]
>>> pltit = PlotTemplate(subplots=[SubPlotElements(
...     curves=lcurves, axnames=('the x-axis', ''), ptype='1D')])
>>> from valjean.javert.mpl import MplPlot
>>> mplplt = MplPlot(pltit)
>>> fig, _ = mplplt.draw()
```

Additional subplots can be drawn if required. The style of the curves is fixed by the index (see *CurveElements*).

```
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
```

(continues on next page)


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
0
20
40
60
80
0
2
4
6
8
the x-axis

(continued from previous page)

```
>>> bins = (np.array(np.arange(10)),)
>>> lcurves1 = []
>>> for icurve in range(3):
...     lcurves1.append(CurveElements(
...         values=bins[0][1:]*0.5*(icurve+1) + icurve*(-1)**(icurve),
...         bins=bins, legend=str(icurve), index=icurve))
>>> sbpe1 = SubPlotElements(curves=lcurves1, axnames=('the x-axis', ''))
>>> lcurves2 = []
>>> for icurve in range(1, 3):
...     lcurves2.append(CurveElements(
...         values=lcurves1[icurve].values/lcurves1[0].values, bins=bins,
...         legend=str(icurve+1), index=icurve))
>>> sbpe2 = SubPlotElements(curves=lcurves2,
...                         axnames=('the x-axis', 'C/ref'))
>>> lcurves3 = []
>>> for icurve in range(1, 3):
...     lcurves3.append(CurveElements(
...         values=((lcurves1[icurve].values-lcurves1[0].values)
...                 /lcurves1[0].values),
...         bins=bins, legend=str(icurve+1), index=icurve))
>>> sbpe3 = SubPlotElements(curves=lcurves3,
...                         axnames=('the x-axis', '(C-ref)/ref'))
>>> pltit = PlotTemplate(subplots=[sbpe1, sbpe2, sbpe3])
>>> from valjean.javert.mpl import MplPlot
>>> mplplt = MplPlot(pltit)
>>> fig, _ = mplplt.draw()
```

These examples also show the default style of the plots.

Style setup

Some style parameters are available in the object *MplStyle*: general style, colors of markers and lines (expected to be the same for the same curve), shapes and fills of markers. Legend keyword arguments can also be given.

General style

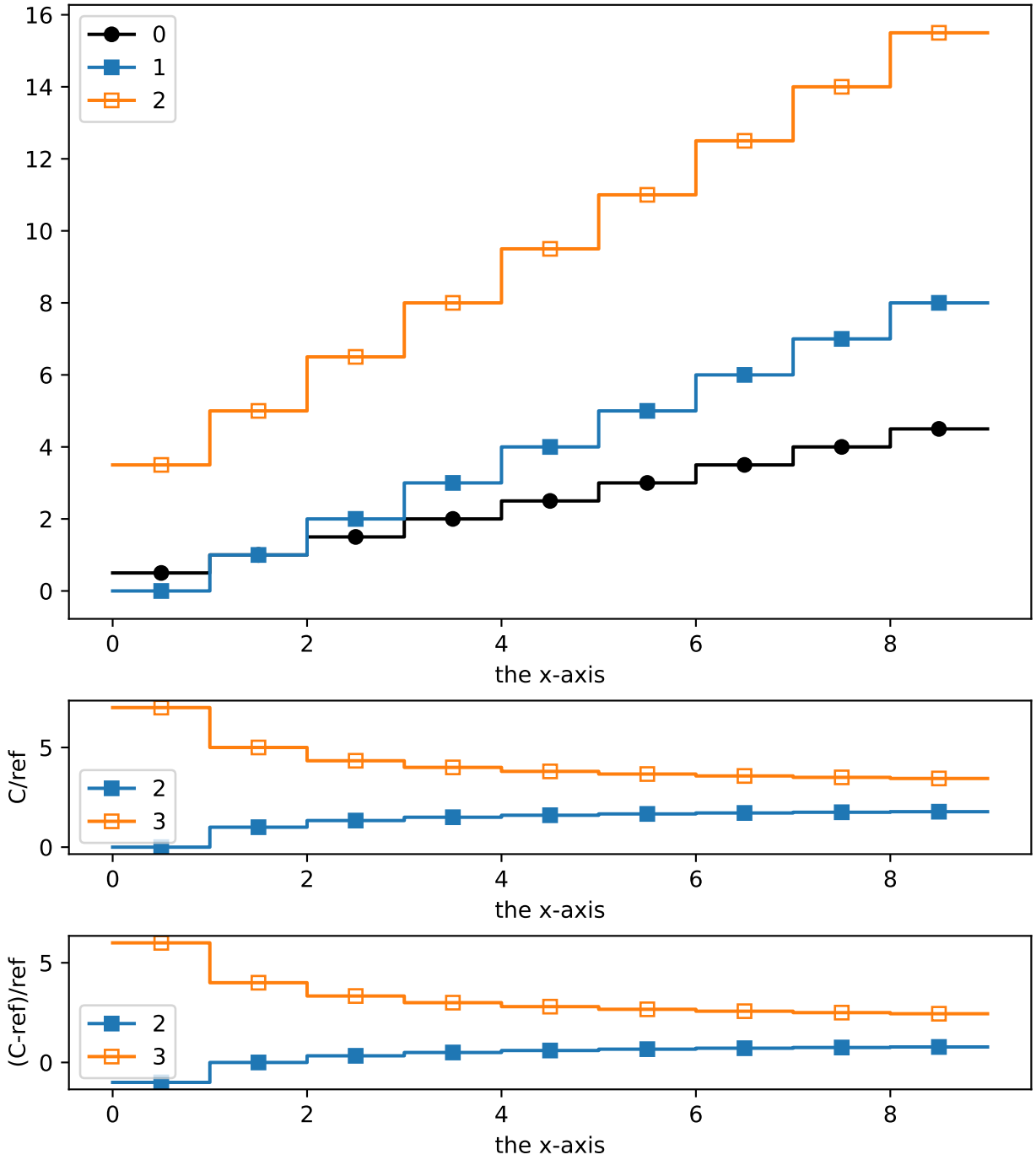
It is possible to change the general style of plots using a predefined one or to use different markers. The predefined styles can be seen in matplotlib styles or be obtained thanks to

```
>>> import matplotlib.pyplot as plt
>>> print(plt.style.available)
```

For example, we can have:

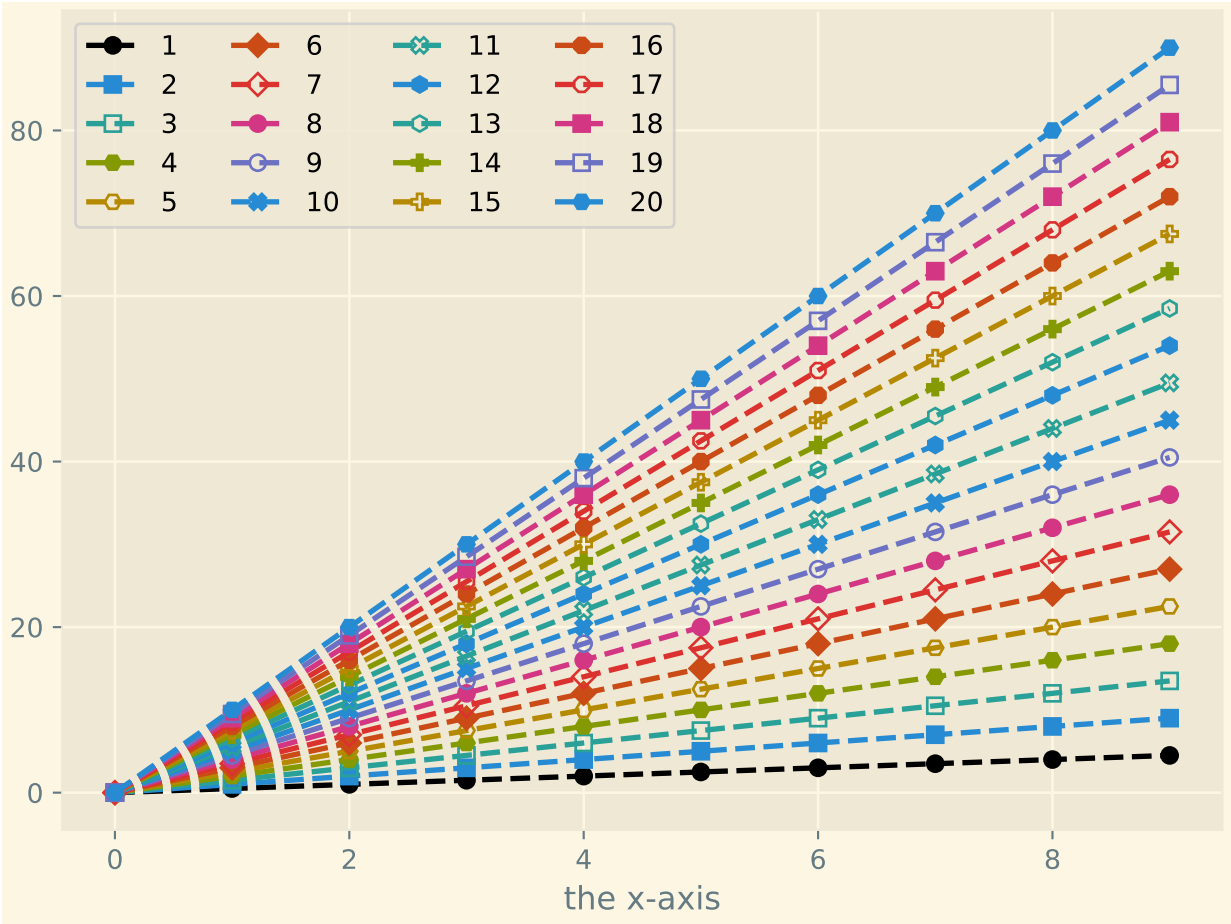
```
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = (np.array(np.arange(10)),)
>>> lcurves = []
>>> for icurve in range(20):
...     lcurves.append(CurveElements(values=bins[0]*0.5*(icurve+1),
...                                  bins=bins, legend=str(icurve+1),
```

(continues on next page)


0
1
2
the x-axis
C/ref
2
3
the x-axis
(C-ref)/ref
2
3
the x-axis

(continued from previous page)

```
...                                     index=icurve))
>>> sbpe = SubPlotElements(curves=lcurves, axnames=['the x-axis', ''])
>>> pltit = PlotTemplate(subplots=[sbpe])
>>> from valjean.javert.mpl import MplPlot, MplStyle
>>> mplplt = MplPlot(pltit, style=MplStyle(style='Solarize_Light2'))
>>> fig, _ = mplplt.draw()
```



Colors and markers

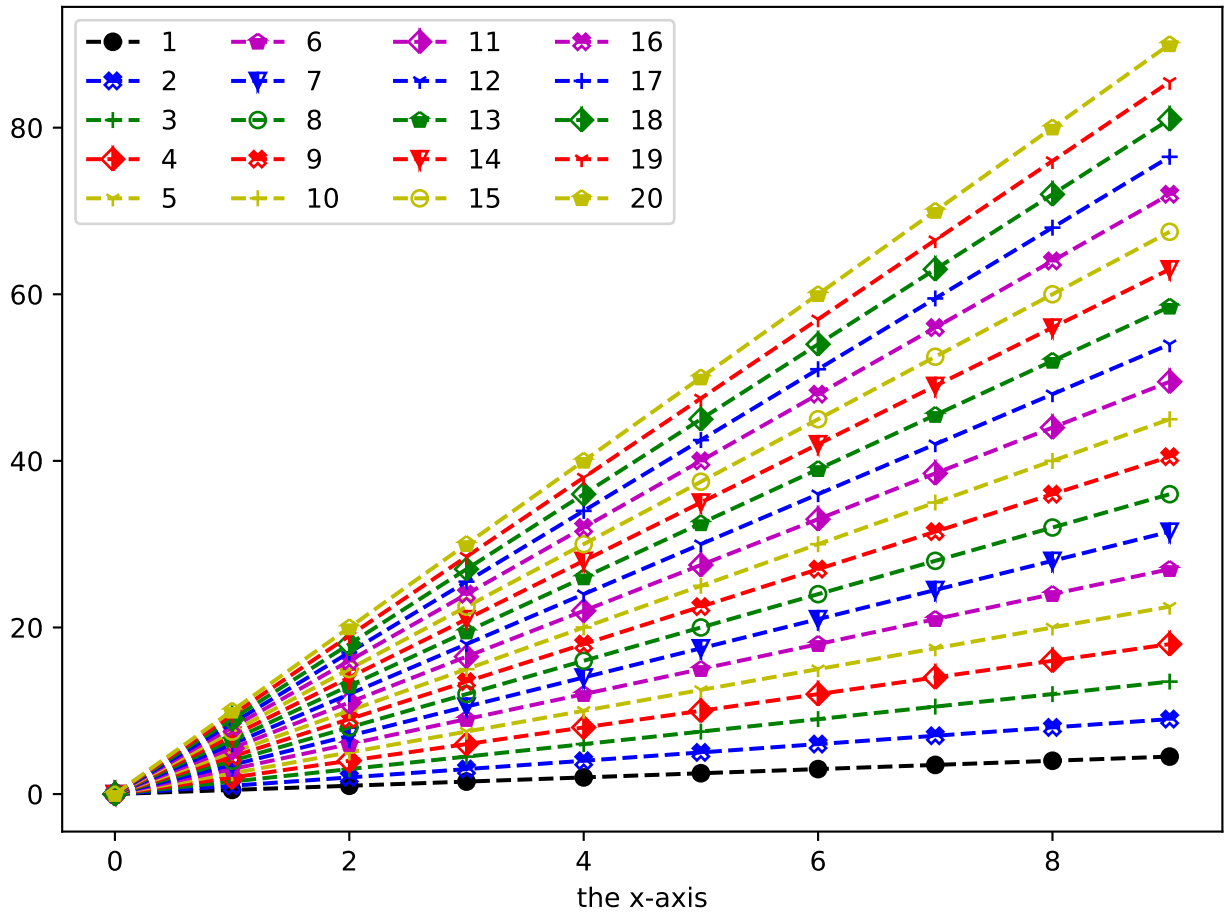
Colors and markers can also be changed directly:

```
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = [np.array(np.arange(10))]
>>> lcurves = []
>>> for icurve in range(20):
...     lcurves.append(CurveElements(values=bins[0]*0.5*(icurve+1),
...                                  bins=bins, legend=str(icurve+1),
...                                  index=icurve))
>>> sbpe = SubPlotElements(curves=lcurves, axnames=['the x-axis', ''])
```

(continues on next page)

(continued from previous page)

```
>>> pltit = PlotTemplate(subplots=[sbpe])
>>> from valjean.javert.mpl import MplPlot, MplStyle
>>> style = MplStyle(colors=['b', 'g', 'r', 'y', 'm'],
...                  mshape=['X', '+', 'D', '1', 'p', 'v', 'o'],
...                  mfill=['top', 'full', 'right', 'none', 'bottom',
...                         'left', 'none'])
>>> mplplt = MplPlot(pltit , style=style)
>>> fig, _ = mplplt.draw()
```



Legends

By default the legend is represented on all panels at the location `matplotlib.pyplot` determines like in *plot 3 panels*.

If you would prefer to get only one legend for all panels, the `suppress_legends` argument in *`PlotTemplate`* should be used. In that case, only fine for 1D plots, the legend will be placed on the largest panel by default.

In the style any keyword argument accepted by matplotlib can be given to modify for example the legend position. This is can be found in the legend documentation.

The next example show the *plot 3 panels* with only one legend which position and shape have been modified.

```
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = [np.array(np.arange(10))]
>>> lcurves1 = []
>>> for icurve in range(3):
...     lcurves1.append(CurveElements(
...         values=bins[0][1:]*0.5*(icurve+1) + icurve*(-1)**(icurve),
...         bins=bins, legend=str(icurve), index=icurve))
>>> sbpe1 = SubPlotElements(curves=lcurves1, axnames=['the x-axis', ''])
>>> lcurves2 = []
>>> for icurve in range(1, 3):
...     lcurves2.append(CurveElements(
...         values=lcurves1[icurve].values/lcurves1[0].values,
...         bins=bins, legend=str(icurve+1)+' vs 0', index=icurve))
>>> sbpe2 = SubPlotElements(curves=lcurves2,
...                         axnames=['the x-axis', 'C/ref'])
>>> lcurves3 = []
>>> for icurve in range(1, 3):
...     lcurves3.append(CurveElements(
...         values=((lcurves1[icurve].values-lcurves1[0].values)
...                 /lcurves1[0].values),
...         bins=bins, legend=str(icurve+1)+' vs 0', index=icurve))
>>> sbpe3 = SubPlotElements(curves=lcurves3,
...                         axnames=['the x-axis', '(C-ref)/ref'])
>>> pltit = PlotTemplate(subplots=[sbpe1, sbpe2, sbpe3],
...                      suppress_legends=True)
>>> from valjean.javert.mpl import MplPlot, MplStyle
>>> style = MplStyle(legends={'loc': 3, 'bbox_to_anchor': (0., 1., 1, 1),
...                           'mode': 'expand'})
>>> mplplt = MplPlot(pltit, style=style)
>>> fig, _ = mplplt.draw()
```

2D plots

2D plots are also done via the class *MplPlot*. The plot type `ptype` in *PlotTemplate* should be `'2D'`. The principle is the same as for 1D plots. Three axes are expected. Each curve has its own plot, no superposition is done, so one subplot is expected to contain only one curve. Each subplot can then have its own properties.

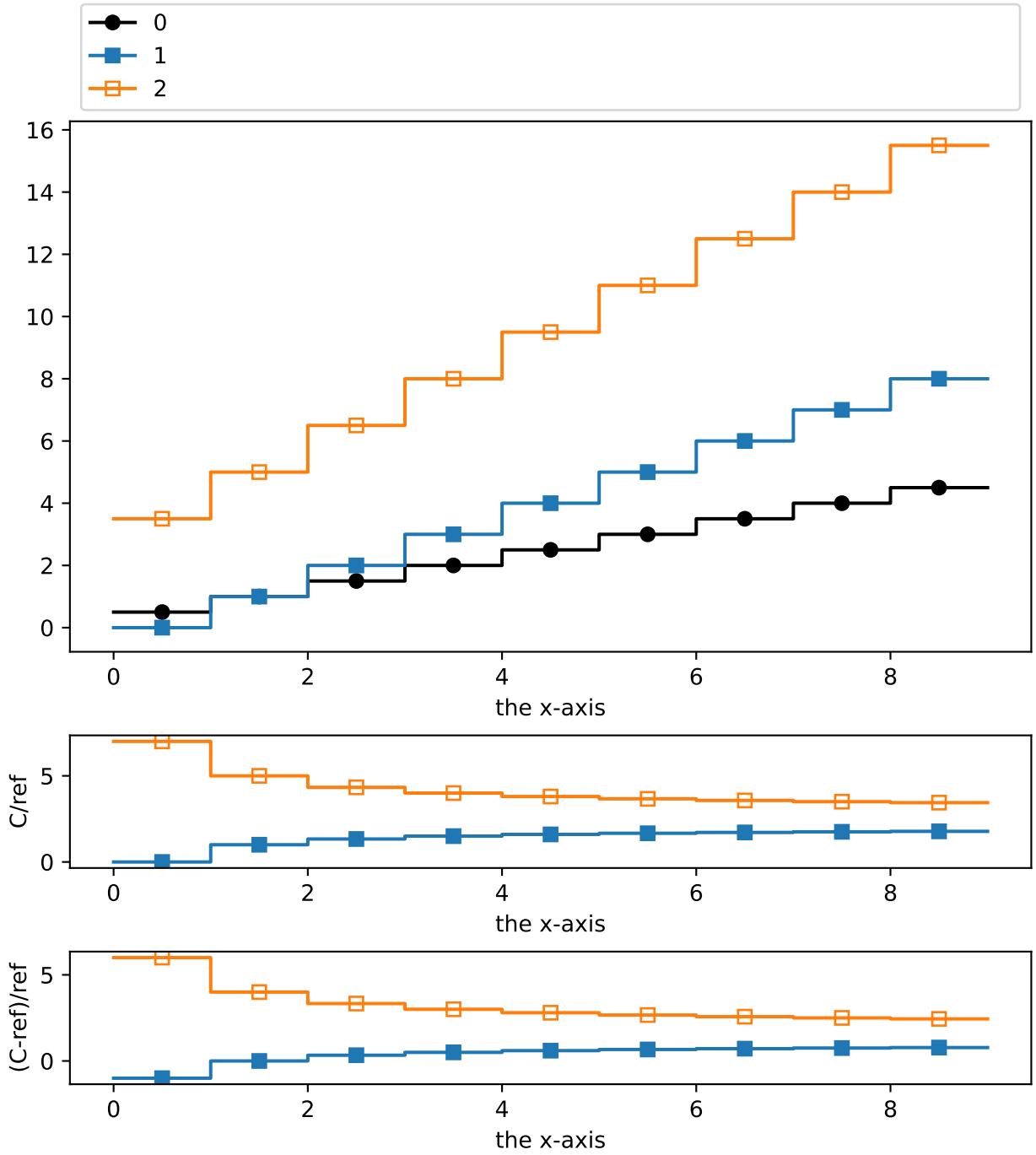
The colorbar axis label is set using the third axis name.

There is no real legend, so `legend` is used as title of each plot.

The index is currently not used.

```
>>> from collections import OrderedDict
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = [np.arange(6), np.arange(17, step=2)]
>>> axnames = ['x', 'y']
>>> incv = np.arange(1, 41).reshape(5, 8)
>>> decv = np.arange(1, 41)[::-1].reshape(5, 8)
>>> lsplts = []
>>> lsplts.append(SubPlotElements(
```

(continues on next page)


0
1
2
16
14
12
10
8
6
4
2
0
0
2
4
6
8
the x-axis
C/ref
5
0
0
2
4
6
8
the x-axis
(C-ref)/ref
5
0
0
2
4
6
8
the x-axis

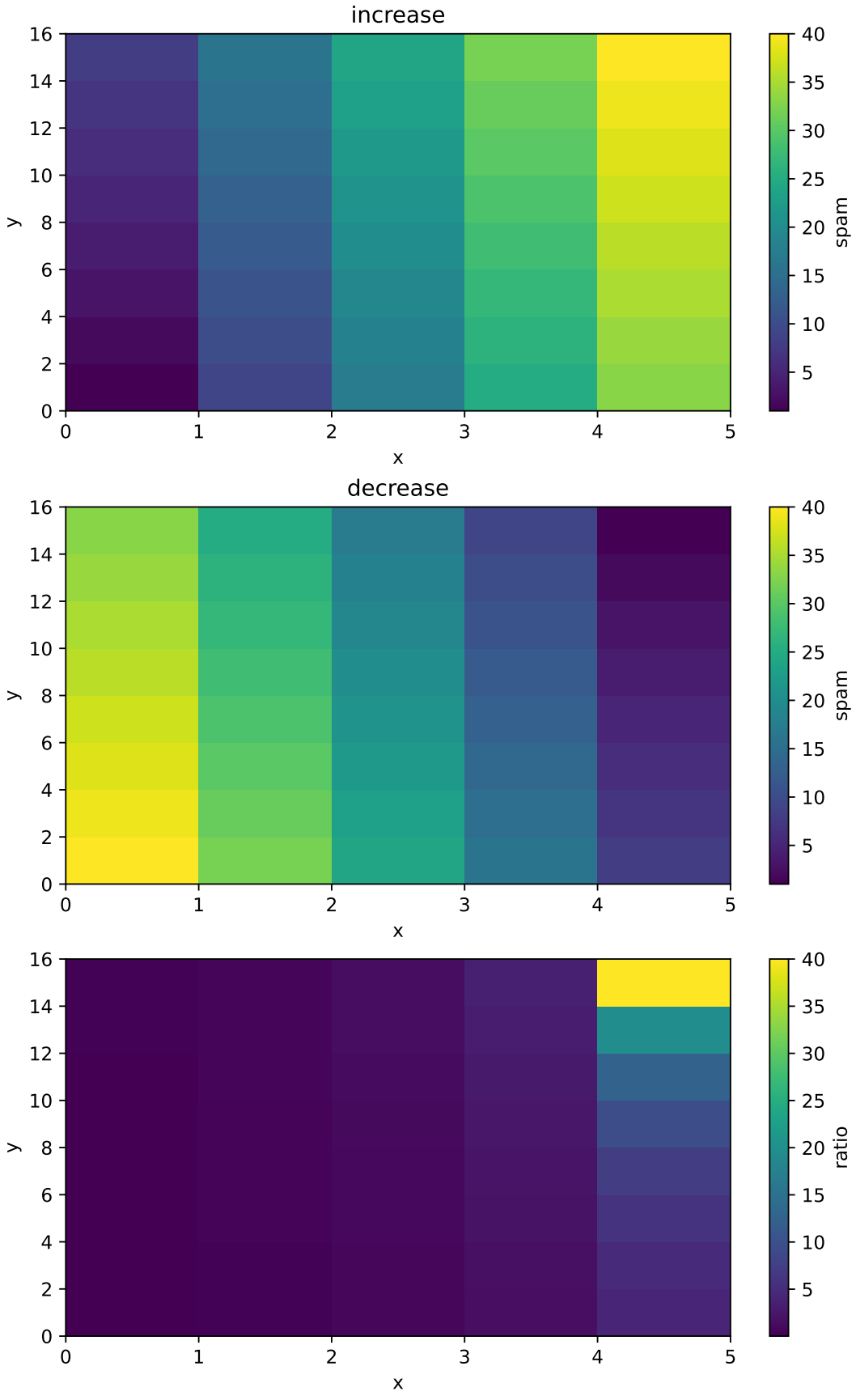
(continued from previous page)

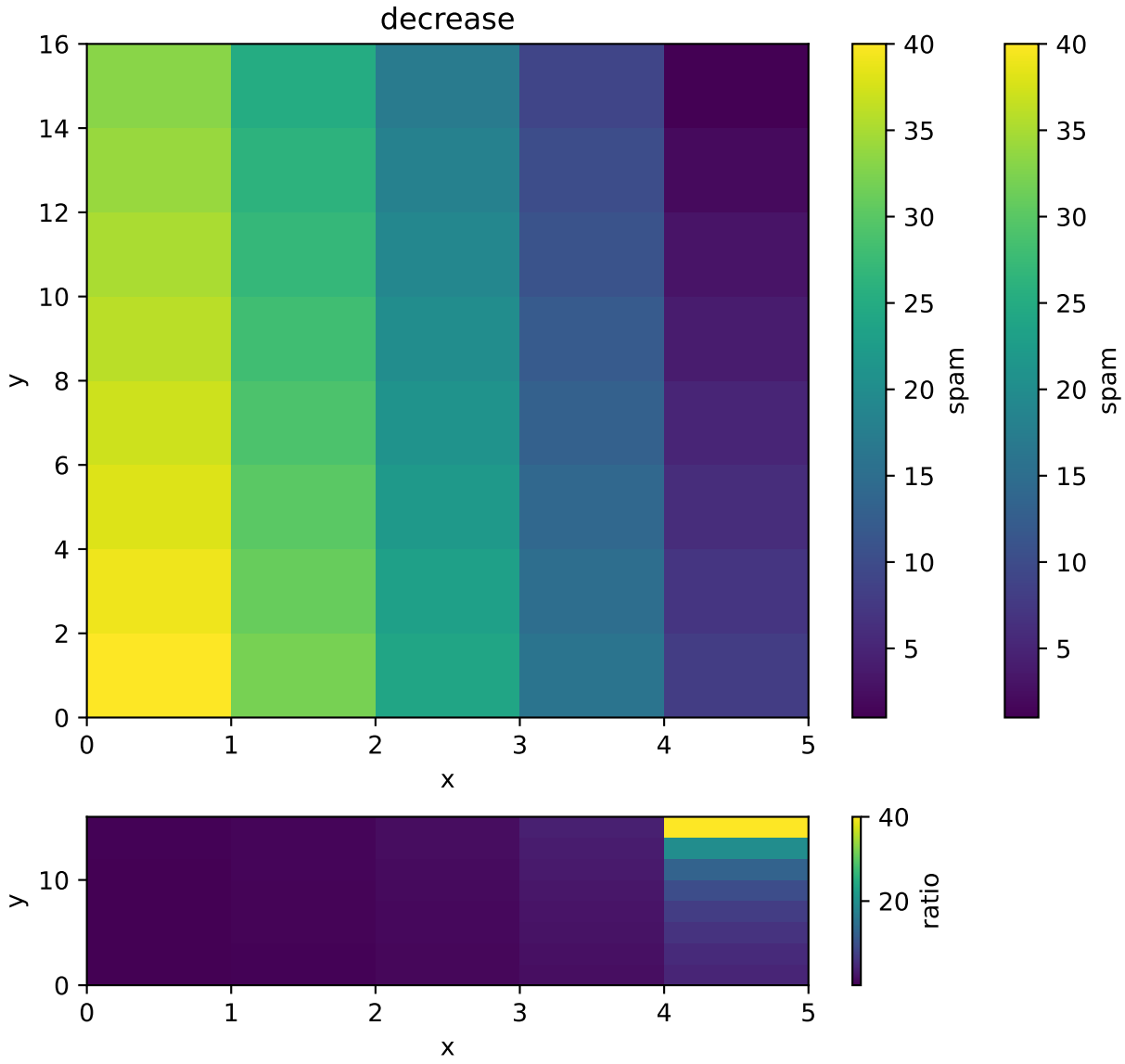
```
...     curves=[CurveElements(
...         values=incv, bins=bins, legend='increase', index=0)],
...     axnames=['x', 'y', 'spam'], ptype='2D'))
>>> lsplts.append(SubPlotElements(
...     curves=[CurveElements(
...         values=decv, bins=bins, legend='decrease', index=0)],
...     axnames=['x', 'y', 'spam'], ptype='2D'))
>>> lsplts.append(SubPlotElements(
...     curves=[CurveElements(
...         values=incv/decv, bins=bins, legend='', index=1)],
...     axnames=['x', 'y', 'ratio'], ptype='2D'))
>>> pltnd = PlotTemplate(subplots=lsplts, small_subplots=False)
>>> from valjean.javert import mpl
>>> mplplt = mpl.MplPlot(pltnd)
>>> fig, _ = mplplt.draw()
```

Note: Per default, additional subplots are small ones, it is probably better in 2D case to set the parameter `small_subplots` to `False` in the *`PlotTemplate`*.

Warning: Requesting more than one curve on a subplot will emit a warning but give unexpected results (typically only one of the 2D plot will be shown).

```
>>> from collections import OrderedDict
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = [np.arange(6), np.arange(17, step=2)]
>>> axnames = ['x', 'y']
>>> incv = np.arange(1, 41).reshape(5, 8)
>>> decv = np.arange(1, 41)[::-1].reshape(5, 8)
>>> lsplts = []
>>> lsplts.append(SubPlotElements(
...     curves=[CurveElements(values=incv, bins=bins, legend='increase'),
...             CurveElements(values=decv, bins=bins, legend='decrease')],
...     axnames=['x', 'y', 'spam'], ptype='2D'))
>>> lsplts.append(SubPlotElements(
...     curves=[CurveElements(
...         values=incv/decv, bins=bins, legend='', index=1)],
...     axnames=['x', 'y', 'ratio'], ptype='2D'))
>>> pltnd = PlotTemplate(subplots=lsplts)
>>> from valjean.javert import mpl
>>> mplplt = mpl.MplPlot(pltnd)
>>> fig, _ = mplplt.draw()
```


increase
decrease
x
y
spam
ratio
0
1
2
3
4
5
0
2
4
6
8
10
12
14
16
5
10
15
20
25
30
35
40


decrease
y
x
spam
spam
ratio

Customization

Some customizations can be done for each subplot with the attributes parameter of *SubPlotElements*: limits to adapt axes ranges, logarithmic scale or lines.

Using the previous 1D example:

```
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = [np.array(np.arange(10))]
>>> lcurves1 = [CurveElements(
...     values=bins[0][1:]*0.5*(icurve+1) + icurve*(-1)**(icurve),
...     bins=bins, legend=str(icurve), index=icurve)
...              for icurve in range(3)]
>>> sbpe1 = SubPlotElements(curves=lcurves1, axnames=['the x-axis', ''])
>>> lcurves2 = [CurveElements(
...     values=lcurves1[icurve].values/lcurves1[0].values, bins=bins,
...     legend=str(icurve+1)+' vs 0', index=icurve)
...              for icurve in range(1, 3)]
>>> sbpe2 = SubPlotElements(curves=lcurves2,
...                         axnames=['the x-axis', 'C/ref'])
>>> lcurves3 = [CurveElements(
...     values=((lcurves1[icurve].values-lcurves1[0].values)
...             /lcurves1[0].values),
...     bins=bins, legend=str(icurve+1)+' vs 0', index=icurve)
...              for icurve in range(1, 3)]
>>> sbpe3 = SubPlotElements(curves=lcurves3,
...                         axnames=['the x-axis', '(C-ref)/ref'])
>>> sbpe1.attributes.logx = True
>>> sbpe2.attributes.limits = [(2, 7)]
>>> sbpe3.attributes.logy = True
>>> pltit = PlotTemplate(subplots=[sbpe1, sbpe2, sbpe3],
...                      small_subplots=False)
>>> from valjean.javert.mpl import MplPlot, MplStyle
>>> mplplt = MplPlot(pltit)
>>> fig, _ = mplplt.draw()
```

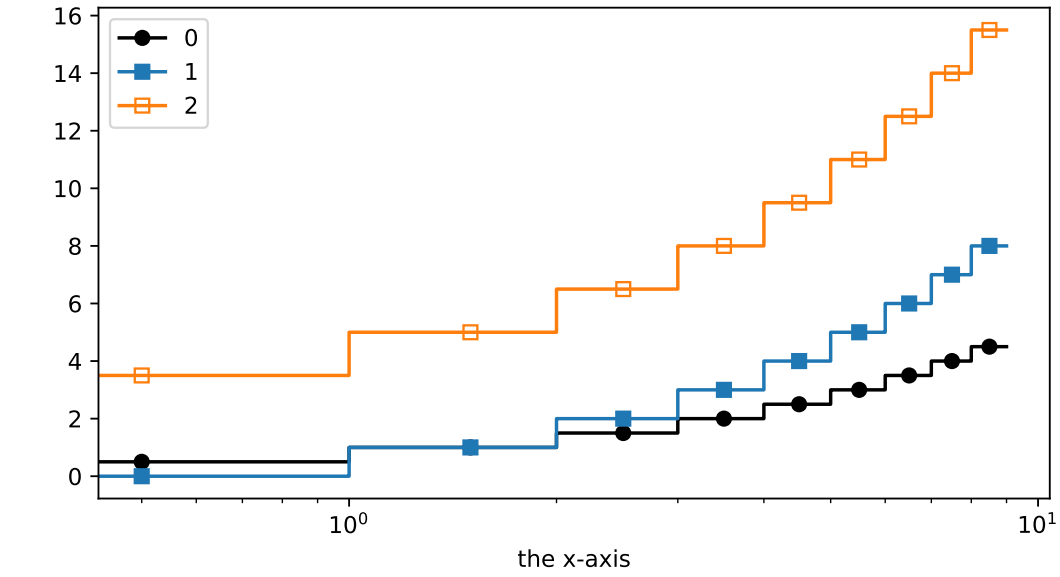
Customizations specific to the backend, here `matplotlib.pyplot`, are passed thanks to the `backend_kw` argument of *PlotTemplate*. The keywords that are currently supported by the `matplotlib.pyplot` backend and used in *valjean* are:

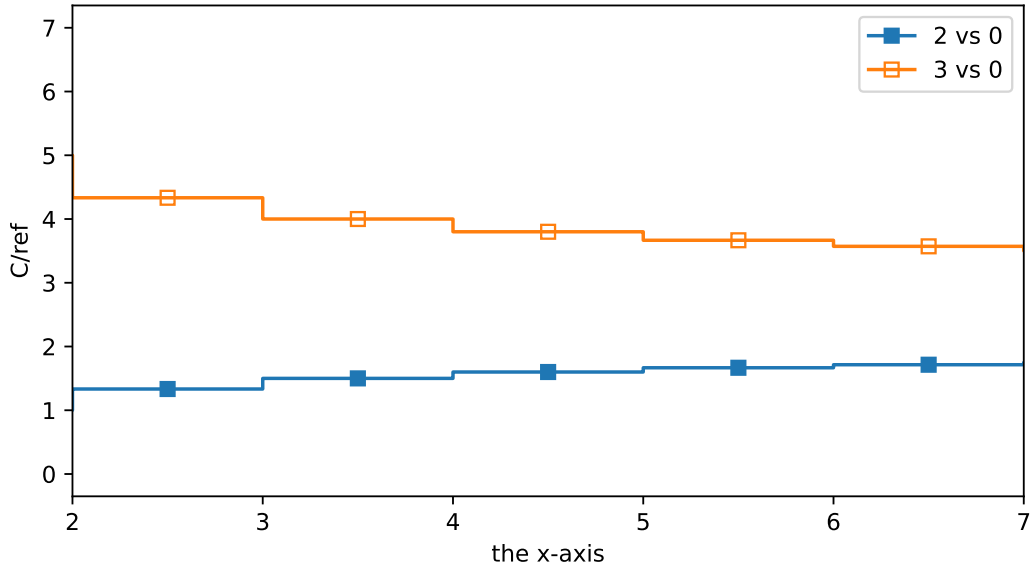
`nrows` and `ncols`
: These keys should be associated to integers. They determine the number of subplots, in case the *PlotTemplate* object contains several *SubPlotElements*. If `ncols` is given, `nrows` also has to be given.

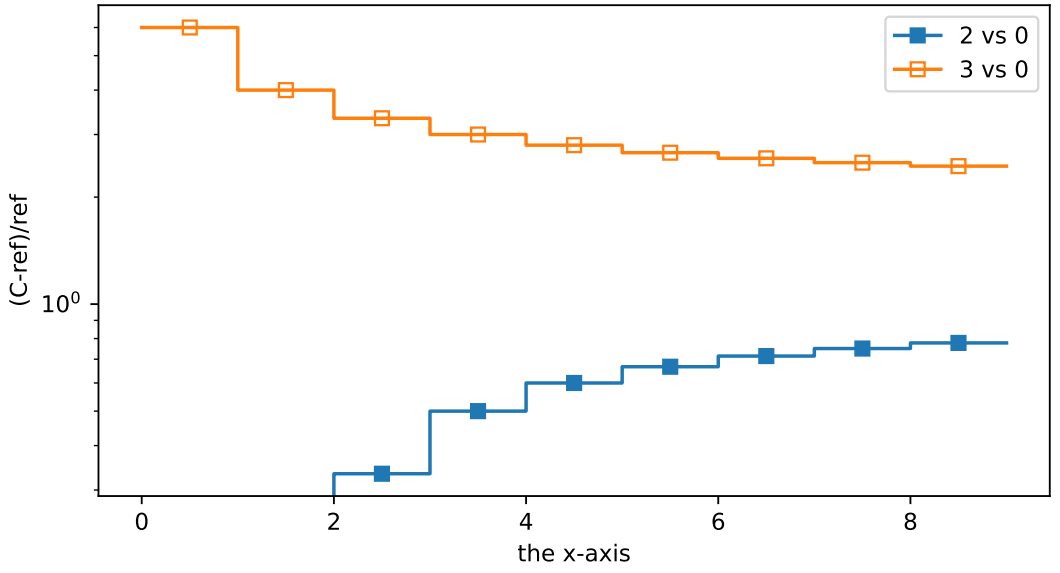
`figsize`
: This key should be associated to tuple of floats (width and height in inches) to specify the figure size.

`subplot_kw`
: This key should be associated to a dictionary of keywords that will be passed as-is to `matplotlib.pyplot.subplots`.

`gridspec_kw`
: This key should be associated to a dictionary to be passed to the grid constructor, see `matplotlib.pyplot.subplots` and `matplotlib.gridspec.GridSpec`.


0
1
2
the x-axis

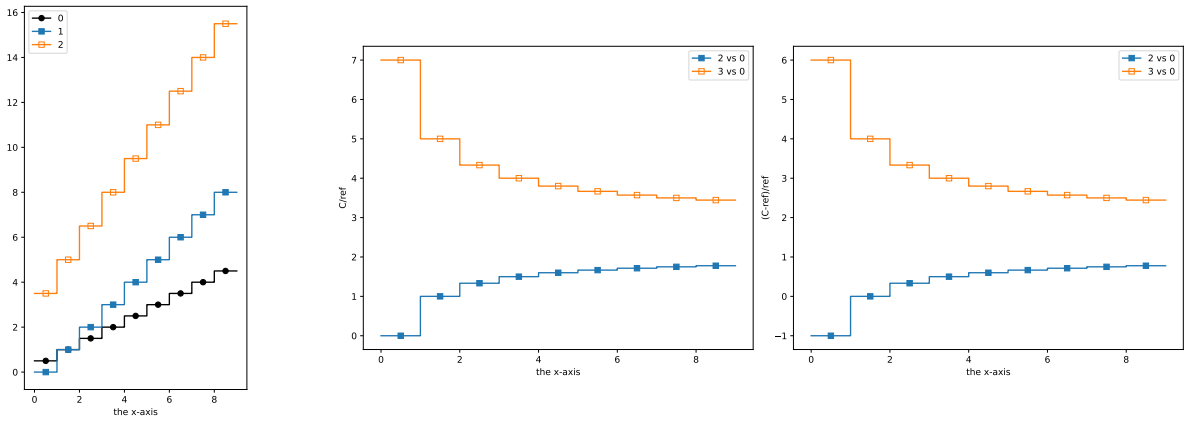

2 vs 0
3 vs 0
C/ref
the x-axis


2 vs 0
3 vs 0
(C-ref)/ref
the x-axis

All possible keyword arguments that can be passed to `matplotlib.pyplot.subplots` are normally supported.

Example is given from the previous 1D one:

```
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = [np.array(np.arange(10))]
>>> lcurves1 = [CurveElements(
...     values=bins[0][1:]*0.5*(icurve+1) + icurve*(-1)**(icurve),
...     bins=bins, legend=str(icurve), index=icurve)
...             for icurve in range(3)]
>>> sbpe1 = SubPlotElements(curves=lcurves1, axnames=['the x-axis', ''])
>>> lcurves2 = [CurveElements(
...     values=lcurves1[icurve].values/lcurves1[0].values, bins=bins,
...     legend=str(icurve+1)+' vs 0', index=icurve)
...             for icurve in range(1, 3)]
>>> sbpe2 = SubPlotElements(curves=lcurves2,
...                         axnames=['the x-axis', 'C/ref'])
>>> lcurves3 = [CurveElements(
...     values=((lcurves1[icurve].values-lcurves1[0].values)
...             /lcurves1[0].values),
...     bins=bins, legend=str(icurve+1)+' vs 0', index=icurve)
...             for icurve in range(1, 3)]
>>> sbpe3 = SubPlotElements(curves=lcurves3,
...                         axnames=['the x-axis', '(C-ref)/ref'])
>>> pltit = PlotTemplate(subplots=[sbpe1, sbpe2, sbpe3],
...                      small_subplots=False,
...                      backend_kw={'ncols': 3, 'nrows': 1,
...                                  'subplot_kw': {'aspect': 'equal'}})
>>> from valjean.javert.mpl import MplPlot, MplStyle
>>> mplplt = MplPlot(pltit)
>>> fig, _ = mplplt.draw()
```



Customization also works on 2D plots. In addition the colorscale and colormap can be put in logarithmic scale.

```
>>> from collections import OrderedDict
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
```

(continues on next page)

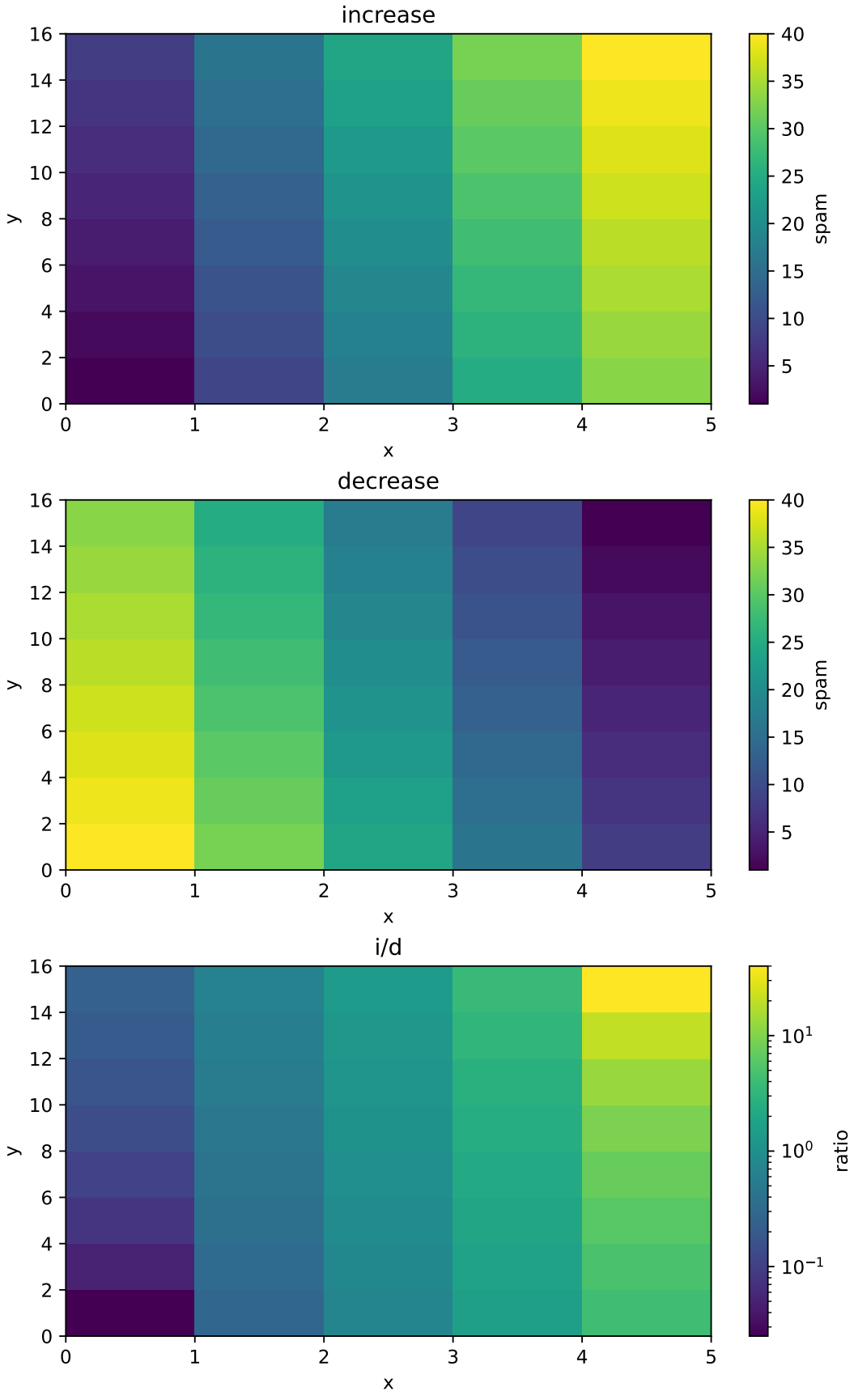
(continued from previous page)

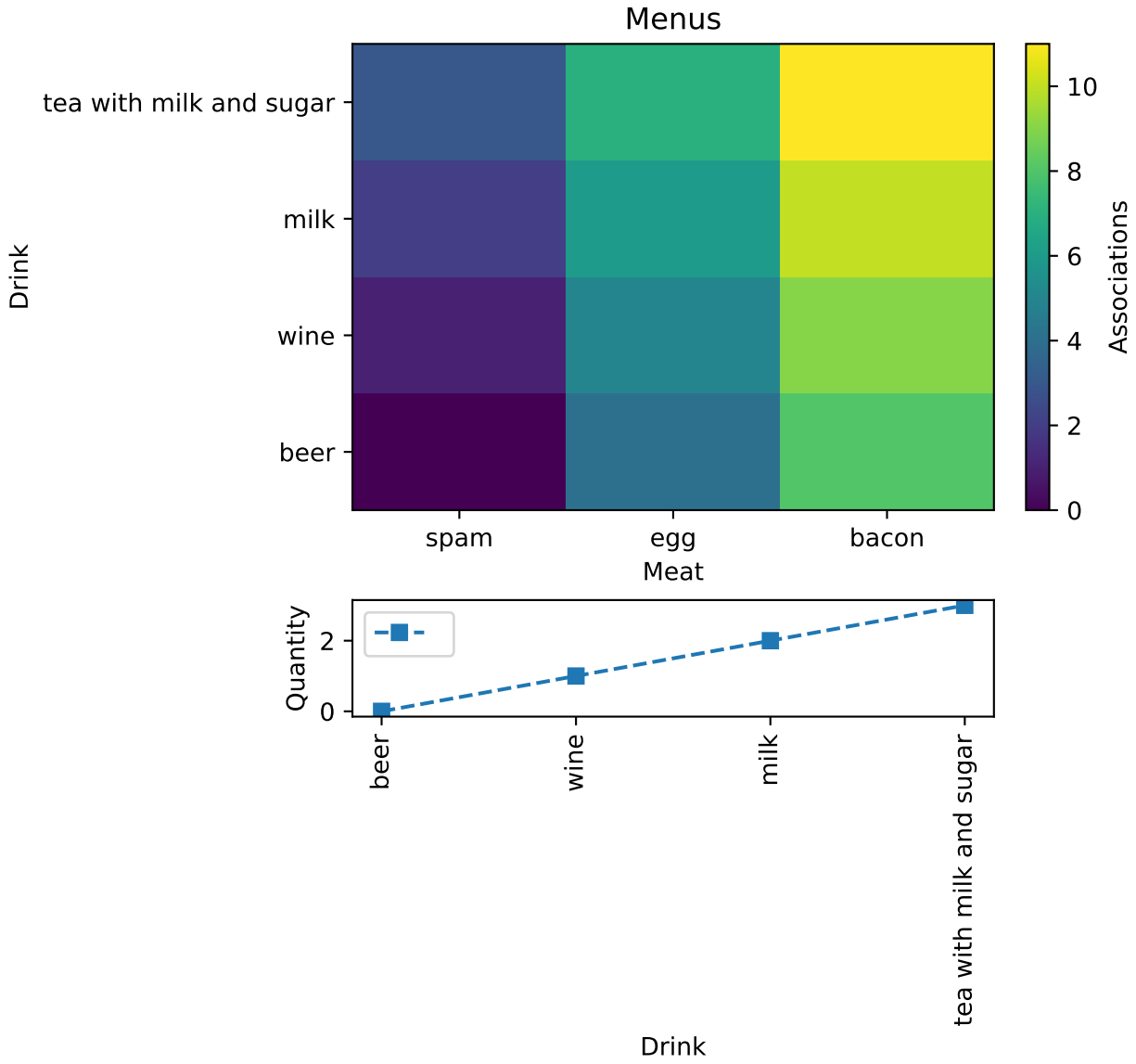
```
>>> bins = [np.arange(6), np.arange(17, step=2)]
>>> axnames = ['x', 'y']
>>> incv = np.arange(1, 41).reshape(5, 8)
>>> decv = np.arange(1, 41)[::-1].reshape(5, 8)
>>> sbp1 = SubPlotElements(
...     curves=[CurveElements(values=incv, bins=bins, legend='increase')],
...     axnames=['x', 'y', 'spam'], ptype='2D')
>>> sbp2 = SubPlotElements(
...     curves=[CurveElements(values=decv, bins=bins, legend='decrease')],
...     axnames=['x', 'y', 'spam'], ptype='2D')
>>> sbp3 = SubPlotElements(
...     curves=[CurveElements(values=incv/decv, bins=bins, legend='i/d')],
...     axnames=['x', 'y', 'ratio'], ptype='2D')
>>> sbp3.attributes.logz = True
>>> pltnd = PlotTemplate(subplots=[sbp1, sbp2, sbp3], small_subplots=False)
>>> from valjean.javert import mpl
>>> mplplt = mpl.MplPlot(pltnd)
>>> fig, _ = mplplt.draw()
```

Strings as bins

It is possible to use strings as bins both in 1D and 2D plots. If strings are too long on x-axis they will be represented vertically.

```
>>> from collections import OrderedDict
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = [np.array(['spam', 'egg', 'bacon']),
...         np.array(['beer', 'wine', 'milk', 'tea with milk and sugar'])]
>>> axnames = ['x', 'y']
>>> v2d = np.arange(12).reshape(3, 4)
>>> v1d = np.arange(4)
>>> lsplts = []
>>> lsplts.append(SubPlotElements(
...     curves=[CurveElements(values=v2d, bins=bins, legend='Menus')],
...     axnames=['Meat', 'Drink', 'Associations'], ptype='2D'))
>>> lsplts.append(SubPlotElements(
...     curves=[CurveElements(
...         values=v1d, bins=bins[1:], legend='', index=1)],
...     axnames=['Drink', 'Quantity'], ptype='1D'))
>>> pltnd = PlotTemplate(subplots=lsplts)
>>> from valjean.javert import mpl
>>> mplplt = mpl.MplPlot(pltnd)
>>> fig, _ = mplplt.draw()
```

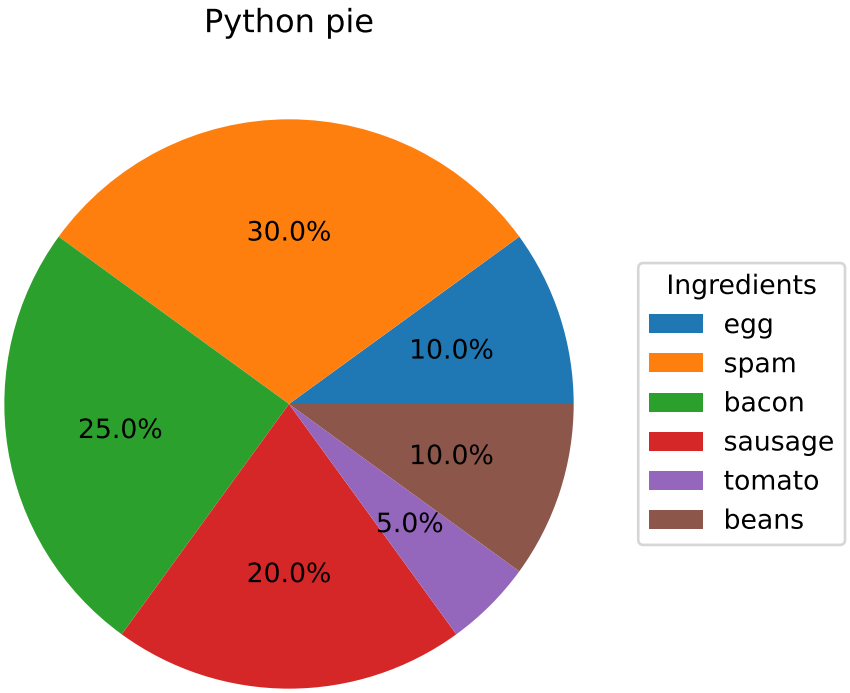

increase
decrease
i/d
x
y
spam
ratio


Menus
tea with milk and sugar
milk
wine
beer
Drink
spam
egg
bacon
Meat
10
8
6
4
2
0
Associations
Quantity
2
0
beer
wine
milk
tea with milk and sugar
Drink

Pie plot

Pie plots can be done if requested. *ptype* should be equal to *'pie'*. Note that the number of *axnames* still has to be N dim + 1, so 2. The first one is the title of the plot, the second one the title of the legend. If the second string is empty no title will be given to the legend.

```
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> ingredients = ['egg', 'spam', 'bacon', 'sausage', 'tomato', 'beans']
>>> proportions = [0.1, 0.3, 0.25, 0.2, 0.05, 0.1]
>>> curve = CurveElements(values=np.array(proportions),
...                       bins=[ingredients], legend='')
>>> sbplt = SubPlotElements(curves=[curve],
...                         axnames=('Python pie', 'Ingredients'),
...                         ptype='pie')
>>> pltpie = PlotTemplate(subplots=[sbplt])
>>> from valjean.javert import mpl
>>> mplplt = mpl.MplPlot(pltpie)
>>> fig, _ = mplplt.draw()
```



Bar plots

Bar plots can be used here with strings as labels in x-axis, like category plots but no errors are expected here. Two kinds of bar plots are available: side-by-side bars and stacked bars.

```
>>> import numpy as np
>>> from valjean.javert.templates import (PlotTemplate, CurveElements,
...                                       SubPlotElements)
>>> bins = [np.array(['spam', 'egg', 'bacon'])]
>>> data = [np.array([1, 3, 4]),  np.array([2, 4, 5]),
...         np.array([5, 3, 2]), np.array([2, 3, 1])]
>>> names = ['Terry', 'John' , 'Graham', 'Eric']
>>> lcurves = []
>>> for datum, name in zip(data, names):
...     lcurves.append(CurveElements(values=datum, bins=bins, legend=name))
>>> speb = SubPlotElements(curves=lcurves,
...                        axnames=['ingredient', 'quantity'], ptype='bar')
>>> spebs = SubPlotElements(curves=lcurves,
...                         axnames=['ingredient', 'quantity'],
...                         ptype='barstack')
>>> pltbar = PlotTemplate(subplots=[speb, spebs], small_subplots=False)
>>> from valjean.javert import mpl
>>> mplplt = mpl.MplPlot(pltbar)
>>> fig, _ = mplplt.draw()
```

Module API

class valjean.javert.mpl.**MplStyle**(*style=None*, *colors=None*, *mshape=None*, *mfill=None*, *legends=None*)

Class to store style characteristics.

__init__(*style=None*, *colors=None*, *mshape=None*, *mfill=None*, *legends=None*)

Initialisation of the style.

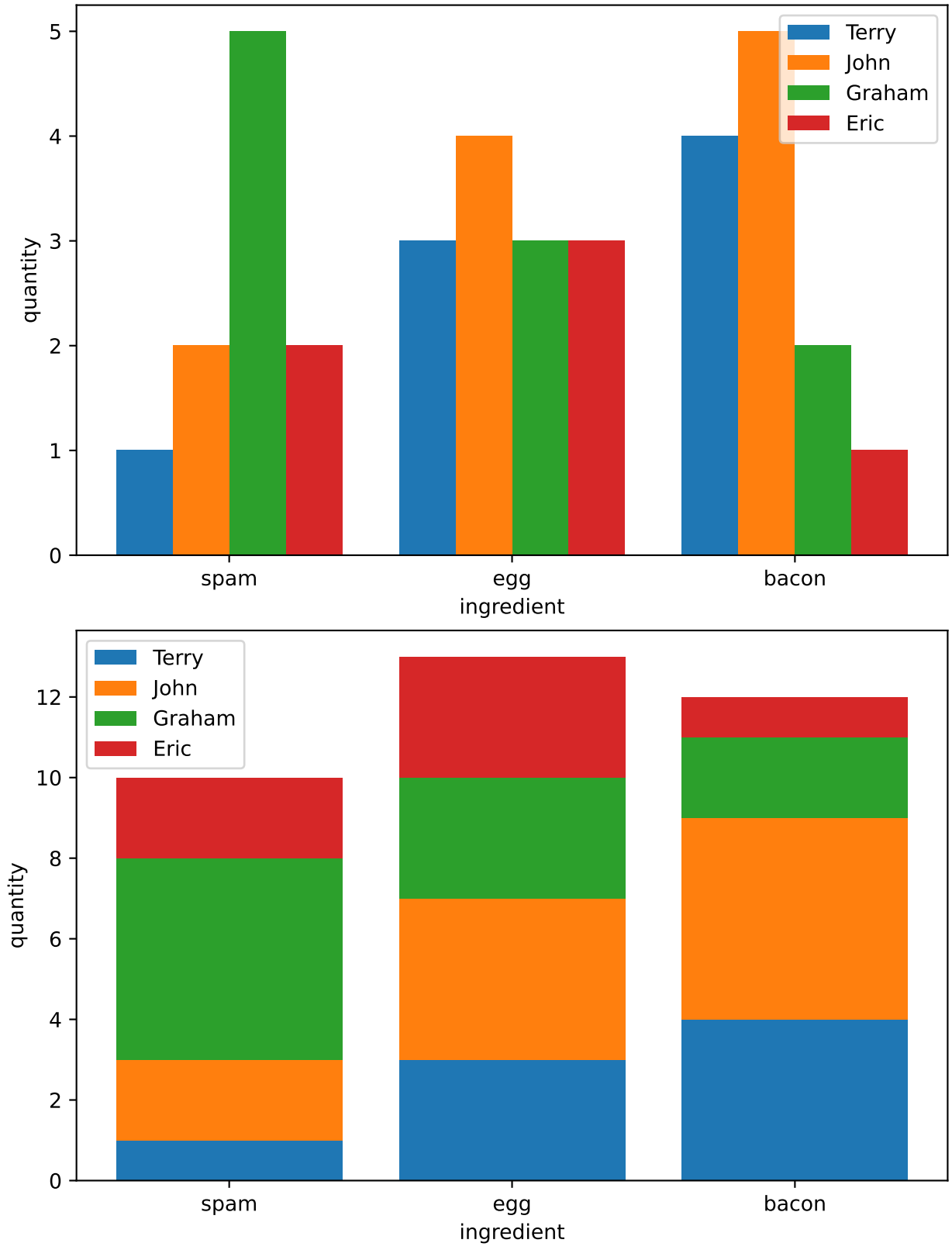
Parameters

- **style** (*str or None*) – style from matplotlib styles or from user one, if None `'default'` is used
- **colors** (*list(str) or None*) – colors from matplotlib colors, if None CN are used
- **mshape** (*list(str) or None*) – marker shapes from matplotlib markers, if None a default sequence has been determined
- **mfill** (*list(str) or None*) – marker fill, None will use an alternance of `'fill'` and `'none'`
- **legends** (*dict or None*) – keyword arguments from legend documentation to be passed to legend builder

An additional instance parameter is available and initialised in *MplPlot* thanks to *styles_sequence*, `fmts`. It builds the suite of styles of 1D curves from colors and markers.

styles_sequence(*indices*)

Define the 1D style suite to be used for the 1D plots.


Terry
John
Graham
Eric
quantity
ingredient
spam
egg
bacon
0
1
2
3
4
5
Terry
John
Graham
Eric
quantity
ingredient
spam
egg
bacon
0
2
4
6
8
10
12

Parameters
indices (*list(int)*) – list of the curves style index

Return type
dict

Returns
dictionary of curve styles indexed by curve index

exception valjean.javert.mpl.**MplPlotException**

Error raised if the plot cannot be made.

class valjean.javert.mpl.**MplPlot**(*data*, *, *style=None*)

Convert a *PlotTemplate* into a matplotlib plot.

__init__(*data*, *, *style=None*)

Construct a *MplPlot* from the given *PlotTemplate*.

Plots are initialized, drawn and finalized in $draw$. Depending on the requested type of the subplots, `ptype` a 1D or a 2D plot will be done. Internal classes are called to draw each kind of available plots.

Available types of plots are stored in the class variable `PTYPES`.

No plot for more than 2 dimensions are done.

Parameters

- **data** (PlotTemplate) – the data to convert
- **style** (MplStyle) – the style to be used in the plot

static figure_properties(*data*)

Define figures properties like figsize or the grid specifications.

Returns
dictionary of keyword arguments directly used by matplotlib

Return type
dict

initialize_figure()

Construct the figure and its subplots.

Return type
tuple(matplotlib.figure.Figure, list(matplotlib.axes.Axes))

finalize_figure(*splts*)

Finalize the figure.

If `suppress_xaxes` is required in **data**, tick labels and label of the x-axis will be deleted on all subplots except the last one.

If `suppress_legends` is required in **data**, legends will be deleted on all plots except the first one. If only one curve is represented in total legend is also deleted.

Parameters
splts (*list(matplotlib.axes.Axes)*) – the subplots

draw()

Draw the plot.

Return type
tuple(matplotlib.figure.Figure, list(matplotlib.axes.Axes))

save(*name='fig.png'*)
Save the plot under the given name.

Parameters
name (*str*) – name of the output file. Expected extensions: png, pdf, svg, eps.

class valjean.javert.mpl.**_MplLegend**(*handle*, *label*, *index*)
Class to store the legend content.

__init__(*handle*, *label*, *index*)
Initialisation of *_MplLegend*.

Parameters
- **handle** (matplotlib.pyplot.errorbar or tuple (matplotlib.pyplot.errorbar)) – curve to be stored (if the curve needs to be drawn in twice a tuple should be given)
- **label** (*str*) – the curve label
- **index** (*int*) – curve index, to identify its style

class valjean.javert.mpl.**_MplPlot1D**(*data*, *style=None*)
Convert a *PlotTemplate* into a matplotlib plot.

__init__(*data*, *style=None*)
Construct a *_MplPlot1D* from the given *SubPlotElements*.

Parameters
- **data** (SubPlotElements) – the data to convert.
- **style** (MplStyle) – style to be used in the subplot

draw(*_fig*, *splt*, **_args*, *fmts*, ***_kwargs*)
Draw method.

Parameters
- **_fig** (*matplotlib.figure.Figure*) – the current figure
- **splt** (*matplotlib.axes.Axes*) – the current subplot
- **fmts** (*dict*) – curves styles

ierror_plot(*splt*, *curve*, *data_fmt*)
Draw the plot with error bars on the plot (update the plot)

Parameters
- **splt** (*matplotlib.axes.Axes*) – the current subplot
- **curve** (CurveElements) – data to plot
- **data_fmt** (*tuple(str)*) – format of the curve (color, marker shape, marker filling)

error_plots(*splt*, *fmts*)

Plot errorbar plot (update the pyplot instance) and build the legend.

Parameters

- **splt** (*matplotlib.axes.Axes*) – the current subplot
- **fmts** (*dict*) – curves styles

_build_legend(*splt*)

Build the legends and add them to the figures.

An automatic number of columns is calculated, depending on the number of curves to be plotted on the subplot. It has been decided to add a new columns each 5 curves.

Parameters

splt (*matplotlib.axes.Axes*) – the current subplot

customize_plots(*splt*)

Customize plots (scale, limits and lines).

class valjean.javert.mpl.**_MplPlot2D**(*data*, *style=None*)

Convert a *SubPlotElements* into a 2D plot.

__init__(*data*, *style=None*)

Construct a *_MplPlot2D* from the given *SubPlotElements*.

Parameters

- **data** (SubPlotElements) – the data to convert.
- **style** (MplStyle) – style to be used in the subplot

draw(*fig*, *splt*, **_args*, ***_kwargs*)

Draw method.

Remark: if the quantity represented is required in logarithmic scale,so the z-axis in logarithmic scale, it has to be done at the histogram declaration and not in the customization step.

Parameters

- **fig** (*matplotlib.figure.Figure*) – the current figure
- **splt** (*matplotlib.axes.Axes*) – the current subplot

static broadcast_bin_centers(*curve*)

Calculate bin centers if edges are given and broadcast all bins: build the (x, y) grid for all bins.

Parameters

curve (CurveElements) – data to plot

Return type

list(numpy.ndarray), list(numpy.ndarray), list(numpy.ndarray)

Returns

grid, bin edges, bin centers

itwod_plot(*fig*, *splt*, *curve*, *axnames*, *norm*)

Draw the 2D distribution on the ith subplot.

Parameters

- **fig** (*matplotlib.figure.Figure*) – the current figure
- **splt** (*matplotlib.axes.Axes*) – the current subplot
- **curve** (CurveElements) – data to plot
- **iplot** (*int*) – number of the subplot
- **norm** (function from matplotlib.colors) – function corresponding to the chosen normalisation and scale (linear or logarithmic)

twod_plots(*fig*, *splt*)

Build 2D plots.

Parameters

- **fig** (*matplotlib.figure.Figure*) – the current figure
- **splt** (*matplotlib.axes.Axes*) – the current subplot

customize_plots(*splt*)

Customize plots (scale and limit).

Parameters

- **fig** (*matplotlib.figure.Figure*) – the current figure
- **splt** (*matplotlib.axes.Axes*) – the current subplot

class valjean.javert.mpl.**_MplPie**(*data*, *_style=None*)

Convert a *PlotTemplate* into a matplotlib pie chart.

__init__(*data*, *_style=None*)

Construct a *_MplPie* from the given *SubPlotElements*.

Parameters

- **data** (SubPlotElements) – the data to convert.
- **style** (MplStyle) – style to be used in the subplot

draw(*_fig*, *splt*, **_args*, ***_kwargs*)

Draw method.

Parameters

- **_fig** (*matplotlib.figure.Figure*) – the current figure
- **splt** (*matplotlib.axes.Axes*) – the current subplot

pie_chart(*splt*)

Prepare the pie chart.

Parameters
splt (*matplotlib.axes.Axes*) – the current subplot

class valjean.javert.mpl.**_MplBar**(*data*, *_style=None*)

Convert a *PlotTemplate* into a matplotlib bar plot.

__init__(*data*, *_style=None*)

Construct a *_MplBar* from the given *SubPlotElements*.

Parameters

- **data** (SubPlotElements) – the data to convert.

- **style** (MplStyle) – style to be used in the subplot

draw(*_fig*, *splt*, **_args*, ***_kwargs*)
Draw method.

Parameters

- **_fig** (*matplotlib.figure.Figure*) – the current figure
- **splt** (*matplotlib.axes.Axes*) – the current subplot

bar_plot(*splt*)
Prepare the bar plot.

class valjean.javert.mpl.**_MplBarStack**(*data*, *_style=None*)
Convert a *PlotTemplate* into a matplotlib bar plot.

__init__(*data*, *_style=None*)
Construct a *_MplBarStack* from the given *SubPlotElements*.

Parameters

- **data** (SubPlotElements) – the data to convert.
- **style** (MplStyle) – style to be used in the subplot

draw(*_fig*, *splt*, **_args*, ***_kwargs*)
Draw method.

Parameters

- **_fig** (*matplotlib.figure.Figure*) – the current figure
- **splt** (*matplotlib.axes.Axes*) – the current subplot

bar_plot(*splt*)
Prepare the bar plot.

5.6.7 test_report – Classes modelling a test report

This module contains a simple class that models a test report. A report consists of a number of sections, and each section contains: * a title; * some optional introductory text; * optionally, other test reports and some test results.

This module also contains a task that creates a report from a list of test-evaluation tasks.

class valjean.javert.test_report.**TestReport**(***, *title*, *text=''*, *content=None*)
This class models a test report. It is essentially a rose tree (i.e. a tree with an arbitrary number of children at each node) with nodes annotated with a title and some introductory text. *TestResult* objects play the role of tree leaves.

__init__(***, *title*, *text=''*, *content=None*)
Create a report.

Parameters

- **title** (*str*) – the report title.
- **text** (*str*) – some introductory text.
- **content** (*list(*TestReport *or* TestResult*)*) – a list of *TestReport* or *TestResult* objects.

class valjean.javert.test_report.**TestReportTask**(*name*, *, *make_report*, *eval_tasks*, *kwargs=None*)

Task class that creates a *TestReport* from a function that classifies test results into report sections.

__init__(*name*, *, *make_report*, *eval_tasks*, *kwargs=None*)

Instantiate a *TestReportTask* from a list of *EvalTestTask* objects. The *make_report* argument is expected to be a function taking one argument, which is a dictionary associating the names of the tasks in *eval_tasks* to the results of their execution (in the sense of the `'result'` key of the associated environment section). For tasks that evaluate tests (*EvalTestTask*), the result typically consists of a list of *TestResult* objects. The *make_report* function must return a *TestReport* object.

Parameters

- **name** (*str*) – the name of this task.
- **make_report** – a function taking a single argument and returning a *TestReport*.
- **eval_tasks** (*list(*Task*)*) – a list of tasks, typically *EvalTestTask*.
- **kwargs** (*dict or None*) – a dictionary of kwargs that will be passed to the *make_report* function.

5.6.8 test_external – Class to import external test

External test used to represent user formatted results in the report.

This test has been designed in order to offer the possibility to define and run user-defined tests outside valjean and include their results in a report, along with the native valjean tests.

The user is responsible for defining and representing the test results. The representation must be done with templates (see *templates* module).

For example, the simplest case is a custom plot to represent a test. In that case, if the path to the image is known a *TextTemplate* can be created and given to the test:

```
>>> txtemp = TextTemplate("And this represents Javert\n\n"
...                       ".. image:: ../../doc/src/images/javert.jpg\n\n")
>>> textr = TestExternal(txtemp, name='', success=True).evaluate()
>>> print(bool(textr))
True
```

class valjean.javert.test_external.**TestResultExternal**(*test*)

Result from a external test.

__bool__()

This test returns the return value specified in the test initialisation.

class valjean.javert.test_external.**TestExternal**(**templates*, *name*, *description=''*, *labels=None*, *success=True*)

Class building a test that will return the boolean value specified at initialisation.

__init__(**templates*, *name*, *description=''*, *labels=None*, *success=True*)

Initialize the *TestExternal* object with the templates to represent in the report, a name, a description of the test (may be long).

External test are performed by the user who only gives its representation as templates and its return value. The return value can be an arbitrary value (a test that always returns *True* or *False*) for example.

Parameters

- **templates** (TextTemplate, PlotTemplate, TableTemplate) – templates representing the external test
- **name** (*str*) – name of the test
- **description** (*str*) – description of the test expected with context
- **labels** (*dict*) – labels for test classification
- **success** (*bool*) – returned value by *TestResultExternal*

evaluate()

Evaluate the test (do nothing in this case except building the TestResult).

Return type
TestResultExternal

5.6.9 rst – Convert items to rst format

This module provides the classes to convert test results to reStructuredText format. reStructuredText is a text markup language meant for easy human consumption. The reStructuredText primer provides a good introduction to its syntax.

class valjean.javert.rst.**Rst**(*representation*, *, *n_workers=None*)

Class to convert *TestResult* objects into reStructuredText format.

__init__(*representation*, *, *n_workers=None*)

Initialize a class instance with the given representation.

Parameters
representation (Representation) – A representation.

clear()

Clear the content of *self*.

format_report(*, *report*, *author*, *version*)

Format a report.

Most of the work is actually done in the *format_report_rec* method.

Parameters
report (TestReport) – the report to format.

Returns
the formatted report.

Return type
FormattedRst

format_report_rec(*, *report*, *tree*)

Recursively format report sections and populate the `tree_dict` and `text_dict` attributes.

Report sections, as represented by the *TestReport* class, are essentially trees, with *TestResult* objects playing the role of the tree leaves and *TestReport* objects playing the role of the tree nodes. This method performs a recursive, pre-order, depth-first traversal of the tree and fills the `tree_dict` and `text_dict` dictionaries with the information that is necessary to instantiate the final *FormattedRst* object.

Given a report object, this method constructs a dictionary key by concatenating in a tuple the titles of the parent reports and the present report. The tuple containing the parent reports' titles is recursively passed as the *tree* argument (and it is initialized to the empty tuple in the call from *format_report*). For instance, consider the following simplified report structure:

```
>>> from valjean.javert.test_report import TestReport
>>> subsub1 = TestReport(title='SubSub1')
>>> subsub2 = TestReport(title='SubSub2')
>>> sub1 = TestReport(title='Sub1', content=[subsub1, subsub2])
>>> sub2 = TestReport(title='Sub2')
>>> main = TestReport(title='Main', content=[sub1, sub2],
...                   text='Si six scies scient six cyprès, '
...                        'six-cent-six scies scient '
...                        'six-cent-six cyprès.')
```

We instantiate an *Rst* object and we format the report:

```
>>> import valjean.javert.representation as rpr
>>> from valjean.javert.rst import Rst
>>> rst = Rst(rpr.Representation(rpr.FullRepresenter()))
>>> formatted_report = rst.format_report(report=main, author='Me',
...                                      version='0.1')
```

Let us look at the dictionary keys:

```
>>> print(sorted(rst.text_dict.keys()))
[(), ('Sub1',), ('Sub1', 'SubSub1'), ('Sub1', 'SubSub2'), ('Sub2',)]
```

The empty tuple, `()`, is associated to the main report. The other keys represent subreports. The length of the tuple corresponds to the depth of the nested report; for instance, the `'SubSub2'` section appears in the `('Sub1', 'SubSub2')` tuple with length 2, because `'SubSub2'` is nested twice (Main/Sub1/SubSub2).

The values of the `text_dict` dictionary are lists of strings representing the text of the given section. For instance, here is what the main section looks like:

```
>>> print('\n'.join(rst.text_dict[()]))
Main
====

Si six scies scient six cyprès, six-cent-six scies scient six-cent-six cyprès.
```

The `tree_dict` dictionary associates a tuple representing a report section to the list of tuples that represent the subsections:

```
>>> print(rst.tree_dict[()])
[('Sub1',), ('Sub2',)]
>>> print(rst.tree_dict[('Sub1',)])
[('Sub1', 'SubSub1'), ('Sub1', 'SubSub2')]
```

If a section does not have any subsection, the corresponding tuple does not appear in the dictionary:

```
>>> ('Sub2',) in rst.tree_dict
False
```

The `tree_dict` and `text_dict` dictionaries contain most of the information required by *FormattedRst* to write out the report in the form of a reStructuredText file tree (see *write*).

format_section(*section*, ***, *depth*)

Format a report section.

Parameters
report (TestReport) – the report section to format.

Returns
the formatted report section.

Return type
str

format_result(*result*)

Format one test result.

Parameters
result (TestResult) – A *TestResult*.

Returns
the formatted test result.

Return type
str

class valjean.javert.rst.**RstFormatter**

Class that dispatches the task of formatting templates as reStructuredText. The concrete formatting is handled by separate classes (*RstTable*...).

header(*name*, *depth*)

Produce the header for formatting a *TestResult*.

Parameters

- **name** (*str*) – A header name
- **depth** (*int*) – the depth of this header, 0 being the top.

Returns
the test header, for inclusion in a reST document.

Return type
str

text(*text*)

Format some text.

Parameters
text (*str*) – some text to include.

Returns
the text, in rst format.

Return type
str

static anchor(*fingerprint*)
Format an anchor with the given fingerprint.

static format_tabletemplate(*table*)
Format a *TableTemplate*.

Parameters
table (TableTemplate) – A table.

Returns
the reST table.

Return type
RstTable

static format_plottemplate(*plot*)
Format a *PlotTemplate*.

Parameters
plot (PlotTemplate) – A plot.

Returns
the formatted plot

Return type
RstPlot

static format_texttemplate(*text*)
Format a *TextTemplate*.

Parameters
text (TextTemplate) – A text with its highlight positions

Returns
the formatted text (text itself)

Return type
str

class valjean.javert.rst.**RstTable**(*table*, *num_fmt='{:11.6g}'*)
Convert a *TableTemplate* into a reStructuredText table.

__init__(*table*, *num_fmt='{:11.6g}'*)
Construct an *RstTable* from the given *TableTemplate*.

Parameters

- **table** (TableTemplate) – The table to convert.
- **num_fmt** (*str*) – A `format` string to specify how numerical table entries should be represented. The default value for this option is `'{:11.6g}'`.

__str__()
Yield the table, as a reST string. This is probably the method that you want to call.

classmethod tabularize(*headers*, *rows*, *, *indent=0*)
Transform a list of headers and a list of rows into a nice reST table.

The *headers* argument must be a list of strings. The *rows* argument represents the table rows, as a list of lists of strings. Each sublist represents a table row, and it must have the same length as *headers*, or terrible things will happen.

The column widths are automatically computed to accommodate the largest template in each column. Smaller templates are automatically right-justified.

```
>>> headers = ['name', 'quest', 'favourite colour']
>>> rows = [['Lancelot', 'to seek the Holy Grail', 'blue'],
...         ['Galahad', 'to seek the Holy Grail', 'yellow']]
>>> table = RstTable.tabularize(headers, rows)
>>> print(table)
========  ====================  ================
  name            quest          favourite colour
========  ====================  ================
Lancelot  to seek the Holy Grail              blue
 Galahad  to seek the Holy Grail            yellow
========  ====================  ================
```

You can also indent the table by a given amount of spaces with the *indent* keyword argument:

```
>>> table = RstTable.tabularize(headers, rows, indent=4)
>>> print(table)
    ========  ====================  ================
      name            quest          favourite colour
    ========  ====================  ================
    Lancelot  to seek the Holy Grail              blue
     Galahad  to seek the Holy Grail            yellow
    ========  ====================  ================
```

Parameters

- **headers** (*list(str)*) – The table headers.
- **rows** (*list(list(str))*) – The table rows.

Returns

The reST table, as a string.

static transpose(*columns*)

Given a matrix as a list of columns, yield the matrix rows. (Equivalently, if the matrix is given as a list of rows, yield the columns). For instance, consider the following list:

```
>>> matrix = [(11, 21, 31), (12, 22, 32), (13, 23, 33)]
>>> for column in matrix:
...     print(' '.join(str(elem) for elem in column))
11 21 31
12 22 32
13 23 33
```

If the tuples are interpreted as columns, *matrix* represents the following matrix:

$$\begin{pmatrix} 11 & 12 & 13 \\ 21 & 22 & 23 \\ 31 & 32 & 33 \end{pmatrix}$$

Transposing it yields:

```
>>> transposed = RstTable.transpose(matrix)
>>> for row in transposed:
...     print(' '.join(str(elem) for elem in row))
11 12 13
21 22 23
31 32 33
```

Note: The matrix elements returned by *transpose* are actually 0-dimensional numpy.ndarray objects. For the purpose of their further manipulation this is of little consequence, as they transparently support most numerical operations.

Parameters
columns (*list(list) or list(tuple) or list(numpy.ndarray)*) – An iterable yielding columns

Raises
ValueError – if the columns do not have the same length.

Returns
a generator for the rows of the transposed matrix.

classmethod format_columns(*columns*, *highlights*, *num_fmt*)

Transform a bunch of columns (containing arbitrary data: floats, bools, ints, strings...) into an iterable of lists of (optionally highlighted) strings representing the table **rows**.

Example:

```
>>> cols = [('European swallow', 'African swallow'),
...         (27.7, 35.1),
...         ('km/h', 'km/h')]
>>> highs = [[False, False], [False, True], [False, True]]
>>> for row in RstTable.format_columns(cols, highs, '{:13.8f}'):
...     print(row)
['European swallow', '  27.70000000', 'km/h']
['African swallow', ':hl:`35.10000000`', ':hl:`km/h`']
```

Parameters

- **columns** – An iterable yielding columns. The table columns may contain any kind of data; if the data type is numeric, it will be formatted according to the *num_fmt* argument; other types will just get stringified (str).
- **highlights** – An iterable yielding a collection of bools for each column. The *j*-th element of the *i*-th iteration decides whether the element in the *j*-th column of the *i*-th row should be highlighted.
- **num_fmt** – The format string to be used for numerical types.

Returns
a generator yielding lists of strings, representing the table rows.

static compute_column_widths(*headers*, *rows*)

Compute the width of the columns that are necessary to accommodate all the headers and table rows.

```
>>> headers = ['swallow sub-species', 'airspeed', 'laden']
>>> rows = [['European', '27.7 km/h', 'no'],
...         ['European', '22.0 km/h', 'yes'],
...         ['African', '35.1 km/h', 'no'],
...         ['African', '26.2 km/h', 'yes']]
>>> RstTable.compute_column_widths(headers, rows)
[19, 9, 5]
>>> [len('swallow sub-species'), len('27.7 km/h'), len('laden')]
[19, 9, 5]
```

Parameters

- **headers** (*list(str)*) – A list of column headers.
- **rows** (*list(list(str))*) – A list of table rows.

Returns

a list of integers indicating how wide each columns must be to accommodate all the table elements and the headers.

Return type

list(int)

classmethod highlight(*val*, *flag*)

Wrap *val* in highlight reST markers if flag is *True*.

```
>>> RstTable.highlight('dis', False)
'dis'
>>> RstTable.highlight('DAT', True)
':hl:`DAT`'
```

classmethod concat_rows(*widths*, *rows*, *just*='>')

Concatenate the given rows and justify them according to the *just* parameter (see the Format Specification Mini-Language). The columns widths (for justification) must be provided using the *width* argument.

Parameters

- **widths** (*list(int)*) – The list of columns widths.
- **rows** (*list(list(str))*) – The list of rows; each row must be a list of strings.
- **just** (*str*) – A format character for the justification (usually one of '<', '^', '>').

Returns

the concatenated, justified rows.

Return type

str

class valjean.javert.rst.**RstPlot**(*plot*)

This class models a plot in an reStructuredText document. It converts a *PlotTemplate* object into an *MplPlot*, and it provides the `.. image::` directive to include in the .rst file.

__init__(*plot*)

__str__()
Return str(self).

filename()
Make up a(n almost) unique filename for this plot.

Returns
the filename from fingerprint (png format)

Return type
str

class valjean.javert.rst.**RstText**(*text*)
Construct an *RstText* from the given *TextTemplate*.

__init__(*text*)

__str__()
Return str(self).

class valjean.javert.rst.**FormattedRst**(*, *author*, *title*, *version*, *tree_dict*, *text_dict*, *plots*, *n_workers=None*)
This class represents a formatted rst document tree, which typically consists of an index file and of several sections.

__init__(*, *author*, *title*, *version*, *tree_dict*, *text_dict*, *plots*, *n_workers=None*)
Create a *FormattedRst* object. The *author*, *title* and *version* arguments are expected to be strings and are self-explanatory.

The *tree_dict* and *text_dict* arguments must be dictionaries. The *tree_dict* dictionary represents the tree structure of the reStructuredText document, and the *text_dict* represent the contents of each section. The report sections, which appear as keys in both dictionaries, are expected to be tuples of strings, with each string representing an additional layer in the document hierarchy. The section contents (the values of *text_dict*) are expected to be lists of strings to be written to disk.

Finally, the *plots* argument is a list of the plots referenced by the text sections. The plots will be written to disk with filenames of the form `plot_`*`fingerprint`*`.png`, where *fingerprint* is the plot fingerprint.

Parameters

- **author** (*str*) – the author of this report.
- **title** (*str*) – the title of this report.
- **version** (*str*) – the version number for this report.
- **tree_dict** (*dict*) – dictionary mapping tuples of sections to lists of tuples of sections.
- **text_dict** (*dict*) – dictionary mapping tuples of sections to lists of strings.
- **plots** (*list(MplPlot)*) – list of plots to be written to disk.
- **n_workers** (*int or None*) – number of subprocesses to use to write out the plots in parallel. If *None* is given, write the plots in sequential mode.

write(*path*)

Write the text files and the plots into the directory specified by *path*.

Parameters
path (*str or pathlib.Path*) – path to the directory that will be written to. It is OK if the directory does not exist.

static tree_to_path(*, *base*, *tree*)

Convert a tree to a file path.

Parameters

- **base** (*pathlib.Path*) – the base path for all subtrees.
- **tree** (*tuple(str)*) – a sequence of tree nodes, starting from the tree root.

toc(*toc_title*, *subtrees*)

Build an rst table of contents.

Parameters
toc_title (*str*) – the title for the table of contents (e.g. `'Contents'`).

setup(*path*)

Set up the output directory for *write*.

Parameters
path (*pathlib.Path*) – the path to the directory.

static configure(*resource*, *dest*, *formatting=True*, ***kwargs*)

Copy a package resource to the specified destination, optionally formatting the resource content using `str.format`.

For more information about resources, see pkg_resources.

Parameters

- **resource** (*str*) – the name of the resource.
- **dest** (*pathlib.Path*) – the destination path.
- **formatting** (*bool*) – whether formatting should be applied.
- **kwargs** (*dict*) – any additional keyword arguments will be passed to the formatting.

class valjean.javert.rst.**RstTestReportTask**(*name*, *, *report_task*, *representation*, *author*, *version*, *deps=None*, *soft_deps=None*)

Task class that transforms a list of tests into a test report. *TestResult* objects in the environment.

classmethod from_tasks(*name*, *, *make_report*, *eval_tasks*, *representation*, *author*, *version*, *kwargs=None*, *deps=None*, *soft_deps=None*)

Construct an *RstTestReportTask* from a list of test evaluation tasks and a function to classify test results and put them in test reports.

__init__(*name*, *, *report_task*, *representation*, *author*, *version*, *deps=None*, *soft_deps=None*)

Initialize the task with a function, a tuple of arguments and a dictionary of kwargs.

Parameters

- **name** (*str*) – The name of the task.
- **func** – A function to be executed.
- **args** (*tuple*) – A tuple of positional arguments to *func*, or *None* if none are required.
- **kwargs** (*dict*) – A dictionary of keyword arguments to *func*, or *None* if none are required.
- **env_kwarg** (*str*) – The name of the keyword argument that will be used to pass the environment to the function, or *None* if the environment should not be passed.
- **config_kwarg** (*str*) – The name of the keyword argument that will be used to pass the config to the function, or *None* if the config should not be passed.
- **deps** (None or collection of *Task* objects.) – If this task depends on other tasks (and valjean cannot automatically discover this), pass them (as a list) to the *deps* parameter.
- **soft_deps** (None or collection of *Task* objects.) – If this task has a soft dependency on other tasks (and valjean cannot automatically discover this), pass them (as a list) to the *soft_deps* parameter.

5.6.10 `verbosity` – Verbosity levels

Verbosity module.

4 levels of verbosity exist to build tables and plots: SILENT, SUMMARY, INTERMEDIATE and FULL_DETAILS.

class valjean.javert.verbosity.**Verbosity**(*value*)

Verbosity enum.

Six levels are currently available: SILENT, SUMMARY, DEFAULT, INTERMEDIATE, FULL_DETAILS, DEVELOPMENT.

6 Changelog

v0.1 (2018-03-30)

- Initial *valjean* release.
- List of available modules:
  - *cosette*
  - *eponine*
  - *cambronne*

7

TODO list

Todo: Write documentation for the executable.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/checkouts/v0.10
line 4.)

Todo: Implement `svn` and `cvs` checkout; `copy` checkout (i.e. copy a directory from somewhere) may also be useful.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/cosette/code.py:docstring of valjean.cosette.code, line 9.)

Todo: Implement `autoconf/configure/make` builds.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/cosette/code.py:docstring of valjean.cosette.code, line 21.)

Todo: Simplify this example by changing back OrderedDict in dict when python 3.5 won't be anymore supported.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/eponine/apollo3/hdf5_reader.py:docstring of valjean.eponine.apollo3.hdf5_reader.dict_to_list, line 22.)

Todo: Think about units. Possibility: using a units package from scipy.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/eponine/dataset.py:docstring of valjean.eponine.dataset.Dataset, line 5.)

Todo: How to deal with bins of N values (= center of bins)

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/eponine/dataset.py:docstring of valjean.eponine.dataset.Dataset, line 9.)

Todo:

Not a standalone code, needs inputs.
To be tested in a more general context.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/eponine/tripoli4/common.py:docstring of valjean.eponine.tripoli4.common,
line 7.)

Todo: Adjoint results: for the moment only IFP is really parsed. Grammar has already more or less adapted to welcome Wielandt method that will have the same kind of outputs (renaming as adjoint_res for example). No key is set for the moment to specify the method, it can be obtained from the response function itself. Adjoint criticality editions are only done for IFP, this may change when the same will be available for Wielandt. Some renaming can also be needed.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/eponine/tripoli4/grammar.py:docstring of valjean.eponine.tripoli4.grammar,
line 179.)

Todo: Change absolute imports in relative ones when main will be moved to *cambronne*.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/eponine/tripoli4/parse.py:docstring of valjean.eponine.tripoli4.parse, line
10.)

Todo: Document the parameters…

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/gavroche/diagnostics/metadata.py:docstring of valjean.gavroche.diagnostics.metadata.TestMetadata, line 3.)

Todo: Possible improvement: turn *Representer* into an *ABC*; loop over the classes in *eponine* that inherit from *:class:`~.TestResult* and add *@abstractmethod* methods in *Representer*. This way, if a new *TestResult* is added to *eponine*, it will no longer be possible to instantiate any of the classes that derive from *Representer*, pointing to the fact that the code in this module needs to be extended to handle the new class. This is better than silently falling back to some default do-nothing implementation, which may lead to bugs.

(The original entry is located in /home/docs/checkouts/readthedocs.org/user_builds/valjean/envs/v0.10.0/lib/
packages/valjean/javert/representation.py:docstring of valjean.javert.representation, line
4.)

8 Indices and tables

- genindex
- modindex
- search

valjean release number: 0.10.0

Python Module Index

V

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z