Javelin Documentation

Release 0.1.0

Ross Whitfield

Nov 19, 2020

Contents

Contents:

CHAPTER 1

About

Javelin is a moderisation of DISCUS written in the Python programming language

CHAPTER 2

Installation

2.1 Install the latest release

- Update once we have a release

2.2 Requirements

- Python >= 3.5
- NumPy >= 1.10
- h5py >= 2.5
- pandas >= 0.17
- xarray >= 0.7
- periodictable >= 1.4
- cython >= 0.23
- pytables >= 3.2

Optional:

- ASE (to use the ase atoms structure) >= 3.14

2.3 Development

A development environment can easily be set up with either conda or PyPI

2.4 Using Conda

```
conda env create
source activate javelin
python setup.py install
```

2.5 Using PyPI

2.6 Tests

The unit tests can be run with pytest

2.6.1 Install with conda

```
conda install pytest
```

2.6.2 Install with PyPI

```
pip install pytest
```

CHAPTER 3

Tutorials

3.1 Working with DISCUS

Javelin is inspired by DISCUS Until javelin has become feature equivalent to DISCUS for disordered strcuture creation DISCUS can still be used to create structures. While javelin can be used to calculate the diffuse scattering and compare to experimental data.

Javelin can read in discus structure files simply by:

```
>>> from javelin.io import read_stru
>>> structure = read_stru("tests/data/pzn2.stru")
Found a = 4.06, b = 4.06, c = 4.06, alpha = 90.0, beta = 90.0, gamma = 90.0
Read in these atoms:
Nb     80
O     375
Pb    125
Zn     45
Name: symbol, dtype: int64
>>> print(structure)
Structure(Nb80O375Pb125Zn45, a=4.06, b=4.06, c=4.06, alpha=90.0, beta=90.0, gamma=90.
↪0)
```

From here it's easy to calculate the scattering.

```
>>> from javelin.io import read_stru  # doctest: +SKIP
>>> structure = read_stru("../../tests/data/pzn2.stru")  # doctest: +SKIP
Found a = 4.06, b = 4.06, c = 4.06, alpha = 90.0, beta = 90.0, gamma = 90.0
Read in these atoms:
Nb     80
O     375
Pb    125
Zn     45
Name: symbol, dtype: int64
>>> from javelin.fourier import Fourier  # doctest: +SKIP
```

(continues on next page)

(continued from previous page)

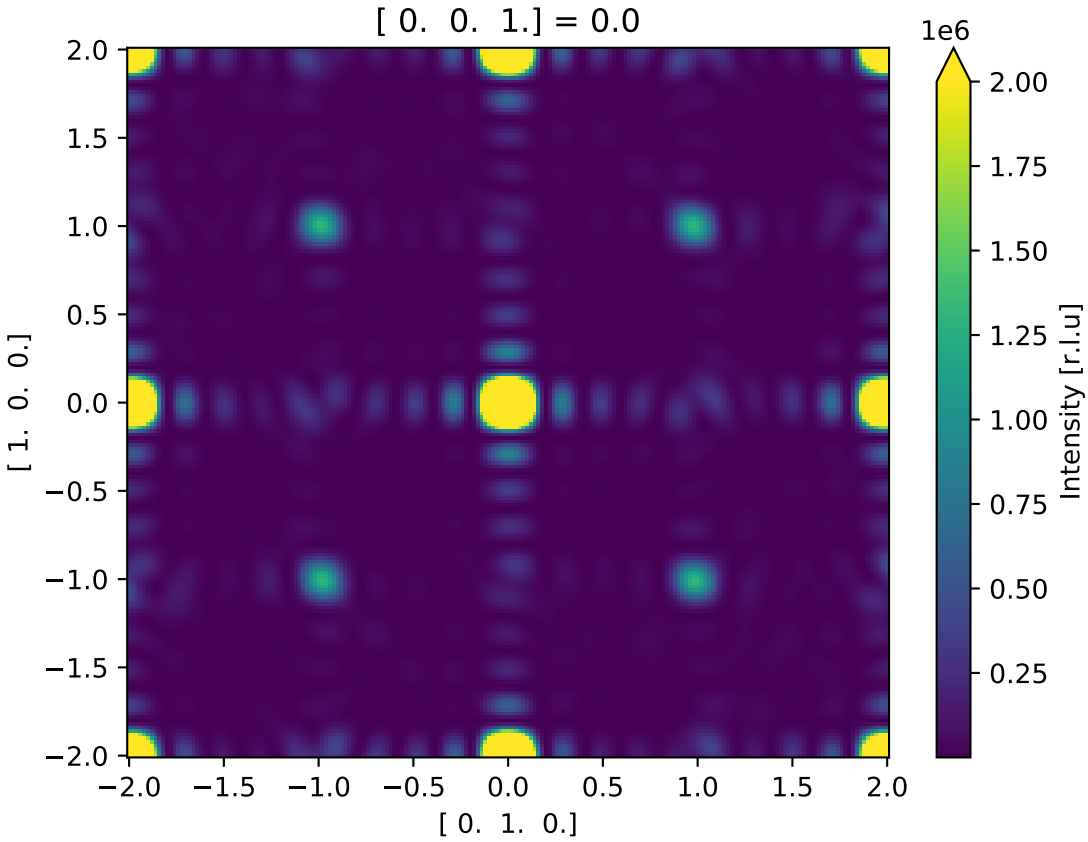
```
>>> fourier = Fourier()  # doctest: +SKIP
>>> fourier.grid.r1 = -2, 2  # doctest: +SKIP
>>> fourier.grid.r2 = -2, 2  # doctest: +SKIP
>>> fourier.grid.bins = 201, 201  # doctest: +SKIP
>>> print(fourier)  # doctest: +SKIP
Structure         : Structure(Nb80O375Pb125Zn45, a=4.06, b=4.06, c=4.06, alpha=90.0,␣
↪beta=90.0, gamma=90.0)
Radiation         : neutron
Fourier volume    : complete crystal
Aver. subtraction : False
<BLANKLINE>
Reciprocal layer  :
lower left  corner :     [-2. -2.  0.]
lower right corner :     [ 2. -2.  0.]
upper left  corner :     [-2.  2.  0.]
top   left  corner :     [-2. -2.  1.]
<BLANKLINE>
hor. increment     :     [ 0.02  0.    0.  ]
vert. increment    :     [ 0.    0.02  0.  ]
top   increment    :     [ 0.  0.  1.]
<BLANKLINE>
# of points        :     201 x 201 x 1
>>> results = fourier.calc(structure)  # doctest: +SKIP
Working on atom number 8 Total atoms: 375
Working on atom number 30 Total atoms: 45
Working on atom number 41 Total atoms: 80
Working on atom number 82 Total atoms: 125
>>> results.plot(vmax=2e6)  # doctest: +SKIP
<matplotlib.collections.QuadMesh object at ...>
```

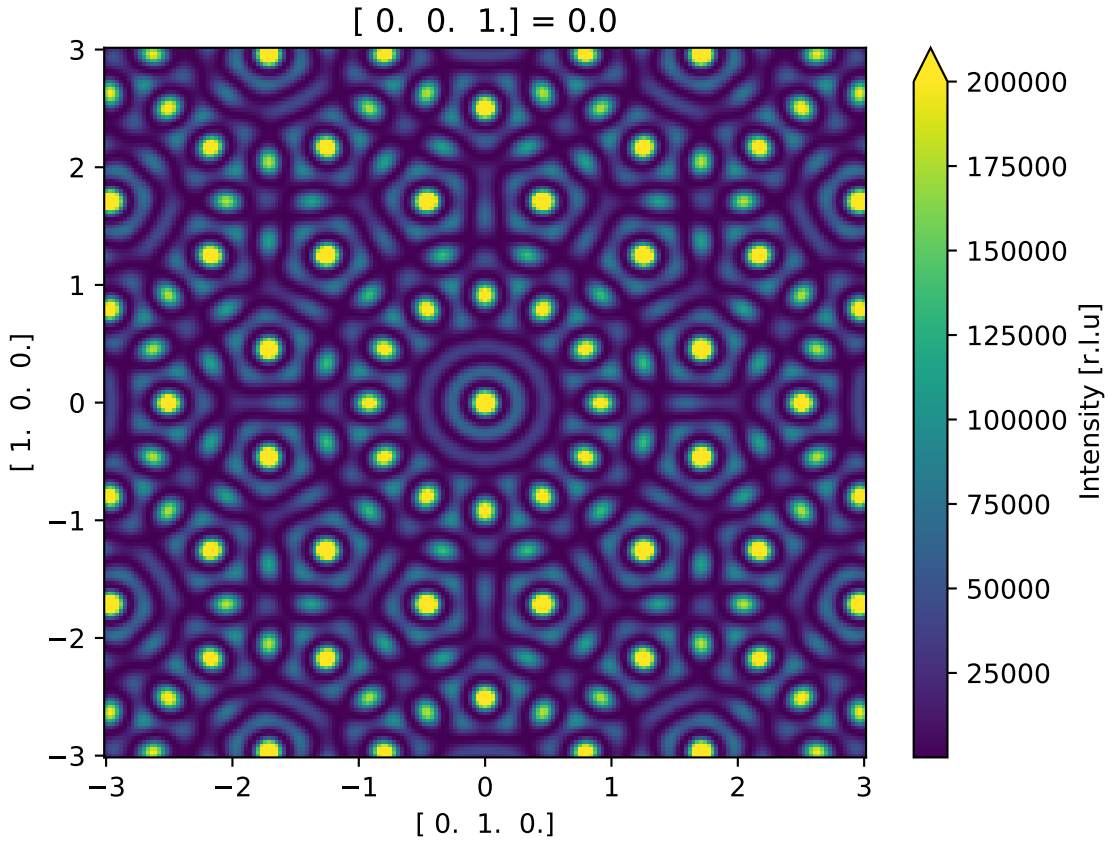
3.2 Working with ASE

The Atomic Simulation Environment (**ASE**) is a set of tools and Python modules for setting up, manipulating, running, visualizing and analyzing atomistic simulations.

Using the `ase.build.nanotube()` as an example of using calculating the scattering directly from an `ase.Atoms` object.

```
>>> from javelin.fourier import Fourier  # doctest: +SKIP
>>> from ase.build import nanotube  # doctest: +SKIP
>>> cnt = nanotube(6, 0, length=4)  # doctest: +SKIP
>>> print(cnt)  # doctest: +SKIP
Atoms(symbols='C96', pbc=[False, False, True], cell=[0.0, 0.0, 17.04])
>>> cnt_four = Fourier()  # doctest: +SKIP
>>> cnt_four.grid.bins = 201, 201  # doctest: +SKIP
>>> cnt_four.grid.r1 = -3, 3  # doctest: +SKIP
>>> cnt_four.grid.r2 = -3, 3  # doctest: +SKIP
>>> results = cnt_four.calc(cnt)  # doctest: +SKIP
Working on atom number 6 Total atoms: 96
>>> results.plot(vmax=2e5)  # doctest: +SKIP
<matplotlib.collections.QuadMesh object at ...>
```


[0. 0. 1.] = 0.0
1e6
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
Intensity [r.l.u]
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
[1. 0. 0.]
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
[0. 1. 0.]


[0. 0. 1.] = 0.0
[1. 0. 0.]
[0. 1. 0.]
Intensity [r.l.u]
200000
175000
150000
125000
100000
75000
50000
25000

3.2.1 Convert structure from ASE to javelin

A `ase.Atoms` can be converted to *`javelin.structure.Structure`* simply by initializing the javelin structure from the ASE atoms.

```
>>> print(cnt)  # doctest: +SKIP
Atoms(symbols='C96', pbc=[False, False, True], cell=[0.0, 0.0, 17.04])
>>> type(cnt)  # doctest: +SKIP
<class 'ase.atoms.Atoms'>
>>> from javelin.structure import Structure  # doctest: +SKIP
>>> javelin_cnt = Structure(cnt)  # doctest: +SKIP
>>> print(javelin_cnt)  # doctest: +SKIP
Structure(C96, a=1, b=1, c=17.04, alpha=90.0, beta=90.0, gamma=90.0)
>>> type(javelin_cnt)  # doctest: +SKIP
<class 'javelin.structure.Structure'>
```

3.2.2 Convert structure from javelin to ASE

To convert *`javelin.structure.Structure`* to `ase.Atoms` you can use *`javelin.structure.Structure.to_ase()`*.

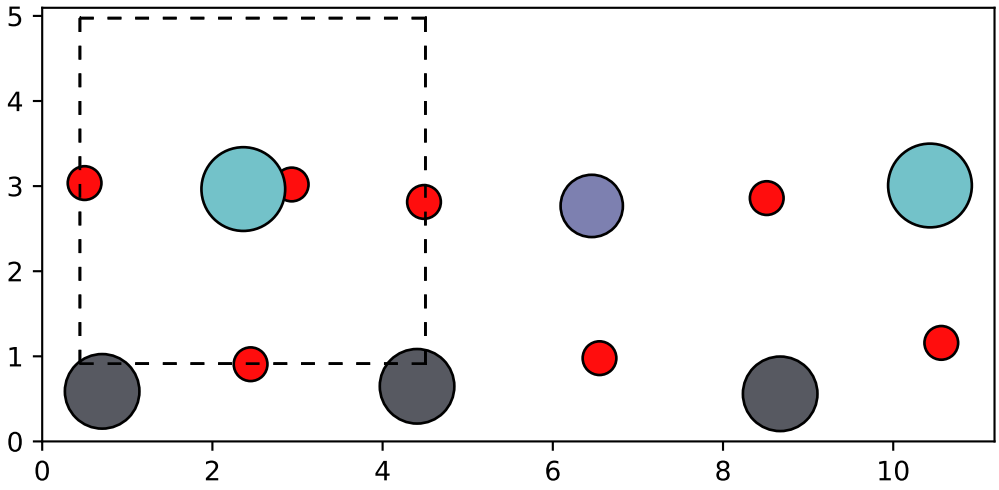
```
>>> from javelin.io import read_stru  # doctest: +SKIP
>>> pzn = read_stru('../../tests/data/pzn.stru')  # doctest: +SKIP
Found a = 4.06, b = 4.06, c = 4.06, alpha = 90.0, beta = 90.0, gamma = 90.0
Read in these atoms:
O     9
Pb    3
Nb    2
Zn    1
Name: symbol, dtype: int64
>>> print(pzn)  # doctest: +SKIP
Structure(O9Pb3Nb2Zn1, a=4.06, b=4.06, c=4.06, alpha=90.0, beta=90.0, gamma=90.0)
>>> type(pzn)  # doctest: +SKIP
<class 'javelin.structure.Structure'>
>>> pzn_ase = pzn.to_ase()  # doctest: +SKIP
>>> print(pzn_ase)  # doctest: +SKIP
Atoms(symbols='PbNbO3PbZnO3PbNbO3', pbc=False, cell=[4.06, 4.06, 4.06])
>>> type(pzn_ase)  # doctest: +SKIP
<class 'ase.atoms.Atoms'>
```

3.2.3 Visualization with ASE

The *`javelin.structure.Structure`* has enough api compatibility with `ase.Atoms` that it can be used with some of ASE's visulization tools.

An example using ASE's matplotlib interface.

```
>>> from javelin.io import read_stru  # doctest: +SKIP
>>> from ase.visualize.plot import plot_atoms  # doctest: +SKIP
>>> pzn = read_stru('../../tests/data/pzn.stru')  # doctest: +SKIP
>>> print(pzn)  # doctest: +SKIP
Structure(O9Pb3Nb2Zn1, a=4.06, b=4.06, c=4.06, alpha=90.0, beta=90.0, gamma=90.0)
>>> plot_atoms(pzn, radii=0.3)  # doctest: +SKIP
<matplotlib.axes._subplots.AxesSubplot object at ...>
```


0
1
2
3
4
5
0
2
4
6
8
10

To use all of ASE's visualization tools, such as `ase.gui`, VMD, Avogadro, or ParaView, first *Convert structure from javelin to ASE*.

```
>>> from ase.visualize import view  # doctest: +SKIP
>>> pzn_ase = pzn.to_ase()  # doctest: +SKIP
>>> view(pzn_ase)  # doctest: +SKIP
>>> view(pzn_ase, viewer='vmd')  # doctest: +SKIP
>>> view(pzn_ase, viewer='avogadro')  # doctest: +SKIP
>>> view(pzn_ase, viewer='paraview')  # doctest: +SKIP
```

3.2.4 File IO

`ase.io` has extensive support for file-formats that can be utilized by javelin. For example reading in '.cif' files using `ase.io.read()`

```
>>> from javelin.structure import Structure  # doctest: +SKIP
>>> from ase.io import read  # doctest: +SKIP
>>> graphite = Structure(read('tests/data/graphite.cif'))  # doctest: +SKIP
>>> print(graphite)  # doctest: +SKIP
Structure(C4, a=2.456, b=2.456, c=6.696, alpha=90.0, beta=90.0, gamma=119.
↪99999999999999)
>>> type(graphite)  # doctest: +SKIP
<class 'javelin.structure.Structure'>
>>> PbTe = Structure(read('tests/data/PbTe.cif'))  # doctest: +SKIP
>>> print(PbTe)  # doctest: +SKIP
Structure(Pb4Te4, a=6.461, b=6.461, c=6.461, alpha=90.0, beta=90.0, gamma=90.0)
>>> type(PbTe)  # doctest: +SKIP
<class 'javelin.structure.Structure'>
```

ASE can also be used to write file to many file-formats using `ase.io.write()`

```
>>> from ase.io import write  # doctest: +SKIP
>>> write('output.xyz', graphite.to_ase())  # doctest: +SKIP
>>> write('output.png', graphite.to_ase())  # doctest: +SKIP
```

3.3 Working with DiffPy

DiffPy is a free and open source software project to provide python software for diffraction analysis and the study of the atomic structure of materials.

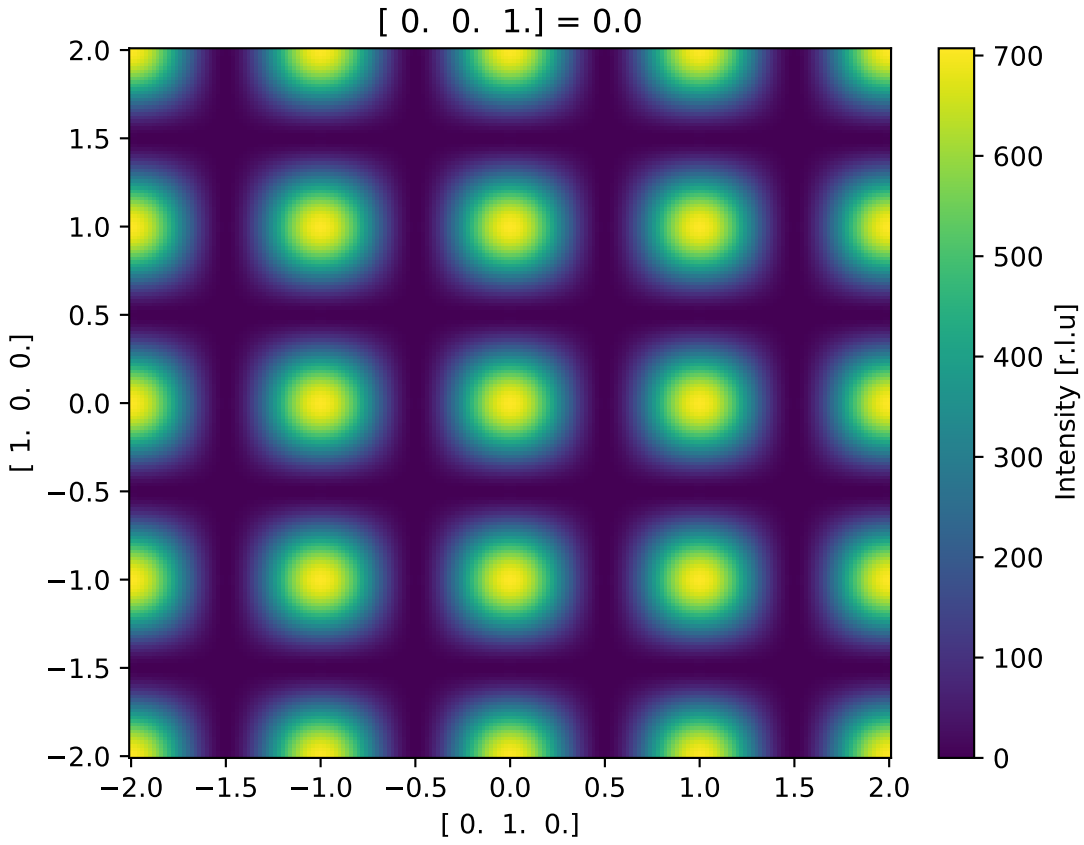
The scattering can be calculated from a `diffpy.Structure.structure.Structure` directly.

```
>>> from diffpy.Structure import Structure, Lattice, Atom  # doctest: +SKIP
>>> from javelin.fourier import Fourier  # doctest: +SKIP
>>> stru = Structure([Atom('C',[0,0,0]), Atom('C',[1,0,0]),
...                   Atom('C',[0,1,0]), Atom('C',[1,1,0])],
...                  lattice=Lattice(1,1,1,90,90,120))  # doctest: +SKIP
>>> print(stru)  # doctest: +SKIP
lattice=Lattice(a=1, b=1, c=1, alpha=90, beta=90, gamma=120)
C     0.000000 0.000000 0.000000 1.0000
C     1.000000 0.000000 0.000000 1.0000
C     0.000000 1.000000 0.000000 1.0000
C     1.000000 1.000000 0.000000 1.0000
>>> type(stru)  # doctest: +SKIP
```

(continues on next page)

(continued from previous page)

```
<class 'diffpy.Structure.structure.Structure'>
>>> four = Fourier()  # doctest: +SKIP
>>> four.grid.bins = 201, 201  # doctest: +SKIP
>>> four.grid.r1 = -2, 2  # doctest: +SKIP
>>> four.grid.r2 = -2, 2  # doctest: +SKIP
>>> results = four.calc(stru)  # doctest: +SKIP
Working on atom number 6 Total atoms: 4
>>> results.plot()  # doctest: +SKIP
<matplotlib.collections.QuadMesh object at ...>
```



3.3.1 Convert structure from diffpy to javelin

A `diffpy.Structure.structure.Structure` can be converted to *javelin.structure.Structure* simply by initializing the javelin structure from the diffpy structure.

```
>>> from diffpy.Structure import Structure as diffpy_Structure, Lattice, Atom  #␣
↪doctest: +SKIP
>>> from javelin.structure import Structure  # doctest: +SKIP
>>> stru = diffpy_Structure([Atom('C',[0,0,0]),Atom('C',[1,1,1])], lattice=Lattice(1,
↪1,1,90,90,120))  # doctest: +SKIP
>>> print(stru)  # doctest: +SKIP
lattice=Lattice(a=1, b=1, c=1, alpha=90, beta=90, gamma=120)
```

(continues on next page)

(continued from previous page)

```
C    0.000000 0.000000 0.000000 1.0000
C    1.000000 1.000000 1.000000 1.0000
>>> type(stru)  # doctest: +SKIP
<class 'diffpy.Structure.structure.Structure'>
>>> javelin_stru = Structure(stru)  # doctest: +SKIP
>>> print(javelin_stru)  # doctest: +SKIP
Structure(C2, a=1.0, b=1.0, c=1.0, alpha=90.0, beta=90.0, gamma=120.0)
>>> type(javelin_stru)  # doctest: +SKIP
<class 'javelin.structure.Structure'>
```

3.3.2 Convert structure from javelin to diffpy

```
>>> type(javelin_stru)  # doctest: +SKIP
<class 'javelin.structure.Structure'>
>>> diffpy_stru = diffpy_Structure([Atom(e, x) for e, x in zip(javelin_stru.element,␣
↪javelin_stru.xyz)],
...                                lattice=Lattice(*javelin_stru.unitcell.cell))  #␣
↪doctest: +SKIP
>>> print(diffpy_stru)  # doctest: +SKIP
lattice=Lattice(a=1, b=1, c=1, alpha=90, beta=90, gamma=120)
C    0.000000 0.000000 0.000000 1.0000
C    1.000000 1.000000 1.000000 1.0000
>>> type(diffpy_stru)  # doctest: +SKIP
<class 'diffpy.Structure.structure.Structure'>
```

3.3.3 File IO

DiffPy file loaders can be utilized by javelin.

```
>>> from diffpy.Structure.Parsers import getParser  # doctest: +SKIP
>>> from javelin.structure import Structure  # doctest: +SKIP
>>> p = getParser('auto')  # doctest: +SKIP
>>> graphite = Structure(p.parseFile('tests/data/graphite.cif'))  # doctest: +SKIP
>>> print(graphite)  # doctest: +SKIP
Structure(C4, a=2.456, b=2.456, c=6.696, alpha=90.0, beta=90.0, gamma=120.0)
>>> type(graphite)  # doctest: +SKIP
<class 'javelin.structure.Structure'>
>>> pzn = Structure(p.parseFile('tests/data/pzn.stru'))  # doctest: +SKIP
>>> print(pzn)  # doctest: +SKIP
Structure(O9Pb3Nb2Zn1, a=12.18, b=4.06, c=4.06, alpha=90.0, beta=90.0, gamma=90.0)
>>> type(pzn)  # doctest: +SKIP
<class 'javelin.structure.Structure'>
```

3.4 Neighbors definition in Javelin

The neighbors above are created in javelin by (starting at 12 o'clock and working clockwise):

```
>>> from javelin.neighborlist import NeighborList
>>> nl = NeighborList([[0, 0, 0, 2, 0],
...                    [0, 1, 0, 0, 0],
...                    [0, 0, 1, 0, 0],
...                    [0, 0, 2,-1, 0],
...                    [0, 1,-2,-1, 0]])
>>> nl
NeighborList([[ 0  0  0  2  0]
              [ 0  1  0  0  0]
              [ 0  0  1  0  0]
              [ 0  0  2 -1  0]
              [ 0  1 -2 -1  0]])
>>> print(nl)
      |     site      |    vector
index | origin target |   i   j   k
    0 |      0      0 |   0   2   0
    1 |      0      1 |   0   0   0
    2 |      0      0 |   1   0   0
    3 |      0      0 |   2  -1   0
    4 |      0      1 |  -2  -1   0
```

3.5 Monte Carlo simulations

3.5.1 Ising Model examples

Creating chemical short-range order

Negative correlation

```
>>> import numpy as np
>>> import xarray as xr
>>> from javelin.structure import Structure
>>> from javelin.energies import IsingEnergy
>>> from javelin.modifier import SwapOccupancy
>>> from javelin.fourier import Fourier
>>> from javelin.mc import MC
>>> n=128
>>> structure = Structure(symbols=np.random.choice(['Na','Cl'],n**2), positions=[(0.,␣
↪0., 0.)]*n**2, unitcell=5)
>>> structure.reindex([n,n,1,1])
>>> e1 = IsingEnergy(11,17,J=0.5)
>>> nl = structure.get_neighbors()[0,-1]
>>> mc = MC()
>>> mc.add_modifier(SwapOccupancy(0))
>>> mc.temperature = 1
>>> mc.cycles = 50
>>> mc.add_target(nl, e1)
>>> out = mc.run(structure)  # doctest: +SKIP
>>> f=Fourier()
>>> f.grid.bins = 200, 200, 1
>>> f.grid.r1 = -3,3
>>> f.grid.r2 = -3,3
>>> f.lots = 8,8,1
```

(continues on next page)

(continued from previous page)

```
>>> f.number_of_lots = 256
>>> f.average = True
>>> results=f.calc(out)  # doctest: +SKIP
>>> # plot
>>> fig, axs = plt.subplots(2, 1, figsize=(6.4,9.6))  # doctest: +SKIP
>>> xr.DataArray.from_series(out.atoms.Z).plot(ax=axs[0])  # doctest: +SKIP
>>> results.plot(ax=axs[1])  # doctest: +SKIP
```

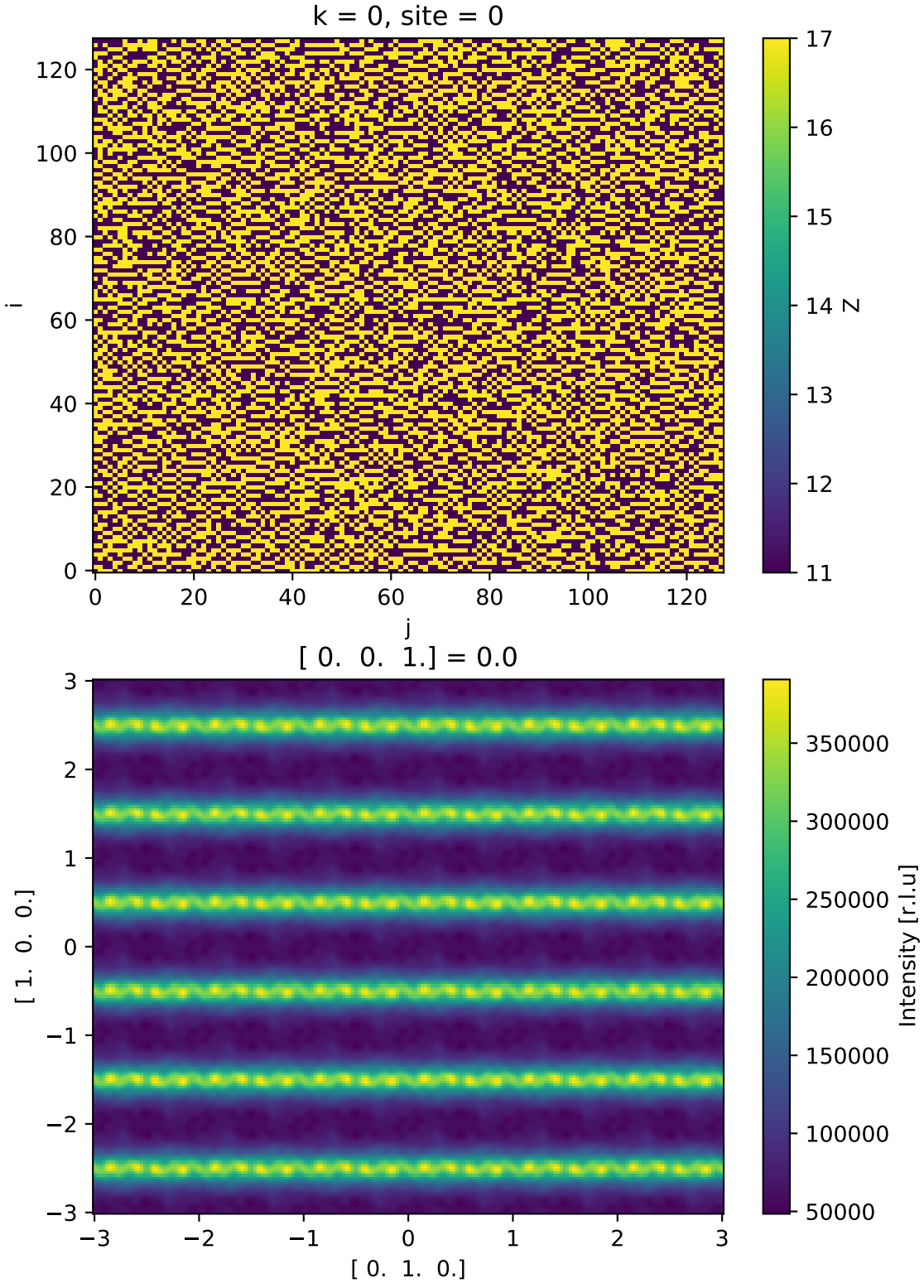
Positive correlation

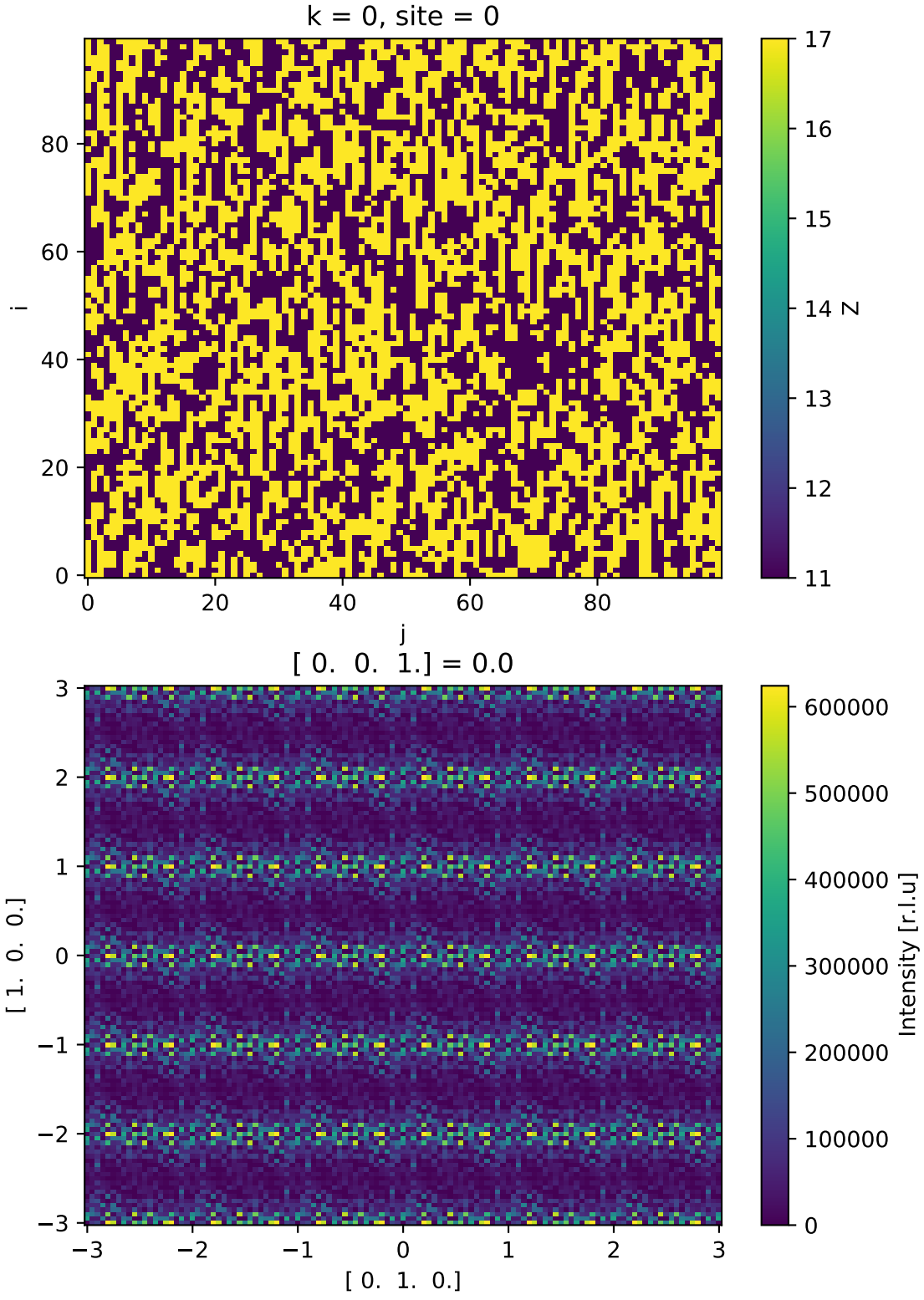
```
>>> import numpy as np
>>> import xarray as xr
>>> from javelin.structure import Structure
>>> from javelin.energies import IsingEnergy
>>> from javelin.modifier import SwapOccupancy
>>> from javelin.fourier import Fourier
>>> from javelin.mc import MC
>>>
>>> n=100
>>> structure = Structure(symbols=np.random.choice(['Na','Cl'],n**2), positions=[(0.,␣
↪0., 0.)]*n**2, unitcell=5)
>>> structure.reindex([n,n,1,1])
>>> e1 = IsingEnergy(11,17,J=-0.5)
>>> nl = structure.get_neighbors()[0,-1]
>>> mc = MC()
>>> mc.add_modifier(SwapOccupancy(0))
>>> mc.temperature = 1
>>> mc.cycles = 50
>>> mc.add_target(nl, e1)
>>> out = mc.run(structure)  # doctest: +SKIP
>>> f=Fourier()
>>> f.grid.bins = 121,121,1
>>> f.grid.r1 = -3,3
>>> f.grid.r2 = -3,3
>>> f.average = True
>>> results=f.calc(out)  # doctest: +SKIP
>>> # plot
>>> fig, axs = plt.subplots(2, 1, figsize=(6.4,9.6))  # doctest: +SKIP
>>> xr.DataArray.from_series(out.atoms.Z).plot(ax=axs[0])  # doctest: +SKIP
>>> results.plot(ax=axs[1])  # doctest: +SKIP
```

Getting a desired correlation

```
>>> import numpy as np
>>> import xarray as xr
>>> from javelin.structure import Structure
>>> from javelin.energies import IsingEnergy
>>> from javelin.modifier import SwapOccupancy
>>> from javelin.fourier import Fourier
>>> from javelin.mc import MC
>>>
>>> n=100
>>> structure = Structure(symbols=np.random.choice(['Na','Cl'],n**2), positions=[(0.,␣
↪0., 0.)]*n**2, unitcell=5)
```

(continues on next page)


k = 0, site = 0
i
j
Z
[0. 0. 1.] = 0.0
[1. 0. 0.]
[0. 1. 0.]
Intensity [r.l.u]


k = 0, site = 0
i
j
N
[0. 0. 1.] = 0.0
[1. 0. 0.]
[0. 1. 0.]
Intensity [r.l.u]

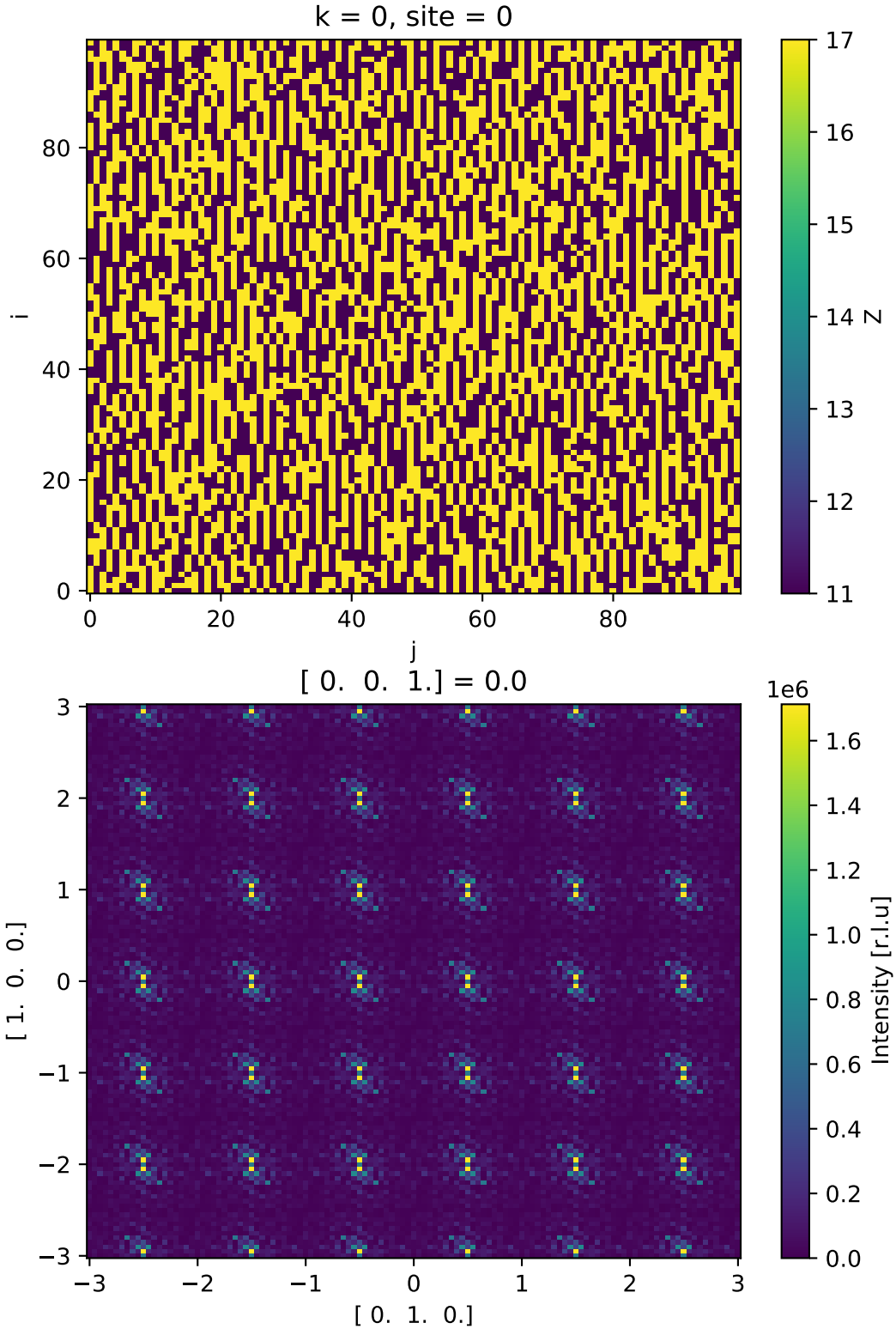
(continued from previous page)

```
>>> structure.reindex([n,n,1,1])
>>> e1 = IsingEnergy(11,17,desired_correlation=0.5)
>>> nl1 = structure.get_neighbors()[0,-1]
>>> e2 = IsingEnergy(11,17,desired_correlation=0)
>>> nl2 = structure.get_neighbors(minD=2.99,maxD=3.01)[0,-1]
>>> e3 = IsingEnergy(11,17,desired_correlation=-0.5)
>>> nl3 = structure.get_neighbors()[1,-2]
>>> mc = MC()
>>> mc.add_modifier(SwapOccupancy(0))
>>> mc.temperature = 1
>>> mc.cycles = 50
>>> mc.add_target(nl1, e1)
>>> mc.add_target(nl2, e2)
>>> mc.add_target(nl3, e3)
>>> out = mc.run(structure)  # doctest: +SKIP
>>> f=Fourier()
>>> f.grid.bins = 121,121,1
>>> f.grid.r1 = -3,3
>>> f.grid.r2 = -3,3
>>> f.average = True
>>> results=f.calc(out)  # doctest: +SKIP
>>> # plot
>>> fig, axs = plt.subplots(2, 1, figsize=(6.4,9.6))  # doctest: +SKIP
>>> xr.DataArray.from_series(out.atoms.Z).plot(ax=axs[0])  # doctest: +SKIP
>>> results.plot(ax=axs[1])  # doctest: +SKIP
```

3.5.2 Size-effect

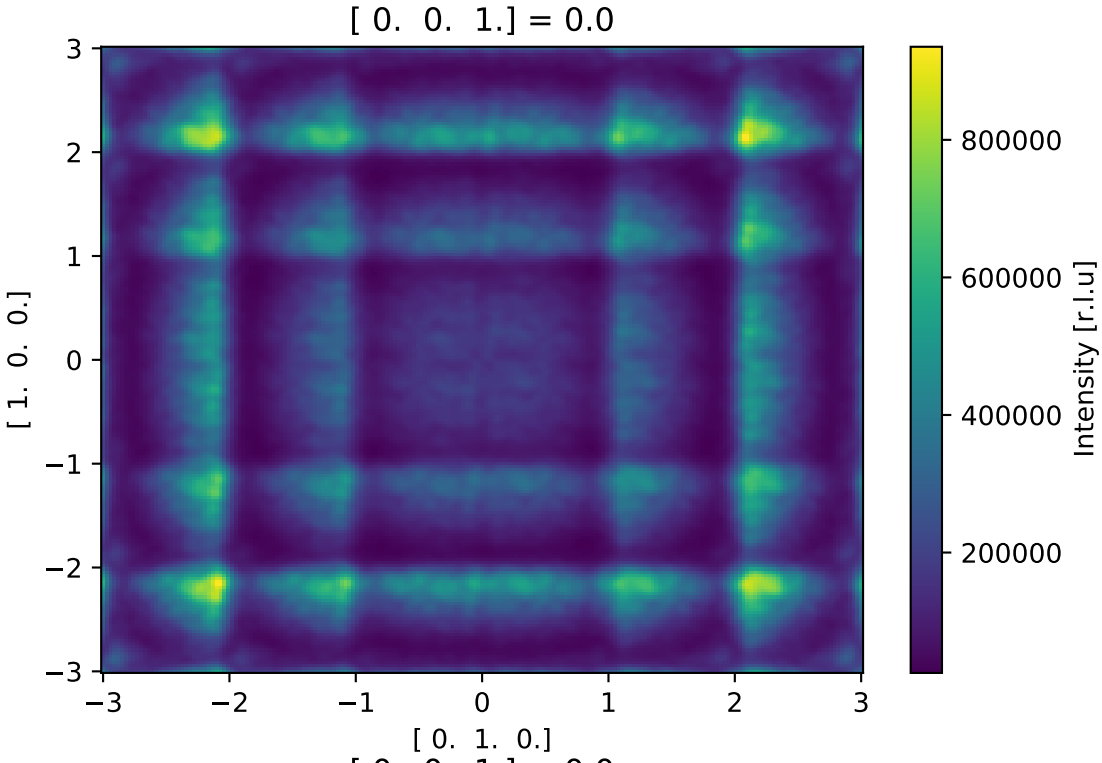
```
>>> import numpy as np
>>> from javelin.structure import Structure
>>> from javelin.energies import LennardJonesEnergy
>>> from javelin.modifier import SetDisplacementNormalXYZ
>>> from javelin.fourier import Fourier
>>> from javelin.mc import MC
>>> n = 128
>>>
>>> x = np.random.normal(0, 0.01, size=n**2)
>>> y = np.random.normal(0, 0.01, size=n**2)
>>> z = np.zeros(n**2)
>>>
>>> structure = Structure(symbols=np.random.choice(['Na','Cl'],n**2), positions=np.
↪vstack((x,y,z)).T, unitcell=5)
>>> structure.reindex([n,n,1,1])
>>>
>>> nl = structure.get_neighbors()[0,1,-2,-1]
>>>
>>> e1 = LennardJonesEnergy(1, 1.05, atom_type1=11, atom_type2=11)
>>> e2 = LennardJonesEnergy(1, 1.0,  atom_type1=11, atom_type2=17)
>>> e3 = LennardJonesEnergy(1, 0.95, atom_type1=17, atom_type2=17)
>>>
>>> mc = MC()
>>> mc.add_modifier(SetDisplacementNormalXYZ(0, 0, 0.02, 0, 0.02, 0, 0))
>>> mc.temperature = 0.001
>>> mc.cycles = 50
>>> mc.add_target(nl, e1)
```

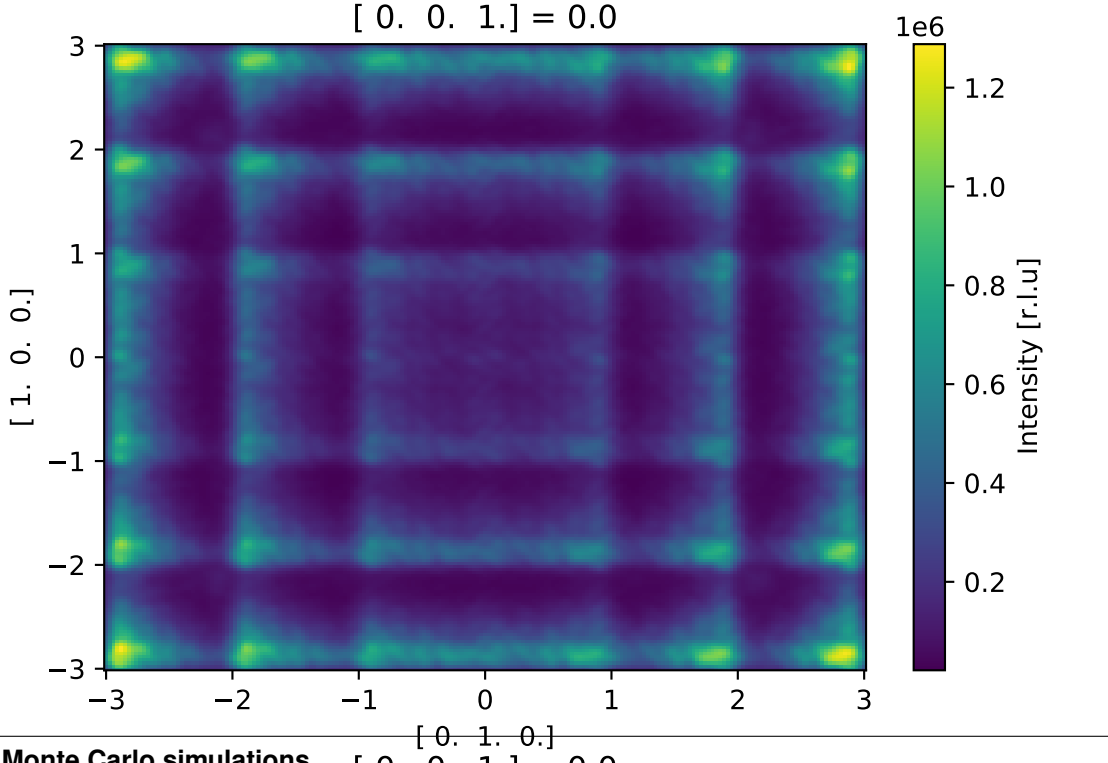
(continues on next page)

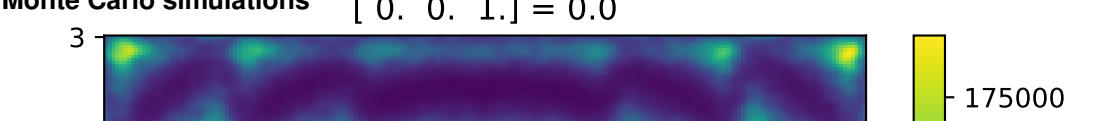

k = 0, site = 0
i
j
Z
[0. 0. 1.] = 0.0
[1. 0. 0.]
[0. 1. 0.]
1e6
Intensity [r.l.u]

(continued from previous page)

```
>>> mc.add_target(nl, e2)
>>> mc.add_target(nl, e3)
>>>
>>> out = mc.run(structure)  # doctest: +SKIP
>>>
>>> f=Fourier()  # doctest: +SKIP
>>> f.grid.bins = (201, 201, 1)  # doctest: +SKIP
>>> f.grid.r1 = -3, 3  # doctest: +SKIP
>>> f.grid.r2 = -3, 3  # doctest: +SKIP
>>> f.lots = 8, 8, 1  # doctest: +SKIP
>>> f.number_of_lots = 256  # doctest: +SKIP
>>> f.average = True  # doctest: +SKIP
>>>
>>> results1=f.calc(out)  # doctest: +SKIP
>>>
>>> out.replace_atom(11,42)  # doctest: +SKIP
>>> out.replace_atom(17,11)  # doctest: +SKIP
>>> out.replace_atom(42,17)  # doctest: +SKIP
>>> results2=f.calc(out)  # doctest: +SKIP
>>>
>>> out.replace_atom(17,11)  # doctest: +SKIP
>>> results3=f.calc(out)  # doctest: +SKIP
>>> fig, axs = plt.subplots(3, 1, figsize=(6.4,14.4))  # doctest: +SKIP
>>> results1.plot(ax=axs[0])  # doctest: +SKIP
>>> results2.plot(ax=axs[1])  # doctest: +SKIP
>>> results3.plot(ax=axs[2])  # doctest: +SKIP
```


[0. 0. 1.] = 0.0
[1. 0. 0.]
[0. 1. 0.]
Intensity [r.l.u]
800000
600000
400000
200000


[0. 0. 1.] = 0.0
1e6
[1. 0. 0.]
[0. 1. 0.]
Intensity [r.l.u]
1.2
1.0
0.8
0.6
0.4
0.2


[0. 0. 1.] = 0.0
175000

CHAPTER 4

Modules

List of javelin modules:

4.1 energies

Custom energies can be created by inheriting from *javelin.energies.Energy* and overriding the *evaluate* method. The evaluate method must have the identical signature and this gives you access to the origin and neighbor sites atom types and xyz's along with the neighbor vector.

For example

```
class MyEnergy(Energy):
    def __init__(self, E=-1):
        self.E = E
    def evaluate(self,
                 a1, x1, y1, z1,
                 a2, x2, y2, z2,
                 neighbor_x, neighbor_y, neighbor_z):
        return self.E
```

This is slower than using compile classes by about a factor of 10. If you are using IPython or Jupyter notebooks you can use Cython magic to compile your own energies. You need load the Cython magic first `%load_ext Cython`. Then *for example*

```
%%cython
from javelin.energies cimport Energy
cdef class MyCythonEnergy(Energy):
    cdef double E
    def __init__(self, double E=-1):
        self.E = E
    cpdef double evaluate(self,
                          int a1, double x1, double y1, double z1,
                          int a2, double x2, double y2, double z2,
```

(continues on next page)

(continued from previous page)

```
                            Py_ssize_t neighbor_x, Py_ssize_t neighbor_y, Py_ssize_t␣
↪neighbor_z) except *:
        return self.E
```

class javelin.energies.**DisplacementCorrelationEnergy**(*double J=0*, *double desired_correlation=NAN*)

You can either set the `desired_correlation` which will automatically adjust the pair interaction energy (J), or set the J directly.

$$E_{dis} = \sum_i \sum_n J_n x_i x_{i-n}$$

```
>>> e = DisplacementCorrelationEnergy(-1)  # J = -1 produces a positive␣
↪correlation
>>> e.evaluate(0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0)  # x1=1, y2=1
-0.0
>>> e.evaluate(0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0)  # x1=1, x2=1
-1.0
>>> e.evaluate(0, 1, 0, 0, 0, -1, 0, 0, 0, 0, 0)  # x1=1, x2=-1
1.0
>>> e.evaluate(0, 1, 1, 1, 0, 1, -1, -1, 0, 0, 0)  # x1=1, y1=1, z1=1, x2=1, y2=-
↪1, z2=-1
0.3333333333333333
```

J

J: 'double' Interaction energy, positive J will creates negative correlations while negative J creates positive correlations

desired_correlation

The desired displacement correlation, this will automatically adjust the interaction energy (J) during the *javelin.mc.MC* execution to achieve the desired correlation. The starting J can also be specified

evaluate(*self*, *int a1*, *double x1*, *double y1*, *double z1*, *int a2*, *double x2*, *double y2*, *double z2*, *Py_ssize_t target_x*, *Py_ssize_t target_y*, *Py_ssize_t target_z*) → double

class javelin.energies.**Energy**

This is the base energy class that all energies must inherit from. Inherited class should then override the evaluate method but keep the same function signature. This energy is always 0.

```
>>> e = Energy()
>>> e.evaluate(1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1)
0.0
```

correlation_type

This is used when feedback is applied, 0 = no_correlations, 1 = occupancy, 2 = displacement. The `desired_correlation` must also be set on the energy for this to work.

evaluate(*self*, *int a1*, *double x1*, *double y1*, *double z1*, *int a2*, *double x2*, *double y2*, *double z2*, *Py_ssize_t target_x*, *Py_ssize_t target_y*, *Py_ssize_t target_z*) → double

This function always returns a double of the energy calcuated. This base Energy class always returns 0.

The evaluate method get passed the atom type (Z) and positions in the unit cell (x, y and z) of the two atoms to be compared, (atom1 is `a1`, `x1`, `y1`, `z1`, atom2 is `a2`, `x2`, `y2`, `z2`) and the neighbor vector (`target_x`, `target_y`, `target_z`) which is the number of unit cells that separate the two atoms to be compared

run(*self*, *int64_t[:, :, :, ::1] a*, *double[:, :, :, ::1] x*, *double[:, :, :, ::1] y*, *double[:, :, :, ::1] z*, *Py_ssize_t[:] cell*, *Py_ssize_t[:, :] neighbors*, *Py_ssize_t number_of_neighbors*, *Py_ssize_t mod_x*, *Py_ssize_t mod_y*, *Py_ssize_t mod_z*) → double

class javelin.energies.**IsingEnergy**(*int atom1*, *int atom2*, *double J=0*, *double desired_correlation=NAN*)

The Ising model

You can either set the `desired_correlation` which will automatically adjust the pair interaction energy (J), or set the J directly.

$$E_{occ} = \sum_i \sum_{n,n \neq i} J_n \sigma_i \sigma_{i-n}$$

The atom site occupancy is represented by Ising spin variables $\sigma_i = \pm 1$. $\sigma = -1$ is when a site is occupied by *atom1* and $\sigma = +1$ is for *atom2*.

```
>>> e = IsingEnergy(13, 42, -1)  # J = -1 produces a positive correlation
>>> e.atom1
13
>>> e.atom2
42
>>> e.J
-1.0
>>> e.evaluate(1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0)  # a1=1, a2=1
-0.0
>>> e.evaluate(13, 0, 0, 0, 13, 0, 0, 0, 0, 0, 0)  # a1=13, a2=13
-1.0
>>> e.evaluate(42, 0, 0, 0, 13, 0, 0, 0, 0, 0, 0)  # a1=42, a2=13
1.0
>>> e.evaluate(42, 0, 0, 0, 42, 0, 0, 0, 0, 0, 0)  # a1=42, a2=42
-1.0
```

J

J: 'double' Interaction energy, positive J will creates negative correlations while negative J creates positive correlations

atom1

atom2

desired_correlation

The desired occupancy correlation, this will automatically adjusted the interaction energy (J) during the *javelin.mc.MC* execution to achieve the desired correlation. The starting J can also be specified

evaluate(*self*, *int a1*, *double x1*, *double y1*, *double z1*, *int a2*, *double x2*, *double y2*, *double z2*, *Py_ssize_t target_x*, *Py_ssize_t target_y*, *Py_ssize_t target_z*) → double

class javelin.energies.**LennardJonesEnergy**(*double D*, *double desired*, *int atom_type1=-1*, *int atom_type2=-1*)

The Lennard-Jones potential is a more realistic potential than *javelin.energies.SpringEnergy* that takes into account the strong repulsion between atoms as a close distance.

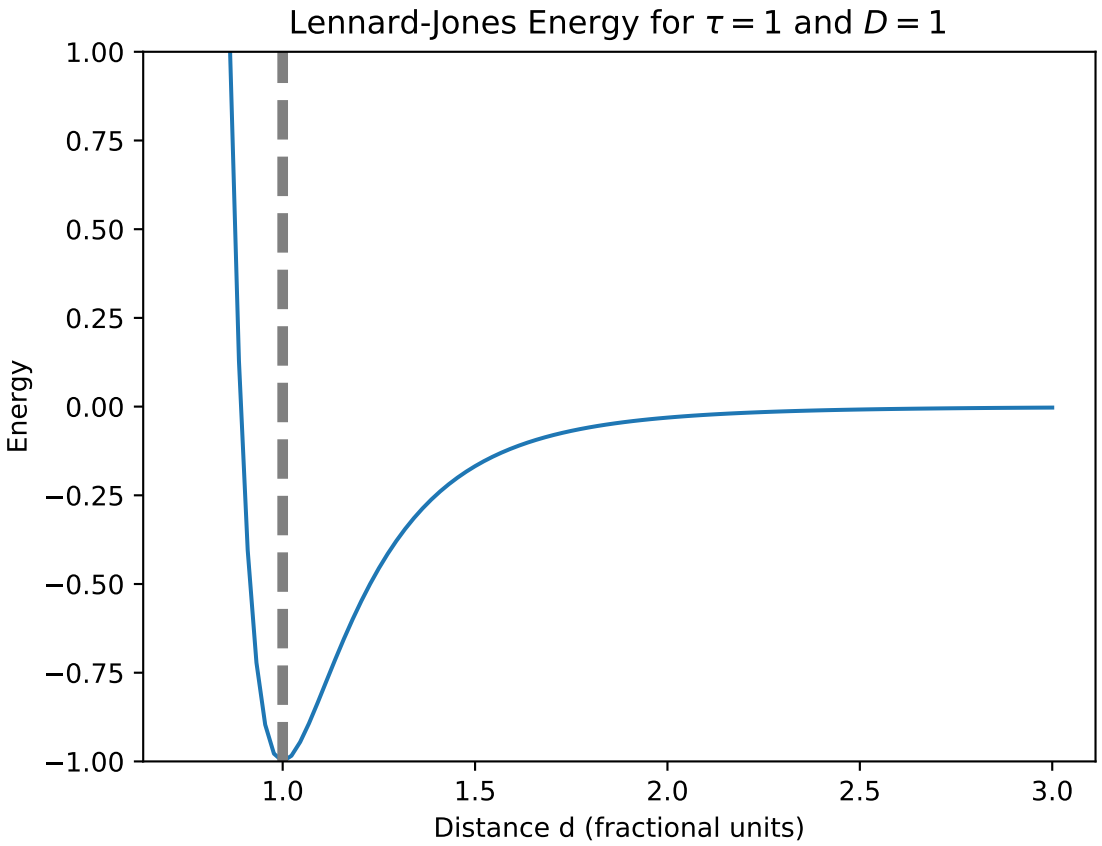
The Lennard-Jones potential is described by E_{lj}

$$E_{lj} = \sum_i \sum_{n \neq i} D \left[\left(\frac{\tau_{in}}{d_{in}} \right)^{12} - 2 \left(\frac{\tau_{in}}{d_{in}} \right)^{6} \right]$$

D is the depth of the potential well, d is the distance between the atoms and τ is the desired distance (minimum energy occurs at $d = \tau$). Distances, d and τ are in fractional coordinates.

```
>>> e = LennardJonesEnergy(1, 1)  # D = 1, desired=1
>>> e.evaluate(0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0)  # target_x=1  d=1
```

(continues on next page)


Lennard-Jones Energy for $\tau = 1$ and $D = 1$
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
Energy
1.0
1.5
2.0
2.5
3.0
Distance d (fractional units)

(continued from previous page)

```
-1.0
>>> e.evaluate(0, 0, 0, 0, 0, 0.5, 0, 0, 1, 0, 0)  # x2=0.5, target_x=1  d=1.5
-0.167876
>>> e.evaluate(0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0)  # x2=1, target_x=1  d=2
-0.031006
>>> e.evaluate(0, 0, 0, 0, 0, -0.5, 0, 0, 1, 0, 0)  # x2=-0.5, target_x=1  d=0.5
3968.0
>>> e.evaluate(0, 0, 0, 0, 0, 0, 0.5, 0, 1, 0, 0)  # y2=0.5, target_x=1  d=1.118
-0.761856
```

Optionally you can define a particular atom combination that only this energy will apply to. You can do this by setting `atom_type1` and `atom_type2` (must set both otherwise this is ignored). If the atoms that are currently being evaluated don't match then the energy will be 0. It is suggested to include energies for all possible atom combinations in the simulation. For example

```
>>> e = LennardJonesEnergy(D=1, desired=1, atom_type1=11, atom_type2=17)  # Na -␣
↪Cl
>>> e.evaluate(99, 0, 0, 0, 99, 0.5, 0, 0, 1, 0, 0)  # a1 = 99, a2 = 99
0.0
>>> e.evaluate(11, 0, 0, 0, 11, 0.5, 0, 0, 1, 0, 0)  # a1 = 11, a2 = 11
0.0
>>> e.evaluate(11, 0, 0, 0, 17, 0.5, 0, 0, 1, 0, 0)  # a1 = 11, a2 = 17
-0.167876
>>> e.evaluate(17, 0, 0, 0, 11, 0.5, 0, 0, 1, 0, 0)  # a1 = 17, a2 = 11
-0.167876
>>>
>>> e = LennardJonesEnergy(D=1, desired=1, atom_type1=17, atom_type2=17)  # Cl -␣
↪Cl
>>> e.evaluate(99, 0, 0, 0, 99, 0.5, 0, 0, 1, 0, 0)  # a1 = 99, a2 = 99
0.0
>>> e.evaluate(11, 0, 0, 0, 11, 0.5, 0, 0, 1, 0, 0)  # a1 = 11, a2 = 11
0.0
>>> e.evaluate(11, 0, 0, 0, 17, 0.5, 0, 0, 1, 0, 0)  # a1 = 11, a2 = 17
0.0
>>> e.evaluate(17, 0, 0, 0, 17, 0.5, 0, 0, 1, 0, 0)  # a1 = 17, a2 = 17
-0.167876
```

D

atom_type1

atom_type2

desired

evaluate(*self*, *int a1*, *double x1*, *double y1*, *double z1*, *int a2*, *double x2*, *double y2*, *double z2*, *Py_ssize_t target_x*, *Py_ssize_t target_y*, *Py_ssize_t target_z*) → double

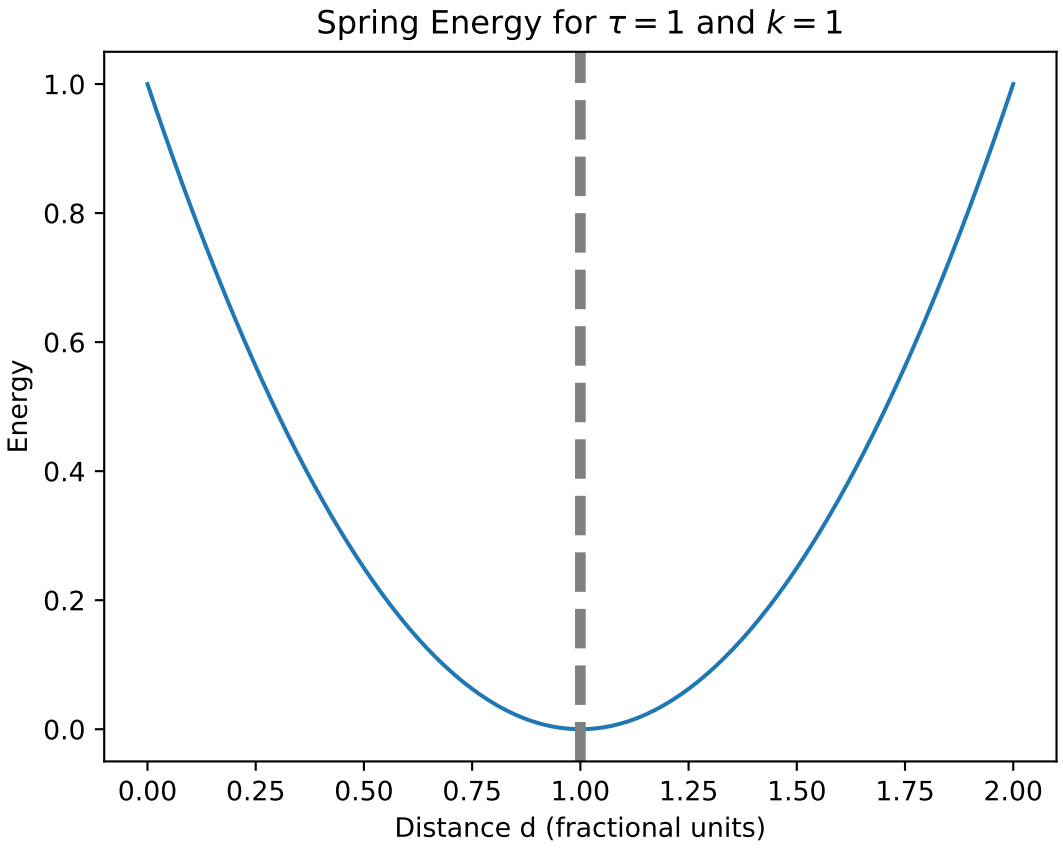
class javelin.energies.**SpringEnergy**(*double K*, *double desired*, *int atom_type1=-1*, *int atom_type2=-1*)

Hooke's law for a simple spring can be used to describe atoms joined together by springs and are in a harmonic potential.

The spring energy is described by E_{spring}

$$E_{spring} = \sum_i \sum_n k_n [d_{in} - \tau_{in}]^2$$

k is the force constant, d is the distance between the atoms and τ is the desired distance (minimum energy occurs at $d = \tau$). Distances, d and τ are in fractional coordinates.


Spring Energy for $\tau = 1$ and $k = 1$
Energy
1.0
0.8
0.6
0.4
0.2
0.0
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
Distance d (fractional units)

```
>>> e = SpringEnergy(1, 1)  # K = 1, desired=1
>>> e.evaluate(0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0)  # target_x=1  d=1
0.0
>>> e.evaluate(0, 0, 0, 0, 0, 0.5, 0, 0, 1, 0, 0)  # x2=0.5, target_x=1  d=1.5
0.25
>>> e.evaluate(0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0)  # x2=1, target_x=1  d=2
1.0
>>> e.evaluate(0, 0, 0, 0, 0, -0.5, 0, 0, 1, 0, 0)  # x2=-0.5, target_x=1  d=0.5
0.25
>>> e.evaluate(0, 0, 0, 0, 0, 0, 0.5, 0, 1, 0, 0)  # y2=0.5, target_x=1  d=1.118
0.013932
```

Optionally you can define a particular atom combination that only this energy will apply to. You can do this by setting `atom_type1` and `atom_type2` (must set both otherwise this is ignored). If the atoms that are currently being evaluated don't match then the energy will be 0. It is suggested to include energies for all possible atom combinations in the simulation. For example

```
>>> e = SpringEnergy(K=1, desired=1, atom_type1=11, atom_type2=17)  # Na - Cl
>>> e.evaluate(99, 0, 0, 0, 99, 0.5, 0, 0, 1, 0, 0)  # a1 = 99, a2 = 99
0.0
>>> e.evaluate(11, 0, 0, 0, 11, 0.5, 0, 0, 1, 0, 0)  # a1 = 11, a2 = 11
0.0
>>> e.evaluate(11, 0, 0, 0, 17, 0.5, 0, 0, 1, 0, 0)  # a1 = 11, a2 = 17
0.25
>>> e.evaluate(17, 0, 0, 0, 11, 0.5, 0, 0, 1, 0, 0)  # a1 = 17, a2 = 11
0.25
>>>
>>> e = SpringEnergy(K=1, desired=1, atom_type1=17, atom_type2=17)  # Cl - Cl
>>> e.evaluate(99, 0, 0, 0, 99, 0.5, 0, 0, 1, 0, 0)  # a1 = 99, a2 = 99
0.0
>>> e.evaluate(11, 0, 0, 0, 11, 0.5, 0, 0, 1, 0, 0)  # a1 = 11, a2 = 11
0.0
>>> e.evaluate(11, 0, 0, 0, 17, 0.5, 0, 0, 1, 0, 0)  # a1 = 11, a2 = 17
0.0
>>> e.evaluate(17, 0, 0, 0, 17, 0.5, 0, 0, 1, 0, 0)  # a1 = 17, a2 = 17
0.25
```

K

atom_type1

atom_type2

desired

evaluate(*self*, *int a1*, *double x1*, *double y1*, *double z1*, *int a2*, *double x2*, *double y2*, *double z2*, *Py_ssize_t target_x*, *Py_ssize_t target_y*, *Py_ssize_t target_z*) → double

4.2 fourier

This module define the Fourier class and other functions related to the fourier transformation.

class javelin.fourier.**Fourier**

The Fourier class contains everything required to calculate the diffuse scattering. The only required thing to be set is `javelin.fourier.Fourier.structure`. There are defaults for all other options including **grid**, **radiation**, **average** structure subtraction and **lots** options.

Examples

```
>>> from javelin.structure import Structure
>>> fourier = Fourier()
>>> print(fourier)
Radiation         : neutron
Fourier volume    : complete crystal
Aver. subtraction : False
<BLANKLINE>
Reciprocal layer  :
lower left  corner :     [ 0.  0.  0.]
lower right corner :     [ 1.  0.  0.]
upper left  corner :     [ 0.  1.  0.]
top   left  corner :     [ 0.  0.  1.]
<BLANKLINE>
hor. increment     :     [ 0.01  0.    0.  ]
vert. increment    :     [ 0.    0.01  0.  ]
top   increment    :     [ 0.  0.  1.]
<BLANKLINE>
# of points        :     101 x 101 x 1
>>> results = fourier.calc(Structure())
>>> print(results) # doctest: +SKIP
<xarray.DataArray 'Intensity' ([ 1.  0.  0.]: 101, [ 0.  1.  0.]: 101, [ 0.  0.
↪1.]: 1)>
array([[[ 0.],
        [ 0.],
        ...,
        [ 0.],
        [ 0.]],
<BLANKLINE>
       [[ 0.],
        [ 0.],
        ...,
        [ 0.],
        [ 0.]],
<BLANKLINE>
       ...,
       [[ 0.],
        [ 0.],
        ...,
        [ 0.],
        [ 0.]],
<BLANKLINE>
       [[ 0.],
        [ 0.],
        ...,
        [ 0.],
        [ 0.]]])
Coordinates:
  * [ 1.  0.  0.]  ([ 1.  0.  0.]) float64 0.0 0.01 0.02 0.03 0.04 0.05 0.06 ...
  * [ 0.  1.  0.]  ([ 0.  1.  0.]) float64 0.0 0.01 0.02 0.03 0.04 0.05 0.06 ...
  * [ 0.  0.  1.]  ([ 0.  0.  1.]) float64 0.0
Attributes:
    units:    r.l.u
```

approximate

This sets the options of calculating the approximate scattering instead of exact. This is much quicker and is likely good enough for most cases.

Getter Returns bool of approximate scattering option

Setter Sets whether approximate sacttering is calculated

Type bool

average
This sets the options of calculating average structure and subtracted it from the simulated scattering

Getter Returns bool of average structure subtraction option

Setter Sets whether average structure is subtracted

Type bool

calc(*structure*)
Calculates the fourier transform

Parameters **structure** – The structure from which fourier transform

is calculated. The calculation work with any of the following types of structures *javelin.structure.Structure*, ase.Atoms or diffpy.Structure.structure.Structure but if you are using average structure subtraction or the lots option it needs to be *javelin.structure.Structure* type.

Returns DataArray containing calculated diffuse scattering

Return type xarray.DataArray

calc_average(*structure*)
Calculates the scattering from the avarage structure

Parameters **structure** – The structure from which fourier transform

is calculated. The calculation work with any of the following types of structures *javelin.structure.Structure*, ase.Atoms or diffpy.Structure.structure.Structure but if you are using average structure subtraction or the lots option it needs to be *javelin.structure.Structure* type.

Returns DataArray containing calculated average scattering

Return type xarray.DataArray

grid = None
The **grid** attribute defines the reciprocal volume from which the scattering will be calculated. Must of type *javelin.grid.Grid* And check *javelin.grid.Grid* for details on how to change the grid.

lots
The size of lots

Getter Returns the lots size

Setter Sets the lots size

Type list of 3 integers or None

magnetic
This sets the options of calculating the magnetic scattering instead of nuclear. This assume neutrons are being used.

Getter Returns bool of magnetic scattering option

Setter Sets whether magnetic sacttering is calculated

Type bool

number_of_lots
The number of lots to use

Getter Returns the number of lots

Setter Sets the number of lots

Type int

radiation
The radiation used

Getter Returns the radiation selected

Setter Sets the radiation

Type str ('xray' or 'neutron')

javelin.fourier.**create_xarray_dataarray**(*values*, *grid*)
Create a xarry DataArray from the input numpy array and grid object.

Parameters

- **values** (numpy.ndarray) – Input array containing the scattering intensities
- **numbers** (*javelin.grid.Grid*) – Grid object describing the array properties

Returns DataArray produced from the values and grid object

Return type xarray.DataArray

javelin.fourier.**get_ff**(*atomic_number*, *radiation*, *q=None*)
Returns the form factor for a given atomic number, radiation and q values

Parameters

- **atomic_number** (*int*) – atomic number
- **radiation** (*str*) – type of radiation ('xray' or 'neutron')
- **q** (float, list, numpy.ndarray) – value or values of q for which to get form factors

Returns form factors for given q

Return type float, numpy.ndarray

Examples

```
>>> get_ff(8, 'neutron')
5.805
```

```
>>> get_ff(8, 'xray', q=2.0)
6.31826029176493
```

```
>>> get_ff(8, 'xray', q=[0.0, 3.5, 7.0])
array([ 7.999706  ,  4.38417867,  2.08928068])
```

javelin.fourier.**get_mag_ff**(*atomic_number*, *q*, *ion=0*, *j=0*)
Returns the j0 magnetic form factor for a given atomic number, radiation and q values

Parameters

- **atomic_number** (*int*) – atomic number
- **q** (float, list, numpy.ndarray) – value or values of q for which to get form factors
- **ion** (*int*) – charge of selected atom
- **j** (*int*) – order of spherical Bessel function (0, 2, 4 or 6)

Returns magnetic form factor for given q

Return type float, `numpy.ndarray`

Examples

```
>>> get_mag_ff(8, q=2, ion=1)
0.58510426376585045
```

```
>>> get_mag_ff(26, q=[0.0, 3.5, 7.0], ion=2)
array([ 1.        ,  0.49729671,  0.09979243])
```

```
>>> get_mag_ff(26, q=[0.0, 3.5, 7.0], ion=4)
array([ 0.9997    ,  0.58273549,  0.13948496])
```

```
>>> get_mag_ff(26, q=[0.0, 3.5, 7.0], ion=4, j=4)
array([ 0.       ,  0.0149604,  0.0759222])
```

4.3 grid

class javelin.grid.**Grid**

Grid class to allow the Q-space grid to be defined in different ways. The grid can be defined be either specifying the corners of the volume or by the axis vectors.

The grid is defined by three vectors **v1**, **v2** and **v3**; the range of these vectors **r1**, **r2** and **r3**; and the number of bin in each of these directions.

Examples

Setting grid by defining corners

```
>>> grid = Grid()
>>> grid.set_corners(ll=[-2,-2,-3], lr=[2,2,-3], ul=[-2,-2,3])
>>> grid.bins = 5, 3
>>> grid.v1
array([ 1.,  1.,  0.])
>>> grid.v2
array([ 0.,  0.,  1.])
>>> grid.v3
array([ 1., -1.,  0.])
>>> grid.r1
array([-2., -1.,  0.,  1.,  2.])
>>> grid.r2
array([-3.,  0.,  3.])
>>> grid.r3
array([ 0.])
>>> print(grid)
lower left  corner :     [-2. -2. -3.]
lower right corner :     [ 2.  2. -3.]
upper left  corner :     [-2. -2.  3.]
top   left  corner :     [-2. -2. -3.]
<BLANKLINE>
hor. increment     :     [ 1.  1.  0.]
vert. increment    :     [ 0.  0.  3.]
top   increment    :     [ 1. -1.  0.]
```

(continues on next page)

(continued from previous page)

```
<BLANKLINE>
# of points         :     5 x 3 x 1
```

Setting grid by vectors and ranges

```
>>> grid = Grid()
>>> grid.v1 = [1, 1, 1]
>>> grid.v2 = [2, -1, -1]
>>> grid.v3 = [0, 1, -1]
>>> grid.r1 = [-1, 1]
>>> grid.r2 = [0, 2]
>>> grid.r3 = [0, 2]
>>> grid.bins = 3, 3, 3
>>> print(grid)
lower left  corner :     [-1 -1 -1]
lower right corner :     [1 1 1]
upper left  corner :     [ 3 -3 -3]
top   left  corner :     [-1  1 -3]
<BLANKLINE>
hor. increment     :     [ 1.  1.  1.]
vert. increment    :     [ 2. -1. -1.]
top   increment    :     [ 0.  1. -1.]
<BLANKLINE>
# of points         :     3 x 3 x 3
```

bins

The number of bins in each direction

```
>>> grid = Grid()
>>> grid.bins
(101, 101, 1)
```

```
>>> grid.bins = 5
>>> grid.bins
(5, 1, 1)
```

```
>>> grid.bins = 5, 6
>>> grid.bins
(5, 6, 1)
```

```
>>> grid.bins = 5, 6, 7
>>> grid.bins
(5, 6, 7)
```

Getter Returns the number of bins in each direction

Setter Sets the number of bins, provide 1, 2 or 3 integers

Type `numpy.ndarray` of int

get_axes_names()

```
>>> grid = Grid()
>>> grid.get_axes_names()
('[ 1.  0.  0.]', '[ 0.  1.  0.]', '[ 0.  0.  1.]')
```

Returns Axis names, vector of each direction

Return type tuple of str

get_q_meshgrid()
Equivalent to numpy.mgrid for this volume

Returns mesh-grid numpy.ndarray all of the same dimensions

Return type tuple of numpy.ndarray

get_squashed_q_meshgrid()
Almost equivalent to numpy.ogrid for this volume. It may have more than one dimension not equal to 1. This can be used with numpy broadcasting.

Returns mesh-grid numpy.ndarray with some dimension equal to 1

Return type tuple of numpy.ndarray

ll

Returns Lower-left corner of reciprocal volume

Return type numpy.ndarray

lr

Returns Lower-right corner of reciprocal volume

Return type numpy.ndarray

r1
Set the range of the first axis, two values min and max

Getter Array of values for each bin in the axis

Setter Range of first axis, two values

Type numpy.ndarray

r2
Set the range of the second axis, two values min and max

Getter Array of values for each bin in the axis

Setter Range of second axis, two values

Type numpy.ndarray

r3
Set the range of the third axis, two values min and max

Getter Array of values for each bin in the axis

Setter Range of third axis, two values

Type numpy.ndarray

set_corners(*ll=(0, 0, 0)*, *lr=None*, *ul=None*, *tl=None*)
Define the axis vectors by the corners of the reciprocal volume. The corners values with be converted into axis vectors, see *javelin.grid.corners_to_vectors()* for details.

Parameters

- **ll** (*array-like of length 3*) – lower-left corner
- **lr** (*array-like of length 3*) – lower-right corner

- **ul** (*array-like of length 3*) – upper-left corner
- **tl** (*array-like of length 3*) – top-left corner

tl

Returns Top-left corner of reciprocal volume

Return type numpy.ndarray

ul

Returns Upper-left corner of reciprocal volume

Return type numpy.ndarray

v1

Getter Set the first axis

Setter Get the first axis

Type numpy.ndarray

v1_delta

Returns Increment vector of first axis

Return type numpy.ndarray

v2

Getter Set the second axis

Setter Get the second axis

Type numpy.ndarray

v2_delta

Returns Increment vector of second axis

Return type numpy.ndarray

v3

Getter Set the third axis

Setter Get the third axis

Type numpy.ndarray

v3_delta

Returns Increment vector of third axis

Return type numpy.ndarray

javelin.grid.**angle**(*v1*, *v2*)

Calculates the angle between two vectors

Parameters

- **v1** (*array-like object of numbers*) – vector 1
- **v2** (*array-like object of numbers*) – vector 2

Returns angle (radians)

Return type float

Examples

```
>>> angle([1,0,0], [-1,0,0])
3.1415926535897931
```

```
>>> angle([1,0,0], [0,1,0])
1.5707963267948966
```

javelin.grid.**check_parallel**(*v1*, *v2*)

Checks if two vectors are parallel

Parameters

- **v1** (*array-like object of numbers*) – vector1
- **v2** (*array-like object of numbers*) – vector2

Returns if parallel

Return type bool

Examples

```
>>> check_parallel([1,0,0], [-1,0,0])
True
```

```
>>> check_parallel([1,0,0], [0,1,0])
False
```

javelin.grid.**corners_to_vectors**(*ll=None*, *lr=None*, *ul=None*, *tl=None*)

This function converts the provided corners into axes vectors and axes ranges. It will also calculate sensible vectors for any unprovided corners.

You must provide at minimum the lower-left (**ll**) and lower-right (**lr**) corners.

Parameters

- **ll** (*array-like object of numbers*) – lower-left corner (required)
- **lr** (*array-like object of numbers*) – lower-right corner (required)
- **ul** (*array-like object of numbers*) – upper-left corner
- **tl** (*array-like object of numbers*) – top-left corner

Returns three axes vectors and three axes ranges

Return type tuple of three `numpy.ndarray` and three tuple ranges

Examples

Using only **ll** and **lr**, the other two vector are calculated using *javelin.grid.find_other_vectors()*

```
>>> v1, v2, v3, r1, r2, r3 = corners_to_vectors(ll=[-3,-3,0], lr=[3, 3, 0])
>>> print(v1, v2, v3)
[ 1.  1.  0.] [ 1.  0.  0.] [ 0.  0.  1.]
>>> print(r1, r2, r3) # doctest: +SKIP
(-3.0, 3.0) (0.0, 0.0) (0.0, 0.0)
```

Using **ll**, **lr** and **ul**, the other vector is the *javelin.grid.norm1()* of the cross product of the first two vectors defined by the corners.

```
>>> v1, v2, v3, r1, r2, r3 = corners_to_vectors(ll=[-3,-3,-2], lr=[3, 3, -2],␣
↪ul=[-3, -3, 2])
>>> print(v1, v2, v3)
[ 1.  1.  0.] [ 0.  0.  1.] [ 1. -1.  0.]
>>> print(r1, r2, r3) # doctest: +SKIP
(-3.0, 3.0) (-2.0, 2.0) (0.0, 0.0)
```

Finally defining all corners

```
>>> v1,v2,v3,r1,r2,r3 = corners_to_vectors(ll=[-5,-6,-7],lr=[-5,-6,7],ul=[-5,6,-
↪7],tl=[5,-6,-7])
>>> print(v1, v2, v3)
[ 0.  0.  1.] [ 0.  1.  0.] [ 1.  0.  0.]
>>> print(r1, r2, r3)
(-7.0, 7.0) (-6.0, 6.0) (-5.0, 5.0)
```

If you provided corners which will create parallel vectors you will get a ValueError

```
>>> corners_to_vectors(ll=[0, 0, 0], lr=[1, 0, 0], ul=[1, 0, 0])
Traceback (most recent call last):
    ...
ValueError: Vector from ll to lr is parallel with vector from ll to ul
```

javelin.grid.**find_other_vectors**(*v*)

This will find two new vectors which in combination with the provided vector (**v**) will form a basis for a complete space filling set.

Parameters **v** (*array-like object of numbers*) – vector

Returns two new space filling vectors

Return type tuple of two numpy.ndarray

Examples

```
>>> find_other_vectors([1, 0, 0])
(array([ 0.,  1.,  0.]), array([ 0.,  0.,  1.]))
```

```
>>> find_other_vectors([0, 0, 1])
(array([ 1.,  0.,  0.]), array([ 0.,  1.,  0.]))
```

```
>>> find_other_vectors([1, 1, 0])
(array([ 1.,  0.,  0.]), array([ 0.,  0.,  1.]))
```

javelin.grid.**get_vector_from_points**(*p1*, *p2*)

Calculates the vector form two points

Parameters

- **p1** (*array-like object of numbers*) – point 1
- **p2** (*array-like object of numbers*) – point 2

Returns vector between points

Return type numpy.ndarray

Examples

```
>>> get_vector_from_points([-1, -1, 0], [1, 1, 0])
array([ 1.,  1.,  0.])
```

```
>>> get_vector_from_points([0, 0, 0], [2, 2, 4])
array([ 1.,  1.,  2.])
```

javelin.grid.**length**(*v*)

Calculates the length of a vector

Parameters **v** (*array-like object of numbers*) – vector

Returns length of vector

Return type float

Examples

```
>>> length([1,0,0])
1.0
```

```
>>> length([1,-1,0])
1.4142135623730951
```

```
>>> length([2,2,2])
3.4641016151377544
```

javelin.grid.**norm**(*v*)

Calculates the normalised vector

Parameters **v** (*array-like object of numbers*) – vector

Returns normalised vector

Return type numpy.ndarray

Examples

```
>>> norm([5, 0, 0])
array([ 1.,  0.,  0.])
```

```
>>> norm([1, 1, 0])
array([ 0.70710678,  0.70710678,  0.        ])
```

javelin.grid.**norm1**(*v*)

Calculate the equivalent vector with the smallest non-zero component equal to one.

Parameters **v** (*array-like object of numbers*) – vector

Returns normalised1 vector

Return type numpy.ndarray

Examples

```
>>> norm1([5, 10, 0])
array([ 1.,  2.,  0.])
```

```
>>> norm1([1, 1, 0])
array([ 1.,  1.,  0.])
```

4.4 io

javelin.io.**load_HDF5_to_xarray**(*filename*)
Load HDF file into an xarray DataArray using pandas HDFStore

requries pytables

javelin.io.**numpy_to_vti**(*array*, *origin*, *spacing*, *filename*)
This function write a VtkImageData vti file from a numpy array.

Parameters

- **array** (numpy.ndarray) – input array
- **origin** (*array like object of values*) – the origin of the array
- **spacing** (*array like object of values*) – the step in each dimension
- **filename** (*str*) – output filename (.vti)

javelin.io.**read_mantid_MDHisto**(*filename*)
Read the saved MDHisto from from Mantid and returns an xarray.DataArray object

javelin.io.**read_stru**(*filename*, *starting_cell=(1, 1, 1)*)
Read in a .stru file saved from DISCUS into a javelin Structure

If the line ncell is not present in the file all the atoms will be read into a single cell.

javelin.io.**read_stru_to_ase**(*filename*)
This function read the legacy DISCUS stru file format into a ASE Atoms object.

Parameters **filename** (*str*) – filename of DISCUS stru file

Returns ASE Atoms object

Return type ase.Atoms

javelin.io.**save_mantid_MDHisto**(*dataArray*, *filename*)
Save a file that can be read in using Mantid's LoadMD

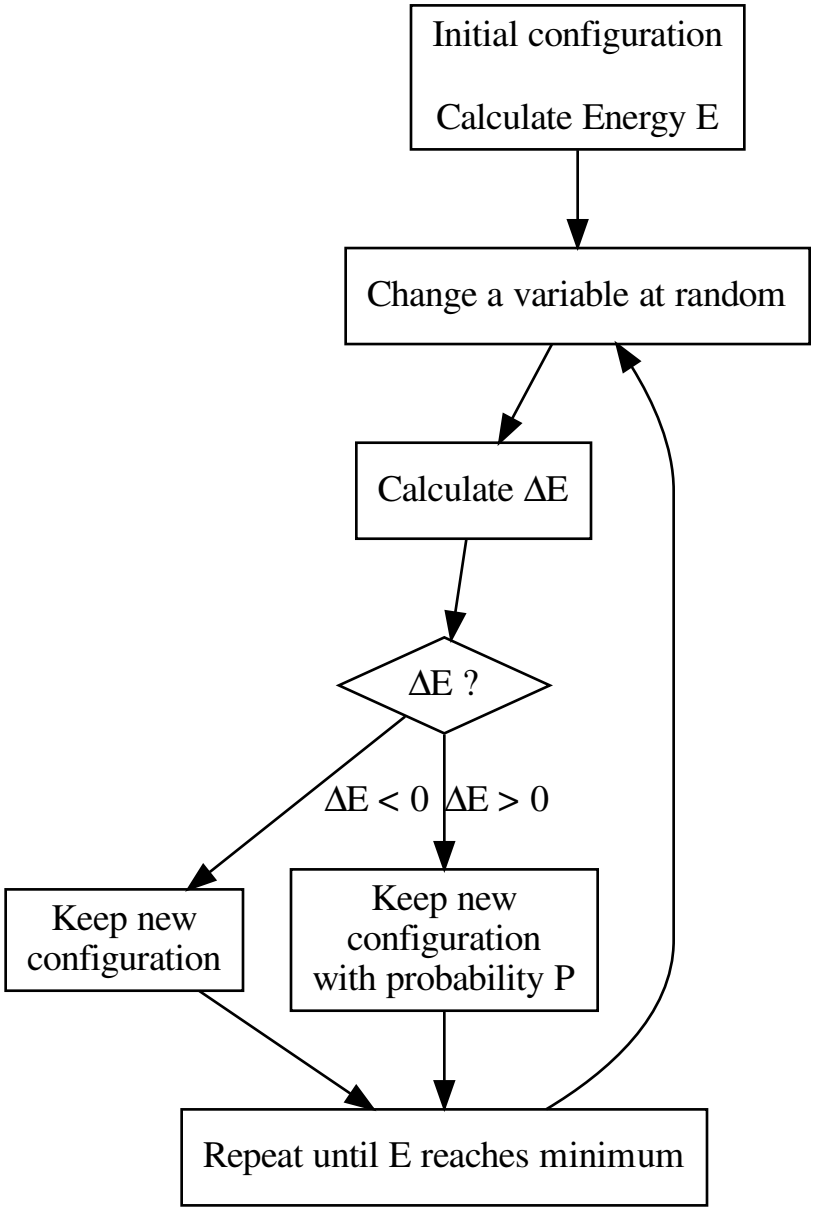
javelin.io.**save_xarray_to_HDF5**(*dataArray*, *filename*, *complib=None*)
Save the xarray DataArray to HDF file using pandas HDFStore

attrs will be saved as metadata via pickle

requries pytables

complib : {'zlib', 'bzip2', 'lzo', 'blosc', None}, default None

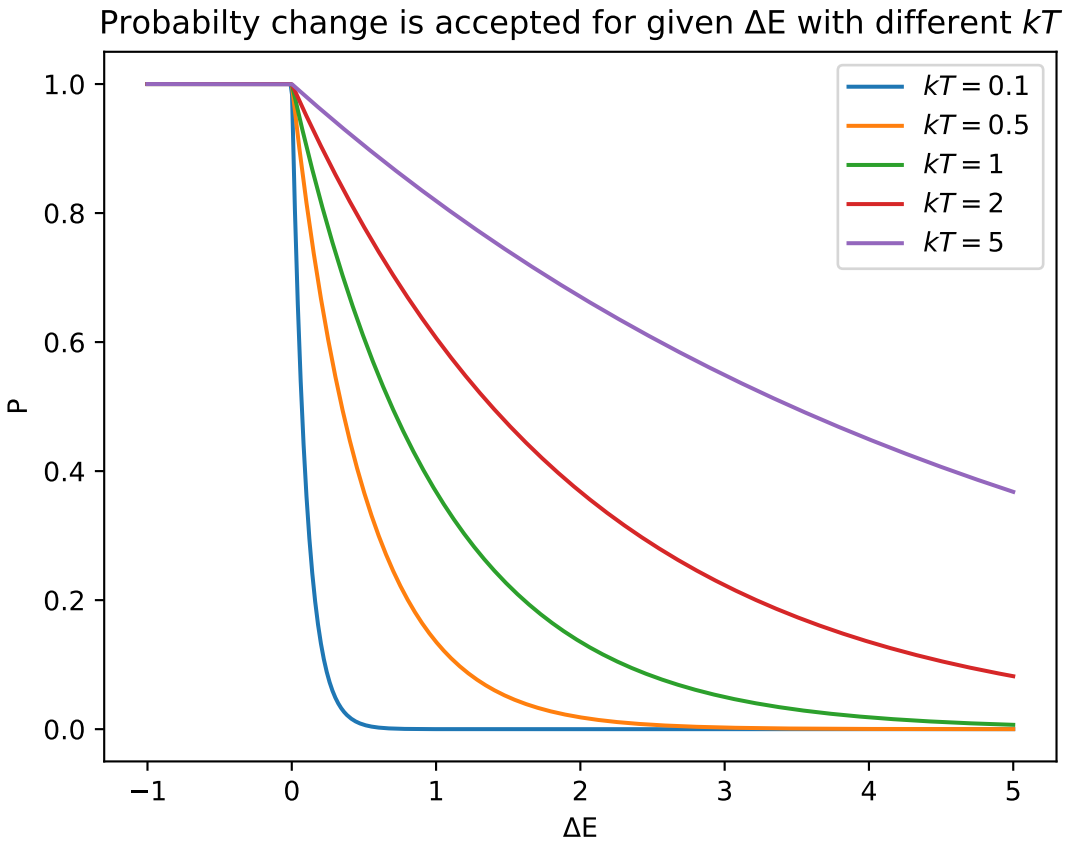
4.5 mc



where

$$P = \exp(-\Delta E/kT)$$

class javelin.mc.**MC**


Probabilty change is accepted for given ΔE with different kT
P
ΔE
1.0
0.8
0.6
0.4
0.2
0.0
−1
0
1
2
3
4
5
$kT = 0.1$
$kT = 0.5$
$kT = 1$
$kT = 2$
$kT = 5$

MonteCarlo class

For the Monte Carlo simulations to run you need to provide an input structure, target (neighbor and energy set) and modifier. A basic, do nothing, example is shown:

```
>>> from javelin.structure import Structure
>>> from javelin.modifier import BaseModifier
>>> from javelin.energies import Energy
>>> structure = Structure(symbols=['Na','Cl'],positions=[[0,0,0],[0.5,0.5,0.5]])
>>>
>>> energy = Energy()
>>> neighbors = structure.get_neighbors()
>>>
>>> mc = MC()
>>> mc.add_modifier(BaseModifier(0))
>>> mc.temperature = 1
>>> mc.cycles = 2
>>> mc.add_target(neighbors, energy)
>>>
>>> new_structure = mc.run(structure)
<BLANKLINE>
Cycle = 0, temperature = 1.0
Accepted 0 good, 1 neutral (dE=0) and 0 bad out of 1
<BLANKLINE>
Cycle = 1, temperature = 1.0
Accepted 0 good, 1 neutral (dE=0) and 0 bad out of 1
>>>
```

add_modifier(*modifier*)

add_target(*neighbors*, *energy*)

This will add an energy calculation and neighbour pair that will be used to calculate and energy for each modification step. You can add as many targets as you like.

Parameters

- **neighbour** (*javelin.neighborlist.NeighborList* or *n x 5* array of neighbor vectors) – neighbour for which the energy will be calculated over
- **energy** (*javelin.energies.Energy*) – the energy function that will be calculated for each neighbor

cycles

The number of cycles to perform.

delete_targets()

This will remove all previously set targets

iterations

The number of iterations (site modifications) to perform for each cycle. Default is equal to the number of unitcells in the structure.

modifier

This is how the structure is to be modified, must be of type *javelin.modifier.BaseModifier*.

run(*structure*, *inplace=False*)

Execute the Monte Carlo routine. You must provide the structure to modify as a parameter. This will by default this will return a new *javelin.structure.Structure* with the results, to modify the provided structure in place set *inplace=True*

Parameters **structure** (*javelin.structure.Structure*) – structure to run the Monte Carlo on

temperature
Temperature parameter (kT) which changes the probability (P) a energy change of ΔE is accepted

$$P = \exp(-\Delta E/kT)$$

Temperature can be a single value for all cycles or you can provide a different temperature for each cycle. This allows you to do quenching of the disorder. If you provide more temperatures than cycles then only the first temperatures corresponding to the number of cycles are used. If there are more cycles than temperature than for remaining cycles the last temperature in the list will be used.

```
>>> mc = MC()
>>> mc.temperature = 0.1
>>> print(mc)
Number of cycles = 100
Temperature[s] = 0.1
Structure modfifiers are []
>>>
>>> mc.temperature = np.linspace(1, 0, 6)
>>> print(mc)
Number of cycles = 100
Temperature[s] = [ 1.   0.8  0.6  0.4  0.2  0. ]
Structure modfifiers are []
```

4.6 mccore

class javelin.mccore.**Target**(*Py_ssize_t[:, :] neighbors*, *Energy energy*)
Class to hold an Energy object with it associated neighbors

energy
energy: javelin.energies.Energy

neighbors

number_of_neighbors

javelin.mccore.**mcrun**(*BaseModifier[:] modifiers*, *Target[:] targets*, *int iterations*, *double temperature*, *int64_t[:, :, ::1] a*, *double[:, :, ::1] x*, *double[:, :, ::1] y*, *double[:, :, ::1] z) -> (int, int, int)*
This function is not meant to be used directly. It is used by *javelin.mc.MC*. The function does very little validation of the input values, it you don't provide exactly what is expected then segmentation fault is likely.

4.7 modifier

The Modifier object is the method by which *javelin.mc.MC* changes a *javelin.structure.Structure*.

All modifiers inherit from *javelin.modifier.BaseModifier*.

class javelin.modifier.**BaseModifier**(*int number_of_cells=1*, *sites=0*)
This class does not actually change the structure but is the base of all modifiers. The *number_of_cells* is number of random location that the modifier will change, *for example* swap type modifiers require 2 sites

while shift requires only one. The methods `self.initialize_cells(int number_of_cells)` and `self.initialize_sites(sites)` must be called to set the number of cells and which sites to use.

cells

get_random_cells (*self*, *Py_ssize_t size_x*, *Py_ssize_t size_y*, *Py_ssize_t size_z*) → Py_ssize_t[:, :]
Sets internally and returns randomly selected cells, shape (number_of_cells, 3), based on *size_x*, *size_y* and *size_z*. This needs to be executed before `self.run`.

number_of_cells

run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void
Modifies the provided arrays (a, x, y, z) for cells selected by `self.get_random_cells`.

sites

undo_last_run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void
Undoes the last modification done by `self.run`.

By default it just executes `self.run` again assuming the process is reversible, such as swapping.

class javelin.modifier.**SetDisplacementNormal** (*sites*, *double mu*, *double sigma*)
Sets the atoms displacement in all directions to a random point in the normal distribution given by `mu` and `sigma`.

run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

undo_last_run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**SetDisplacementNormalXYZ** (*sites*, *double x_mu*, *double x_sigma*, *double y_mu*, *double y_sigma*, *double z_mu*, *double z_sigma*)
Sets the atoms displacement in all directions to a random point in the normal distribution given by `mu` and `sigma` for each direction.

run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

undo_last_run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**SetDisplacementRange** (*sites*, *double minimum*, *double maximum*)
Sets the atoms displacement in all directions to a random point in the given range.

run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

undo_last_run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**SetDisplacementRangeXYZ** (*sites*, *double x_min*, *double x_max*, *double y_min*, *double y_max*, *double z_min*, *double z_max*)
Sets the atoms displacement in all directions to a random point in the given range for each direction

run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

undo_last_run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**ShiftDisplacementNormal** (*sites*, *double mu*, *double sigma*)
Shifts the atoms displacement in all directions by a random amount in the normal distribution given by `mu` and `sigma`.

run (*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

undo_last_run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**ShiftDisplacementNormalXYZ**(*sites*, *double x_mu*, *double x_sigma*, *double y_mu*, *double y_sigma*, *double z_mu*, *double z_sigma*)

Shifts the atoms displacement in all directions by a random amount in the normal distribution given by `mu` and `sigma` for each direction.

run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

undo_last_run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**ShiftDisplacementRange**(*sites*, *double minimum*, *double maximum*)

Shifts the atoms displacement in all directions by a random amount in the given range.

run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

undo_last_run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**ShiftDisplacementRangeXYZ**(*sites*, *double x_min*, *double x_max*, *double y_min*, *double y_max*, *double z_min*, *double z_max*)

Shifts the atoms displacement in all directions by a random amount in the given range for each direction

run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

undo_last_run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**Swap**(*sites*)

Swap the atom occupancy and displacement at swap_site between two cells.

run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**SwapDisplacement**(*sites*)

Swap the atom displacement at swap_site between two cells.

run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

class javelin.modifier.**SwapOccupancy**(*sites*)

Swap the atoms occupancy at swap_site between two cells.

run(*self*, *int64_t[:, :, :, :] a*, *double[:, :, :, :] x*, *double[:, :, :, :] y*, *double[:, :, :, :] z*) → void

4.8 neighborlist

class javelin.neighborlist.**NeighborList**(*vectors=None*)

The NeighborList class

Contains an *n x 5* array of neighbor vectors each being *[origin_site, target_site, i, j, k]*.

```
>>> nl = NeighborList()
>>> print(nl)
      |     site      |    vector
index | origin target |   i   j   k
<BLANKLINE>
>>> nl = NeighborList([[0,1,1,0,0],[0,1,0,1,0],[0,1,0,0,1]])
>>> print(nl)
      |     site      |    vector
```

(continues on next page)

(continued from previous page)

```
index | origin target |   i   j   k
    0 |      0      1 |   1   0   0
    1 |      0      1 |   0   1   0
    2 |      0      1 |   0   0   1
```

You can slice a NeighborList:

```
>>> print(nl[0])
      |     site      |    vector
index | origin target |   i   j   k
    0 |      0      1 |   1   0   0
>>> print(nl[-1])
      |     site      |    vector
index | origin target |   i   j   k
    0 |      0      1 |   0   0   1
>>> print(nl[1:3])
      |     site      |    vector
index | origin target |   i   j   k
    0 |      0      1 |   0   1   0
    1 |      0      1 |   0   0   1
>>> print(nl[0,2])
      |     site      |    vector
index | origin target |   i   j   k
    0 |      0      1 |   1   0   0
    1 |      0      1 |   0   0   1
```

You can set a vector:

```
>>> nl[0] = [5, 5, 5, 5, 5]
>>> nl[-1] = [2, 2, 2, 2, 2]
>>> print(nl)
      |     site      |    vector
index | origin target |   i   j   k
    0 |      5      5 |   5   5   5
    1 |      0      1 |   0   1   0
    2 |      2      2 |   2   2   2
```

You can add NeighborLists together:

```
>>> nl2 = NeighborList([[1,1,1,0,0],[1,1,0,1,0]])
>>> nl3 = nl + nl2
>>> print(nl3)
      |     site      |    vector
index | origin target |   i   j   k
    0 |      5      5 |   5   5   5
    1 |      0      1 |   0   1   0
    2 |      2      2 |   2   2   2
    3 |      1      1 |   1   0   0
    4 |      1      1 |   0   1   0
```

And finallly you can delete vectors from the NeighborList

```
>>> del nl3[-1]
>>> print(nl3)
      |     site      |    vector
index | origin target |   i   j   k
```

(continues on next page)

(continued from previous page)

```
    0 |      5      5 |   5   5   5
    1 |      0      1 |   0   1   0
    2 |      2      2 |   2   2   2
    3 |      1      1 |   1   0   0
>>> del nl3[1:3]
>>> print(nl3)
      |     site      |    vector
index | origin target |   i   j   k
    0 |      5      5 |   5   5   5
    1 |      1      1 |   1   0   0
```

append(*vectors*)

Append one or more vectors to the NeighborList

```
>>> nl = NeighborList()
>>> print(nl)
      |     site      |    vector
index | origin target |   i   j   k
<BLANKLINE>
>>> nl.append([0,1,1,0,0])
>>> print(nl)
      |     site      |    vector
index | origin target |   i   j   k
    0 |      0      1 |   1   0   0
>>> nl.append([[0,1,0,1,0],[0,1,0,0,1]])
>>> print(nl)
      |     site      |    vector
index | origin target |   i   j   k
    0 |      0      1 |   1   0   0
    1 |      0      1 |   0   1   0
    2 |      0      1 |   0   0   1
```

values

Returns the neighbor vectors as a `numpy.ndarray`

This allows you to directly modify or set the vectors but be careful to maintain an (n, 5) array.

```
>>> nl = NeighborList([[0,1,1,0,0],[0,1,0,1,0],[0,1,0,0,1]])
>>> print(nl)
      |     site      |    vector
index | origin target |   i   j   k
    0 |      0      1 |   1   0   0
    1 |      0      1 |   0   1   0
    2 |      0      1 |   0   0   1
>>> nl.values
array([[0, 1, 1, 0, 0],
       [0, 1, 0, 1, 0],
       [0, 1, 0, 0, 1]])
>>> nl.values[:,1] = 2
>>> print(nl)
      |     site      |    vector
index | origin target |   i   j   k
    0 |      0      2 |   1   0   0
    1 |      0      2 |   0   1   0
    2 |      0      2 |   0   0   1
```

Returns array of neighbor vectors

Return type numpy.ndarray

4.9 structure

class javelin.structure.**Structure**(*symbols=None*, *numbers=None*, *unitcell=1*, *ncells=None*, *positions=None*, *rotations=False*, *translations=False*, *magnetic_moments=False*)

The structure class is made up of a **unitcell** and a list of **atoms**

Structure can be initialize using either another *javelin.structure.Structure*, ase.Atoms or diffpy.Structure.structure.Structure. It is recommended you use *javelin.structure.Structure.reindex()* after initializing from a foreign type in order to get the correct unitcell structure type.

Parameters

- **symbols** (*list*) – atoms symbols to initialize structure
- **numbers** (*list*) – atomic numbers to initialize structure
- **unitcell** (*javelin.unitcell.UnitCell*) – unitcell of structure, can be *javelin.unitcell.UnitCell* or values used to initialize the UnitCell
- **ncells** (*list*) – **ncells** has four components, (**i**, **j**, **k**, **n**) where **i**, **j**, **k** are the number of unitcell in each direction and **n** is the number of site positions in each unitcell. The product of **ncells** must equal the total number of atoms in the structure.
- **positions** (*3 x n array-like*) – array of atom coordinates

add_atom(*i=0*, *j=0*, *k=0*, *site=0*, *Z=None*, *symbol=None*, *position=None*)

Adds a single atom to the structure. It the atom exist as provided **i**, **j**, **k** and **site** it will be replaced.

Parameters

- **i** (*int*) – unitcell index
- **j** (*int*) – unitcell index
- **k** (*int*) – unitcell index
- **site** (*int*) – site index
- **Z** (*int*) – atomic number
- **symbol** (*int*) – chemical symbol
- **position** (*vector*) – position within the unitcell

```
>>> stru=Structure(numbers=[12],positions=[[0.,0.,0.]],unitcell=5.64)
>>> stru.atoms # doctest: +NORMALIZE_WHITESPACE
             Z symbol    x    y    z
i j k site
0 0 0 0     12     Mg  0  0  0
>>> stru.add_atom(Z=13, position=[0.,0.5,0.])
>>> stru.atoms # doctest: +NORMALIZE_WHITESPACE
             Z symbol    x    y    z
i j k site
0 0 0 0     13     Al  0...  0.5  0...
>>> stru.add_atom(Z=13, position=[0.5,0.,0.], i=1)
>>> stru.atoms # doctest: +NORMALIZE_WHITESPACE
```

(continues on next page)

(continued from previous page)

```
             Z symbol    x    y    z
i j k site
0 0 0 0     13     Al  0.0  0.5  0...
1 0 0 0     13     Al  0.5  0.0  0...
```

atoms = None

Attribute storing list of atom type and positions as a `pandas.DataFrame`

Example

```
>>> stru = Structure(symbols=['Na','Cl','Na'],positions=[[0,0,0],[0.5,0.5,0.
↪5],[0,1,0]])
>>> stru.atoms
             Z symbol    x    y    z
i j k site
0 0 0 0     11     Na  0.0  0.0  0.0
      1     17     Cl  0.5  0.5  0.5
      2     11     Na  0.0  1.0  0.0
```

element

Array of all elements in the structure

Returns array of element symbols

Return type `numpy.ndarray`

Example

```
>>> stru = Structure(symbols=['Na','Cl','Na'],positions=[[0,0,0],[0.5,0.5,0.
↪5],[0,1,0]])
>>> stru.element
array(['Na', 'Cl', 'Na'], dtype=object)
```

get_atom_Zs()

Get a list of unique atomic number in structure

Returns array of Zs

Return type `numpy.ndarray`

Example

```
>>> stru = Structure(symbols=['Na','Cl','Na'],positions=[[0,0,0],[0.5,0.5,0.
↪5],[0,1,0]])
>>> print(stru.get_atom_Zs())
[11 17]
```

get_atom_count()

Returns a count of each different type of atom in the structure

Returns series of atom count

Return type `pandas.Series`

Example

```
>>> stru = Structure(symbols=['Na','Na'],positions=[[0,0,0],[0.5,0.5,0.5]])
>>> stru.get_atom_count()
Na    2
Name: symbol, dtype: int64
```

get_atom_symbols()
Get a list of unique atom symbols in structure

Returns array of atom symbols

Return type numpy.ndarray

Example

```
>>> stru = Structure(symbols=['Na','Cl','Na'],positions=[[0,0,0],[0.5,0.5,0.
↪5],[0,1,0]])
>>> stru.get_atom_symbols()
array(['Na', 'Cl'], dtype=object)
```

get_atomic_numbers()
Array of all atomic numbers in the structure

Returns array of atomic numbers

Return type numpy.ndarray

Example

```
>>> stru = Structure(symbols=['Na','Cl','Na'],positions=[[0,0,0],[0.5,0.5,0.
↪5],[0,1,0]])
>>> print(stru.get_atomic_numbers())
[11 17 11]
```

get_average_site(*site=0*, *separate_site=True*)

get_average_structure(*separate_sites=True*)

get_cell()

get_celldisp()

get_chemical_formula()
Returns the chemical formula of the structure

Returns chemical formula

Return type str

Example

```
>>> stru = Structure(symbols=['Na','Cl','Na'],positions=[[0,0,0],[0.5,0.5,0.
↪5],[0,1,0]])
>>> stru.get_chemical_formula()
'Cl1Na2'
```

get_chemical_symbols()
Same as *javelin.structure.Structure.element*

get_displacement_correlation(*vectors*, *direction=(1, 1, 1)*, *direction2=None*)

Parameters **vectors** (*javelin.neighborlist.NeighborList* or *n x 5* array of neighbor vectors) – neighbor vectors

Returns displacement correlation

Return type float

get_magnetic_moments()

get_neighbors(*site=0*, *target_site=None*, *minD=0.01*, *maxD=1.1*)
Return a *javelin.neighborlist.NeighborList* for the given sites and distances

get_occupational_correlation(*vectors*, *atom*)

Parameters

- **vectors** (*javelin.neighborlist.NeighborList* or *n x 5* array of neighbor vectors) – neighbor vectors
- **atom** (*int*) – atom type for which to calculate correlation

Returns occupational correlation

Return type float

get_positions()
Same as *javelin.structure.Structure.xyz_cartn*

get_scaled_positions()
Array of all xyz positions in fractional lattice units of the atoms in the structure

Returns 3 x n array of atom positions

Return type numpy.ndarray

Example

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> stru.get_scaled_positions()
array([[ 0. ,  0. ,  0. ],
       [ 0.5,  0.5,  0.5]])
```

info
Dictionary of key-value pairs with additional information about the system.

Not implemented, only for ASE compatibility.

number_of_atoms
The total number of atoms in the structure

Returns number of atoms in structure

Return type int

Example

```
>>> stru = Structure(symbols=['Na','Cl','Na'],positions=[[0,0,0],[0.5,0.5,0.
↪5],[0,1,0]])
>>> stru.number_of_atoms
3
```

rattle(*scale=0.001*, *seed=None*)
Randomly move all atoms by a normal distbution with a standard deviation given by scale.

Parameters

- **scale** (*float*) – standard deviation
- **seed** (*int*) – seed for random number generator

Example

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> print(stru.xyz)
[[ 0.   0.   0. ]
 [ 0.5  0.5  0.5]]
>>> stru.rattle(seed=42)
>>> print(stru.xyz)
[[  4.96714153e-04  -1.38264301e-04   6.47688538e-04]
 [  5.01523030e-01   4.99765847e-01   4.99765863e-01]]
```

reindex (*ncells*)

This will reindex the list of atoms into the unitcell framework of this structure

ncells has four components, (**i**, **j**, **k**, **n**) where **i**, **j**, **k** are the number of unitcell in each direction and **n** is the number of site positions in each unitcell. The product of **ncells** must equal the total number of atoms in the structure.

Example

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> stru.atoms # doctest: +NORMALIZE_WHITESPACE
             Z symbol    x    y    z
i j k site
0 0 0 0     11     Na  0.0  0.0  0.0
      1     17     Cl  0.5  0.5  0.5
>>> stru.reindex([2,1,1,1])
>>> stru.atoms # doctest: +NORMALIZE_WHITESPACE
             Z symbol    x    y    z
i j k site
0 0 0 0     11     Na  0.0  0.0  0.0
1 0 0 0     17     Cl  0.5  0.5  0.5
```

repeat (*rep*)

Repeat the cells a number of time along each dimension

rep argument should be either three value like *(1,2,3)* or a single value *r* equivalent to *(r,r,r)*.

Parameters **rep** (*1 or 3 ints*) – repeating rate

Examples

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> print(stru.element)
['Na' 'Cl']
>>> print(stru.xyz_cartn)
[[ 0.    0.    0.  ]
 [ 2.82  2.82  2.82]]
>>> stru.repeat((2,1,1))
>>> print(stru.element)  # doctest: +ALLOW_UNICODE
['Na' 'Cl' 'Na' 'Cl']
>>> print(stru.xyz_cartn)
[[  0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  2.82000000e+00   2.82000000e+00   2.82000000e+00]
 [  5.64000000e+00   9.06981174e-16   9.06981174e-16]
 [  8.46000000e+00   2.82000000e+00   2.82000000e+00]]
```

```
>>> stru = Structure(symbols=['Na'], positions=[[0,0,0]], unitcell=5.64)
>>> print(stru.element)
['Na']
>>> print(stru.xyz_cartn)
[[ 0.  0.  0.]]
>>> stru.repeat(2)
>>> print(stru.element)  # doctest: +ALLOW_UNICODE
['Na' 'Na' 'Na' 'Na' 'Na' 'Na' 'Na' 'Na']
>>> print(stru.xyz_cartn)
[[  0.00000000e+00   0.00000000e+00   0.00000000e+00]
 [  0.00000000e+00   0.00000000e+00   5.64000000e+00]
 [  0.00000000e+00   5.64000000e+00   3.45350397e-16]
 [  0.00000000e+00   5.64000000e+00   5.64000000e+00]
 [  5.64000000e+00   9.06981174e-16   9.06981174e-16]
 [  5.64000000e+00   9.06981174e-16   5.64000000e+00]
 [  5.64000000e+00   5.64000000e+00   1.25233157e-15]
 [  5.64000000e+00   5.64000000e+00   5.64000000e+00]]
```

replace_atom(*to_replace: int*, *value: int*) → None

Replace all atoms in the structure that has Z='to_replace' with Z='value'. This uses `pandas.DataFrame.replace()` to replace the atom Z values

Parameters

- **to_replace** (*int*) – Z value to replace
- **value** (*int*) – what it is going to be replaced with

Example

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> print(stru.get_atom_count())
Cl    1
Na    1
Name: symbol, dtype: int64
>>> stru.replace_atom(11, 111)
>>> print(stru.get_atom_count())
Cl    1
Rg    1
Name: symbol, dtype: int64
```

to_ase()

unitcell = None

Attribute containing the unitcell of the structure. Must be of type *javelin.unitcell.UnitCell*

update_atom_symbols()

This will update the atom symbol list from the Z numbers, this should be run if the Z numbers are modified directly

x

Array of all x positions of in fractional lattice units of the atoms within the unitcell of the structure

Returns array of atom x positions

Return type `numpy.ndarray`

Example

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> stru.x
array([ 0. ,  0.5])
```

xyz

Array of all xyz positions in fractional lattice units of the atoms within the unitcell of the structure

Returns 3 x n array of atom positions

Return type numpy.ndarray

Example

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> stru.xyz
array([[ 0. ,  0. ,  0. ],
       [ 0.5,  0.5,  0.5]])
```

xyz_cartn

Array of all xyz positions in cartesian coordinates of the atoms in the structure

Returns 3 x n array of atom positions

Return type numpy.ndarray

Example

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> stru.xyz_cartn
array([[ 0.  ,  0.  ,  0.  ],
       [ 2.82,  2.82,  2.82]])
```

y

Array of all y positions of in fractional lattice units of the atoms within the unitcell of the structure

Returns array of atom y positions

Return type numpy.ndarray

Example

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> stru.y
array([ 0. ,  0.5])
```

z

Array of all z positions of in fractional lattice units of the atoms within the unitcell of the structure

Returns array of atom z positions

Return type numpy.ndarray

Example

```
>>> stru = Structure(symbols=['Na', 'Cl'], positions=[[0,0,0],[0.5,0.5,0.5]],␣
↪unitcell=5.64)
>>> stru.z
array([ 0. ,  0.5])
```

javelin.structure.**axisAngle2Versor**(*x*, *y*, *z*, *angle*, *unit='degrees'*)

javelin.structure.**get_miindex**(*length=0*, *ncells=None*)

javelin.structure.**get_rotation_matrix**(*l*, *m*, *n*, *theta*, *unit='degrees'*)

javelin.structure.**get_rotation_matrix_from_versor**(*w*, *x*, *y*, *z*)

4.10 unitcell

class javelin.unitcell.**UnitCell**(**args*)

The UnitCell object can be set with either 1, 3 or 6 parameters corresponding to cubic `a` parameters, `(a, b, c)` or `(a, b, c, alpha, beta, gamma)`, where angles are in degrees.

```
>>> cubic = UnitCell(5)
>>> cubic.cell
(5.0, 5.0, 5.0, 90.0, 90.0, 90.0)
```

```
>>> orthorhombic = UnitCell(5, 6, 7)
>>> orthorhombic.cell
(5.0, 6.0, 7.0, 90.0, 90.0, 90.0)
```

```
>>> unitcell = UnitCell(4.0, 3.0, 6.0, 89.0, 90.0, 97.0)
>>> unitcell.cell
(4.0, 3.0, 6.0, 89.0, 90.0, 97.0)
```

UnitCell objects can be set after being created simply by

```
>>> unitcell = UnitCell()
>>> unitcell.cell = 6
>>> unitcell.cell
(6.0, 6.0, 6.0, 90.0, 90.0, 90.0)
>>> unitcell.cell = 3, 4, 5
>>> unitcell.cell
(3.0, 4.0, 5.0, 90.0, 90.0, 90.0)
>>> unitcell.cell = 6, 7, 8, 91.0, 90, 89
>>> unitcell.cell
(6.0, 7.0, 8.0, 91.0, 90.0, 89.0)
>>> # or using a list or tuple
>>> unitcell.cell = [8, 7, 6, 89, 90, 90]
>>> unitcell.cell
(8.0, 7.0, 6.0, 89.0, 90.0, 90.0)
```

B

Returns the **B** matrix

Binv

Returns the inverse **B** matrix

G

Returns the metric tensor **G**

Gstar

Returns the reciprocal metric tensor **G***

cartesian(*u*)

Return Cartesian coordinates of a lattice vector.

```
>>> unitcell = UnitCell(3,4,5,90,90,120)
>>> unitcell.cartesian([1,0,0])
array([  2.59807621e+00,  -1.50000000e+00,   3.25954010e-16])
```

A array of atoms position can also be passed

```
>>> positions = [[1,0,0], [0,0,0.5]]
>>> unitcell.cartesian(positions)
array([[  2.59807621e+00,  -1.50000000e+00,   3.25954010e-16],
       [  0.00000000e+00,   0.00000000e+00,   2.50000000e+00]])
```

cell
Return the unit cell parameters (*a*, *b*, *c*, *alpha*, *beta*, *gamma*) in degrees.

d(*h*, *k*, *l*)
Returns d-spacing for given h,k,l

dstar(*h*, *k*, *l*)
Returns d*=1/d for given h,k,l

fractional(*u*)
Return Cartesian coordinates of a lattice vector.

```
>>> unitcell = UnitCell(3,4,5,90,90,120)
>>> unitcell.fractional([0,4,0])
array([  0.00000000e+00,   1.00000000e+00,  -4.89858720e-17])
```

A array of atoms position can also be passed

```
>>> positions = [[0,2,0], [0,0,5]]
>>> unitcell.fractional(positions)
array([[  0.00000000e+00,   5.00000000e-01,  -2.44929360e-17],
       [  0.00000000e+00,   0.00000000e+00,   1.00000000e+00]])
```

recAngle(*h1*, *k1*, *l1*, *h2*, *k2*, *l2*, *degrees=False*)
Calculates the angle between two reciprocal vectors

reciprocalCell
Return the reciprocal unit cell parameters (*a**, *b**, *c**, *alpha**, *beta**, *gamma**) in degrees.

reciprocalVolume
Returns the unit cell reciprocal volume

volume
Returns the unit cell volume

4.11 utils

javelin.utils.**get_atomic_number_symbol**(*Z=None*, *symbol=None*)
This function returns a tuple of matching arrays of atomic numbers (Z) and chemical symbols (symbol).

Parameters

- **Z** (*int, array like object of int's*) – atomic numbers
- **symbol** (*str, array like object of str*) – chemical symbols

Returns arrays of atomic numbers and chemical symbols

Return type tuple of `numpy.ndarray`

Note: If both Z and symbol are provided the symbol will win out and change the Z to match.

Examples

```
>>> Z, symbol = get_atomic_number_symbol(Z=[12, 24, 26, 48])
>>> print(Z)
[12 24 26 48]
>>> print(symbol)  # doctest: +ALLOW_UNICODE
['Mg' 'Cr' 'Fe' 'Cd']
```

```
>>> Z, symbol = get_atomic_number_symbol(symbol=['C', 'H', 'N', 'O'])
>>> print(Z)
[6 1 7 8]
>>> print(symbol)
['C' 'H' 'N' 'O']
```

`javelin.utils.`**`get_atomic_numbers`**(*structure*)
Wrapper to get the atomic numbers from different structure classes

`javelin.utils.`**`get_positions`**(*structure*)
Wrapper to get the positions from different structure classes

`javelin.utils.`**`get_unitcell`**(*structure*)
Wrapper to get the unit cell from different structure classes

`javelin.utils.`**`is_structure`**(*structure*)
Check if an object is a structure that javelin can understand.

ase.atoms with have cell, get_scaled_positions and get_atomic_numbers attributes diffpy.structure with have lattice, xyz, and element attributes

Python Module Index

j

Index

G

I

J

K

L

M

N

R

S

T

U

V

X

Y

Z