ipyMD Documentation

Release 0.4.2

Chris Sewell

January 24, 2017

Contents

This package aims to provide a means of producing **reusable** analysis of Molecular Dynamics (MD) output in the IPython Notebook.

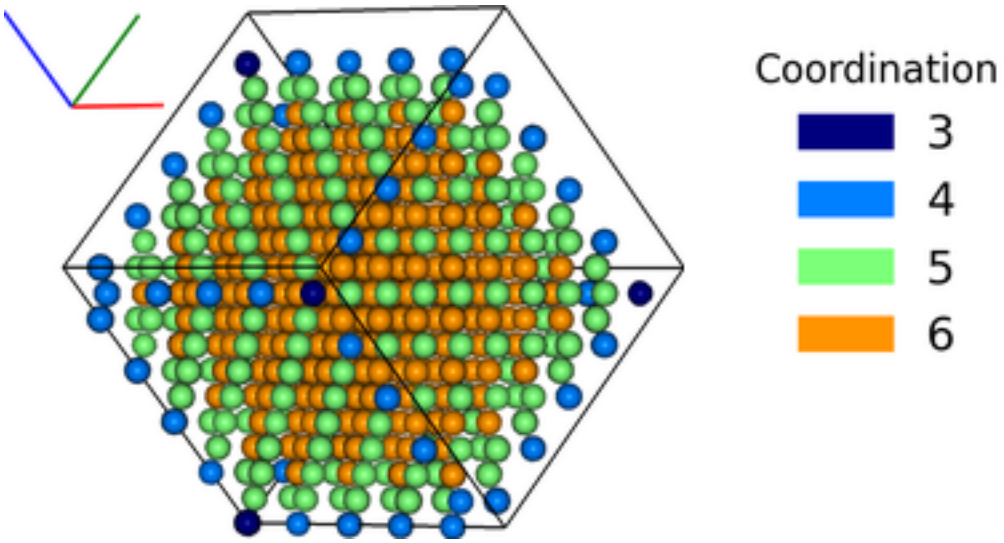


Fig. 1: Analysis of the atomic coordination of Na, wrt Cl, for an NaCl nano-crystal.

There are many programs for 3D visualisation of MD output (my favourite being Ovito). However, there lacks a means to produce a more thorough, documented analysis of the data. IPython Notebooks are ideal for this type of analysis and so the objective of *ipymd* is to produce a Python package that can be used in conjuction with programmes like Ovito, to produce documented and reuseable analysis.

The aim of *ipymd* is to produce IPython Notebooks that include:

- Static images of the simulations
- Analysis of simulation data, including graphical plots

It has been created with the goal to be:

- Easy to use
- Easy to extend

chemlab It builds primarily on the chemlab package, that is an API layer on top of OpenGL. Data is parsed in standard formats, such as [pandas](http://pandas.pydata.org/) dataframes, which are easy to create and use independantly from this package, in order to extend its functionality.

Author	Chris Sewell
Project Page	https://github.com/chrisjsewell/ipymd

CHAPTER 1

Contents

1.1 Quick Start

Anaconda is recommended to create a Python environment within which to use ipymd:

```
conda create -n ipymd -c cjs14 ipymd
source activate ipymd
jupyter notebook
```

Currently the conda package is only available for OSX. For other operating systems, or to use the latest version from Github, the following environment should work:

```
conda create -n ipymd python=2.7.11=0 numpy scipy matplotlib pandas ipython ipython-notebook pillow p
```

If there are any issues, see the known working package dependancies list: https://github.com/chrisjsewell/ipymd/blob/master/working_dependencies_list_osx.txt

1.2 User Tutorial

In the IPython Notebook, the ipymd package must first be imported:

```
import ipymd
print ipymd.version()
```

```
0.4.2
```

1.2.1 Basic Atom Creation and Visualisation

The input for a basic atomic visualisation, is a pandas Dataframe that specifies the coordinates, size and color of each atom in the following manner:

```
import pandas as pd
df = pd.DataFrame(
        [[2,3,4,1,[0, 0, 255],1],
         [1,3,3,1,'orange',1],
         [4,3,1,1,'blue',1]],
        columns=['x','y','z','radius','color','transparency'])
```

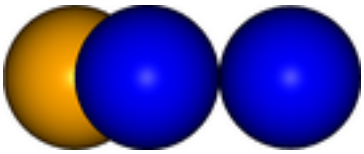
Distances are measured in Angstroms, and colors can be defined in [r,g,b] format (0 to 255) or as a string defined in `available_colors`.

```
print(ipymd.available_colors()['reds'])
```

```
['light_salmon', 'salmon', 'dark_salmon', 'light_coral', 'indian_red', 'crimson', 'fire_brick', 'dark
```

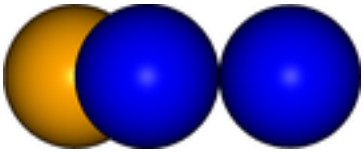
The `Visualise_Sim` class can then be used to setup a visualisation, which is returned in the form of a `PIL` image.

```
vis = ipymd.visualise_sim.Visualise_Sim()
vis.add_atoms(df)
img1 = vis.get_image(size=400,quality=5)
img1
```



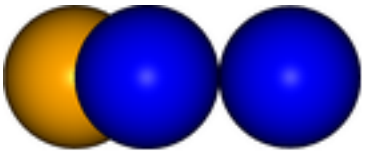
To convert this into an image viewable in IPython, simply parse it to the `visualise` function.

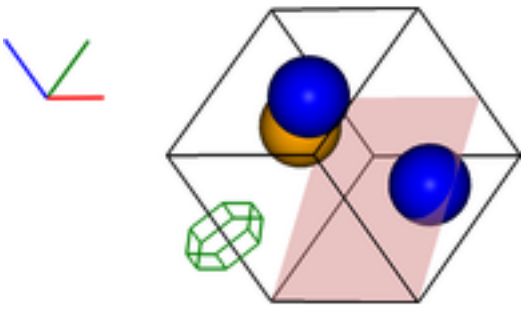
```
vis.visualise(img1)
```



Extending this basic procedure, additional objects can be added to the visualisation, the viewpoint can be rotated and multiple images can be output at once, as shown in the following example:

```
vis.add_axes(length=0.2, offset=(-0.3,0))
vis.add_box([5,0,0],[0,5,0],[0,0,5])
vis.add_plane([[5,0,0],[0,5,2]],alpha=0.3)
vis.add_hexagon([[1,0,0],[0,0,.5]],[0,0,2],color='green')

img_ex1 = vis.get_image(xrot=45, yrot=45)
#img2 = vis.draw_colorlegend(img2,1,2)
vis.visualise([img1,img_ex1])
```





1.2.2 Atom Creation From Other Sources

The `ipymd.data_input` module includes a number of classes to automate the intial creation of the atoms Dataframe, from various sources.

All classes will return a sub-class of `DataInput`, that requires the `setup_data` method to be run first, and then the `get_atoms` method returns the atoms Dataframe and the `get_meta_data` method returns a Pandas Series (which includes the vertexes and origin of the simulation box).

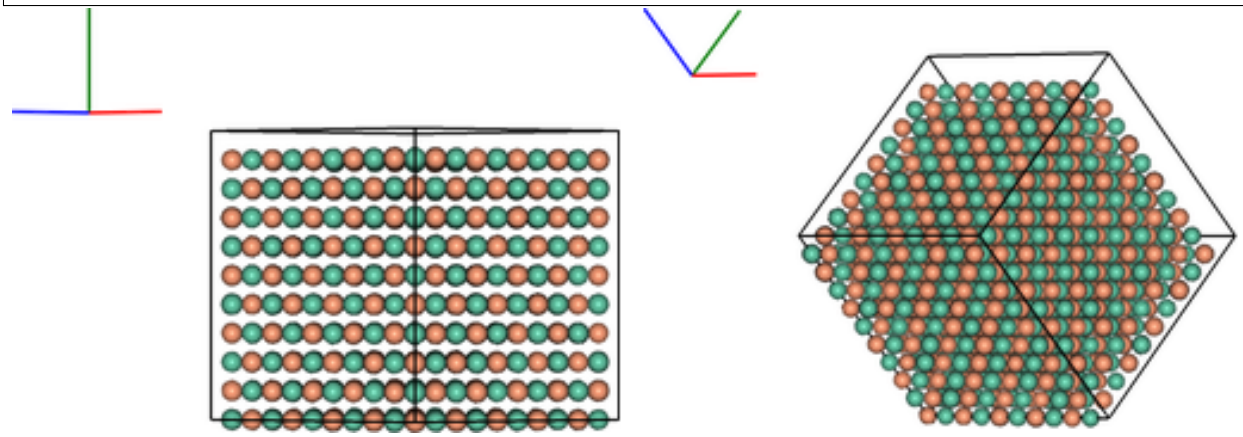
Crystal Parameters

This class allows atoms to be created in ordered crystal, as defined by their space group and crystal parameters:

```
data = ipymd.data_input.crystal.Crystal()
data.setup_data(
    [[0.0, 0.0, 0.0], [0.5, 0.5, 0.5]], ['Na', 'Cl'],
    225, cellpar=[5.4, 5.4, 5.4, 90, 90, 90],
    repetitions=[5, 5, 5])
meta = data.get_meta_data()
print meta
atoms_df = data.get_atom_data()
atoms_df.head(2)
```

```
origin                                    (0.0, 0.0, 0.0)
a                                        (27.0, 0.0, 0.0)
b                          (1.65327317885e-15, 27.0, 0.0)
c           (1.65327317885e-15, 1.65327317885e-15, 27.0)
dtype: object
```

```
vis2 = ipymd.visualise_sim.Visualise_Sim()
vis2.add_axes()
vis2.add_box(meta.a,meta.b,meta.c, meta.origin)
vis2.add_atoms(atoms_df)
images = [vis2.get_image(xrot=xrot,yrot=45) for xrot in [0,45]]
vis2.visualise(images, columns=2)
```



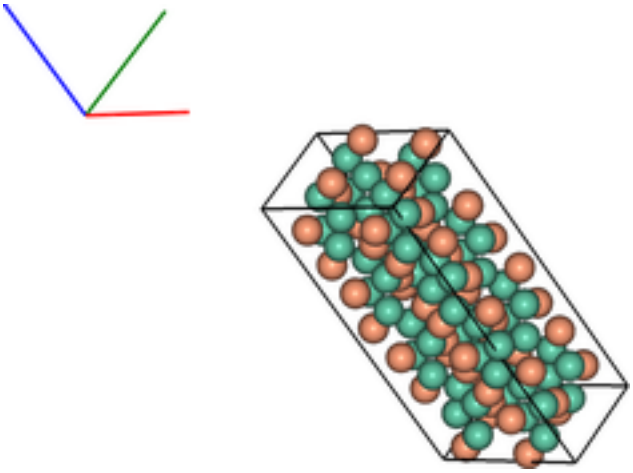
A dataframe is available which lists the alternative names for each space group:

```
df = ipymd.data_input.crystal.get_spacegroup_df()
df.loc[[1,194,225]]
```

Crystallographic Information Files

.cif files are a common means to store crystallographic data and can be loaded as follows:

```
cif_path = ipymd.get_data_path('example_crystal.cif')
data = ipymd.data_input.cif.CIF()
data.setup_data(cif_path)
meta = data.get_meta_data()
vis = ipymd.visualise_sim.Visualise_Sim()
vis.basic_vis(data.get_atom_data(), data.get_meta_data(),
              xrot=45,yrot=45)
```



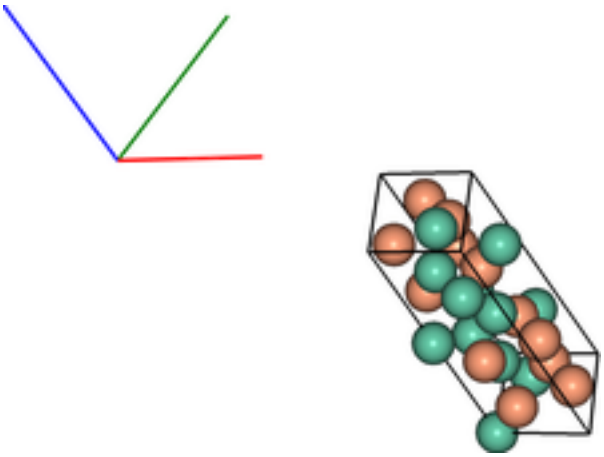
NB: at present, fractional occupancies of lattice sites are returned in the atom Dataframe, but cannot be visualised as such. It is intended that eventually occupancy will be visualised by partial spheres.

```
data.get_atom_data().head(1)
```

Lammps Input Data

The input data for LAMMPS simulations (supplied to `read_data`) can be input. Note that the `get_atom_data` method requires that the atom_style is defined, in order to define what each data column refers to.

```
lammps_path = ipymd.get_data_path('lammps_input.data')
data = ipymd.data_input.lammps.LAMMPS_Input()
data.setup_data(lammps_path,atom_style='charge')
vis = ipymd.visualise_sim.Visualise_Sim()
vis.basic_vis(data.get_atom_data(), data.get_meta_data(),
              xrot=45,yrot=45)
```

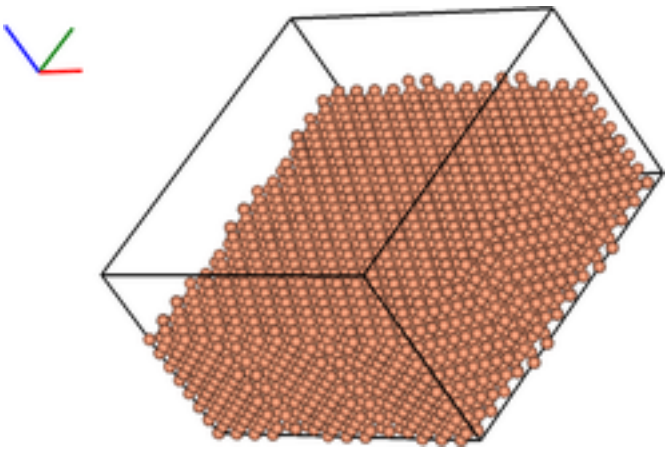


Lammps Output Data

Output data can be read in the form of a single file or, it is advisable for efficiency, that a single file is output for each timestep, where * is used to define the variable section of the filename. The `get_atoms` and `get_simulation_box` methods now take a variable to define which configuration is returned.

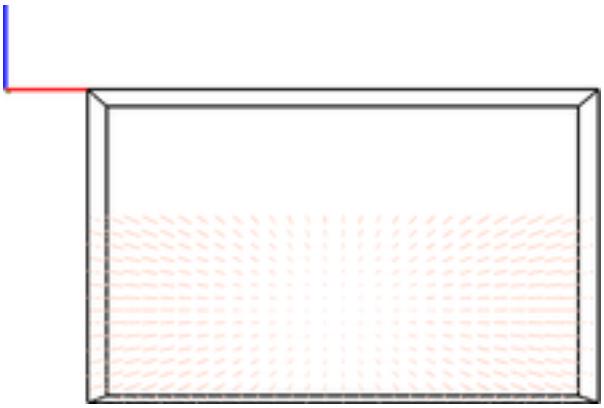
```
lammps_path = ipymd.get_data_path('atom_onefile.dump')
data = ipymd.data_input.lammps.LAMMPS_Output()
data.setup_data(lammps_path)
print data.count_configs()

vis = ipymd.visualise_sim.Visualise_Sim()
vis.basic_vis(data.get_atom_data(99), data.get_meta_data(99),
              spheres=True,xrot=45,yrot=45,quality=5)
```

```
99
```



```
lammps_path = ipymd.get_data_path(['atom_dump','atoms_*.dump'])
data = ipymd.data_input.lammps.LAMMPS_Output()
data.setup_data(lammps_path)
print data.count_configs()

vis = ipymd.visualise_sim.Visualise_Sim()
vis.basic_vis(data.get_atom_data(99), data.get_meta_data(99),
              spheres=False,xrot=90,yrot=0)
```

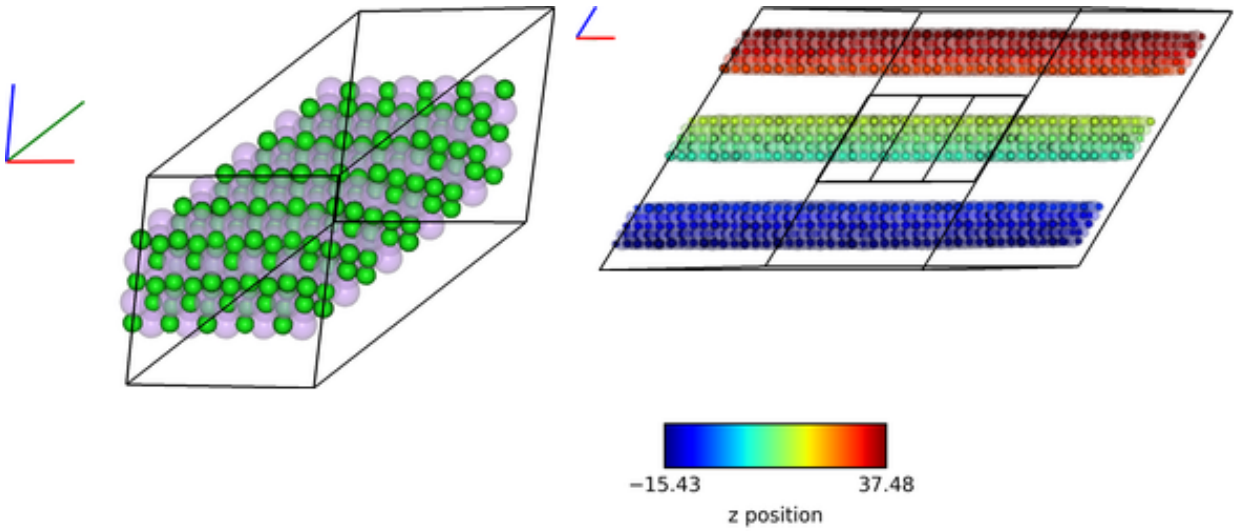
```
99
```



1.2.3 Atom Manipulation

The atoms Dataframe is already very easy to manipulate using the standard pandas methods. But an `Atom_Manipulation` class has also been created to carry out standard atom manipulations, such as setting variables dependant on atom type or altering the geometry, as shown in this example:

```
data = ipymd.data_input.crystal.Crystal()
data.setup_data(
    [[0.0, 0.0, 0.0], [0.5, 0.5, 0.5]], ['Na', 'Cl'],
    225, cellpar=[5.4, 5.4, 5.4,60,60,60],
    repetitions=[5, 5, 5])
meta = data.get_meta_data()

manipulate_atoms = ipymd.atom_manipulation.Atom_Manipulation

new_df = manipulate_atoms(data.get_atom_data(), data.get_meta_data(),undos=2)

new_df.apply_map({'Na':1.5, 'Cl':1},'radius')
new_df.apply_map('color','color',default='grey')
new_df.change_type_variable('Na', 'transparency', 0.5)
new_df.slice_fraction(cmin=0.3, cmax=0.7,update_uc=False)

vis2 = ipymd.visualise_sim.Visualise_Sim()
vis2.add_box(*new_df.meta[['a','b','c','origin']])
vis2.add_axes([meta.a,meta.b,meta.c],offset=(-1.3,-0.7))
vis2.add_atoms(new_df.df, spheres=True)

img1 = vis2.get_image(xrot=45,yrot=45)

vis2.remove_atoms()
new_df.repeat_cell((-1,1),(-1,1),(-1,1))
new_df.color_by_variable('z')
vis2.add_atoms(new_df.df, spheres=True)
vis2.add_box(*new_df.meta[['a','b','c','origin']])
img2 = vis2.get_image(xrot=90,yrot=0)
img3 = ipymd.plotting.create_colormap_image(new_df.df.z.min(), new_df.df.z.max(),
                                            horizontal=True,title='z position',size=150)

vis2.visualise([img1,img2, (280,1), img3], columns=2)
```



NB: default atom variables (such as color and radii can be set using the `apply_map` method and any column name from those given in `ipymd.shared.atom_data()`:

```
ipymd.shared.atom_data().head(1)
```

1.2.4 Geometric Analysis

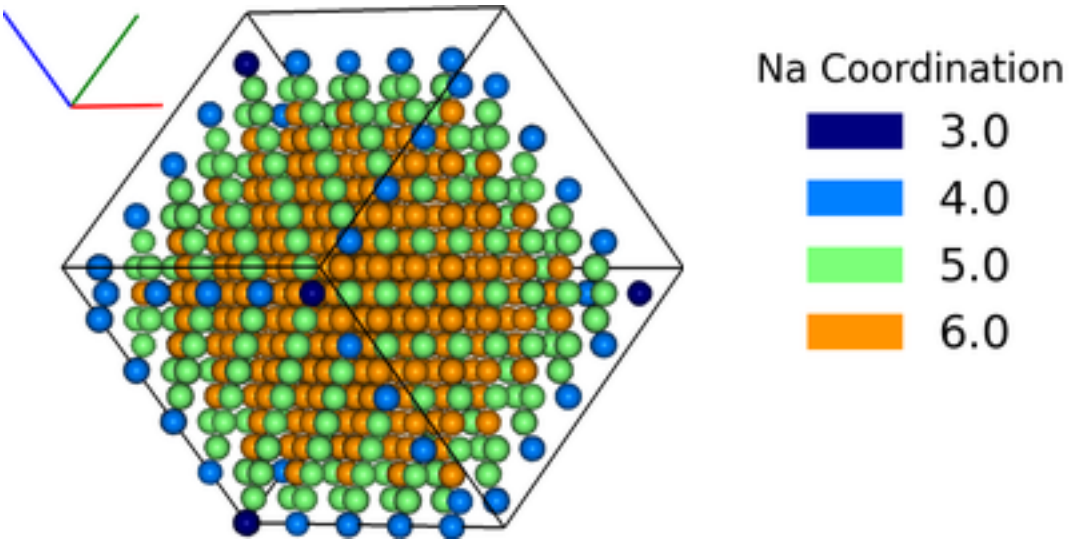
Given the simple and flexible form of the atomic data and visualisation, it is now easier to add more complex geometric analysis. These analyses are being contained in the `atom_analysis` package, and some initial examples are detailed below. Functions in the `atom_analysis.nearest_neighbour` package are based on the `scipy.spatial.cKDTree` algorithm for identifying nearest neighbours.

Atomic Coordination

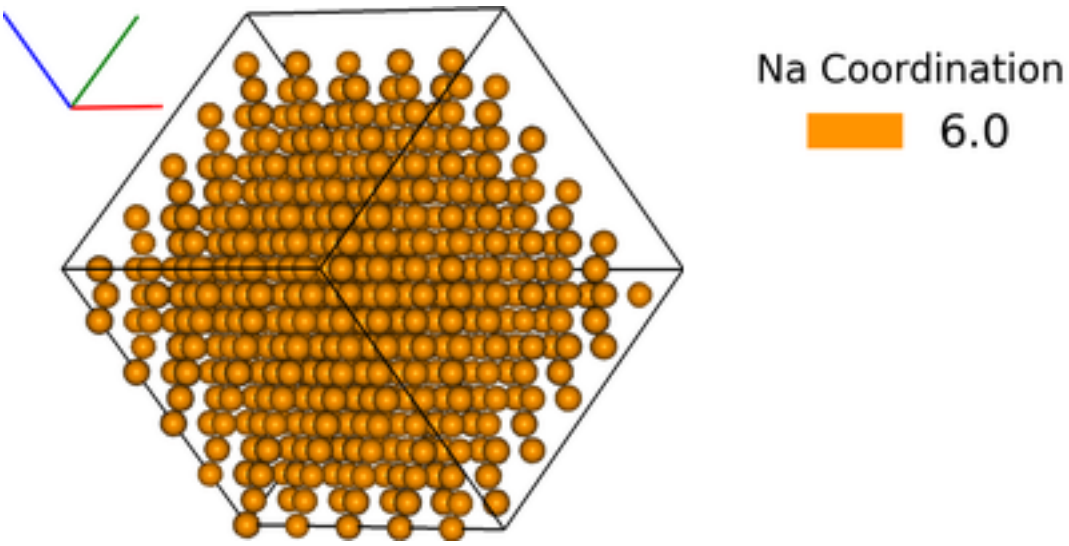
The two examples below show computation of the coordination of Na, w.r.t Cl, in a simple NaCl crystal (which should be 6). The first does not include a consideration of the repeating boundary conditions, and so outer atoms have a lower coordination number. But the latter computation provides a method which takes this into consideration, by repeating the Cl lattice in each direction before computation.

```
data = ipymd.data_input.crystal.Crystal()
data.setup_data(
    [[0.0, 0.0, 0.0], [0.5, 0.5, 0.5]], ['Na', 'Cl'],
    225, cellpar=[5.4, 5.4, 5.4, 90, 90, 90],
    repetitions=[5, 5, 5])
df = data.get_atom_data()
meta = data.get_meta_data()

df = ipymd.atom_analysis.nearest_neighbour.coordination_bytype(df, 'Na','Cl')

new_df = manipulate_atoms(df,meta)
new_df.filter_variables('Na')
new_df.color_by_variable('coord_Na_Cl',minv=3,maxv=7)

vis = ipymd.visualise_sim.Visualise_Sim()
vis.add_axes(offset=(0,-0.3))
vis.add_box(*meta[['a','b','c','origin']])
```

```
vis.add_atoms(new_df.df)

img = vis.get_image(xrot=45,yrot=45)

img2 = ipymd.plotting.create_legend_image(new_df.df.coord_Na_Cl,new_df.df.color, title='Na Coordinati

vis.visualise([img,img2],columns=2)
```




```
data = ipymd.data_input.crystal.Crystal()
data.setup_data(
    [[0.0, 0.0, 0.0], [0.5, 0.5, 0.5]], ['Na', 'Cl'],
    225, cellpar=[5.4, 5.4, 5.4, 90, 90, 90],
    repetitions=[5, 5, 5])
df = data.get_atom_data()
meta = data.get_meta_data()

df = ipymd.atom_analysis.nearest_neighbour.coordination_bytype(
    df, 'Na','Cl',repeat_meta=meta)

new_df = manipulate_atoms(df)
new_df.filter_variables('Na')
new_df.color_by_variable('coord_Na_Cl',minv=3,maxv=7)

vis = ipymd.visualise_sim.Visualise_Sim()
vis.add_box(*meta[['a','b','c','origin']])
vis.add_axes(offset=(0,-0.3))
vis.add_atoms(new_df.df)

img = vis.get_image(xrot=45,yrot=45)

img2 = ipymd.plotting.create_legend_image(new_df.df.coord_Na_Cl,new_df.df.color, title='Na Coordinati

vis.visualise([img,img2],columns=2)
```



Atomic Structure Comparison

`compare_to_lattice` takes each atomic coordinate in df1 and computes the distance to the nearest atom (i.e. lattice site) in df2:

```
import numpy as np
data1 = ipymd.data_input.crystal.Crystal()
data1.setup_data(
    [[0.0, 0.0, 0.0], [0.5, 0.5, 0.5]], ['Na', 'Cl'],
    225, cellpar=[5.4, 5.4, 5.4, 90, 90, 90],
    repetitions=[5, 5, 5])
df1 = data1.get_atom_data()

print(('Average distance to nearest atom (identical)',
       np.mean(ipymd.atom_analysis.nearest_neighbour.compare_to_lattice(df1,df1))))

data2 = ipymd.data_input.crystal.Crystal()
data2.setup_data(
    [[0.0, 0.0, 0.0], [0.5, 0.5, 0.5]], ['Na', 'Cl'],
    225, cellpar=[5.41, 5.4, 5.4, 90, 90, 90],
    repetitions=[5, 5, 5])
df2 = data2.get_atom_data()

print(('Average distance to nearest atom (different)',
       np.mean(ipymd.atom_analysis.nearest_neighbour.compare_to_lattice(df1,df2))))
```

```
('Average distance to nearest atom (identical)', 0.0)
('Average distance to nearest atom (different)', 0.022499999999999343)
```

Common Neighbour Analysis (CNA)

CNA (Honeycutt and Andersen, J. Phys. Chem. 91, 4950) is an algorithm to compute a signature for pairs of atoms, which is designed to characterize the local structural environment. Typically, CNA is used as an effective filtering method to classify atoms in crystalline systems (Faken and Jonsson, Comput. Mater. Sci. 2, 279, with the goal to get a precise understanding of which atoms are associated with which phases, and which are associated with defects.

Common signatures for nearest neighbours are:

- FCC = 12 x 4,2,1
- HCP = 6 x 4,2,1 & 6 x 4,2,2
- BCC = 6 x 6,6,6 & 8 x 4,4,4
- Diamond = 12 x 5,4,3 & 4 x 6,6,3

which are tested below:

```
data = ipymd.data_input.crystal.Crystal()
data.setup_data(
    [[0.0, 0.0, 0.0]], ['Al'],
    225, cellpar=[4.05, 4.05, 4.05, 90, 90, 90],
    repetitions=[5, 5, 5])
fcc_meta = data.get_meta_data()
fcc_df = data.get_atom_data()

data = ipymd.data_input.crystal.Crystal()
data.setup_data(
    [[0.33333,0.66667,0.25000]], ['Mg'],
    194, cellpar=[3.21, 3.21, 5.21, 90, 90, 120],
    repetitions=[5,5,5])
hcp_meta = data.get_meta_data()
hcp_df = data.get_atom_data()

data = ipymd.data_input.crystal.Crystal()
data.setup_data(
    [[0,0,0]], ['Fe'],
    229, cellpar=[2.866, 2.866, 2.866, 90, 90, 90],
    repetitions=[5,5,5])
bcc_meta = data.get_meta_data()
bcc_df = data.get_atom_data()

data = ipymd.data_input.crystal.Crystal()
data.setup_data(
    [[0,0,0]], ['C'],
    227, cellpar=[3.57, 3.57, 3.57, 90, 90, 90],
    repetitions=[2,2,2])
diamond_meta = data.get_meta_data()
diamond_df = data.get_atom_data()
```

```
print(ipymd.atom_analysis.nearest_neighbour.cna_sum(fcc_df,repeat_meta=fcc_meta))
print(ipymd.atom_analysis.nearest_neighbour.cna_sum(hcp_df,repeat_meta=hcp_meta))
print(ipymd.atom_analysis.nearest_neighbour.cna_sum(bcc_df,repeat_meta=bcc_meta))
print(ipymd.atom_analysis.nearest_neighbour.cna_sum(diamond_df,upper_bound=10,max_neighbours=16,
                                                    repeat_meta=diamond_meta))
```

```
Counter({'4,2,1': 6000})
Counter({'4,2,2': 1500, '4,2,1': 1500})
Counter({'6,6,6': 2000, '4,4,4': 1500})
Counter({'5,4,3': 768, '6,6,3': 256})
```

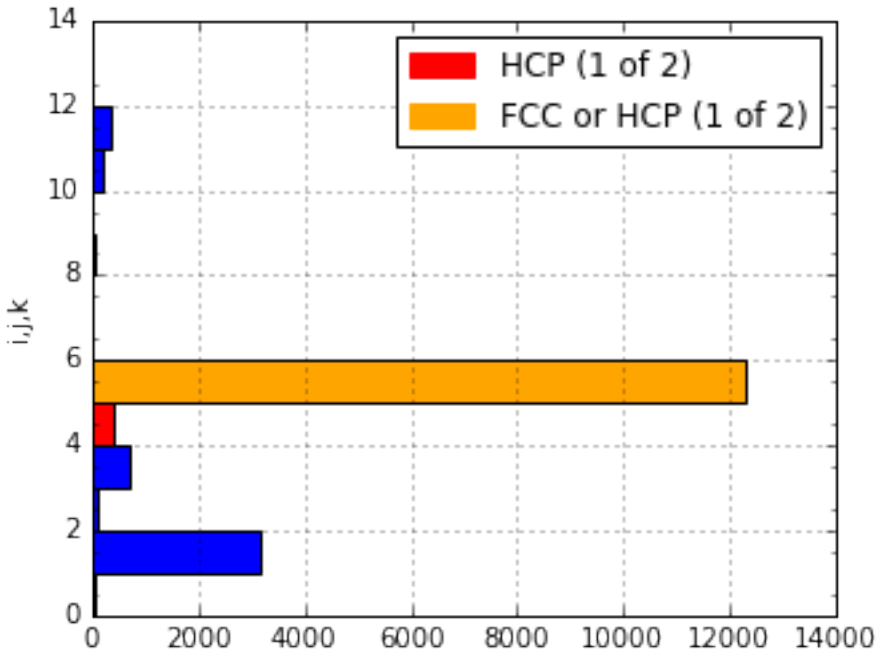
For each atom, the CNA for each nearest-neighbour can be output:

```
ipymd.atom_analysis.nearest_neighbour.common_neighbour_analysis(hcp_df,repeat_meta=hcp_meta).head(5)
```

This can be used to produce a plot identifying likely structure of an unknown structure:

```
lammps_path = ipymd.get_data_path('thermalized_troilite.dump')
data = ipymd.data_input.lammps.LAMMPS_Output()
```

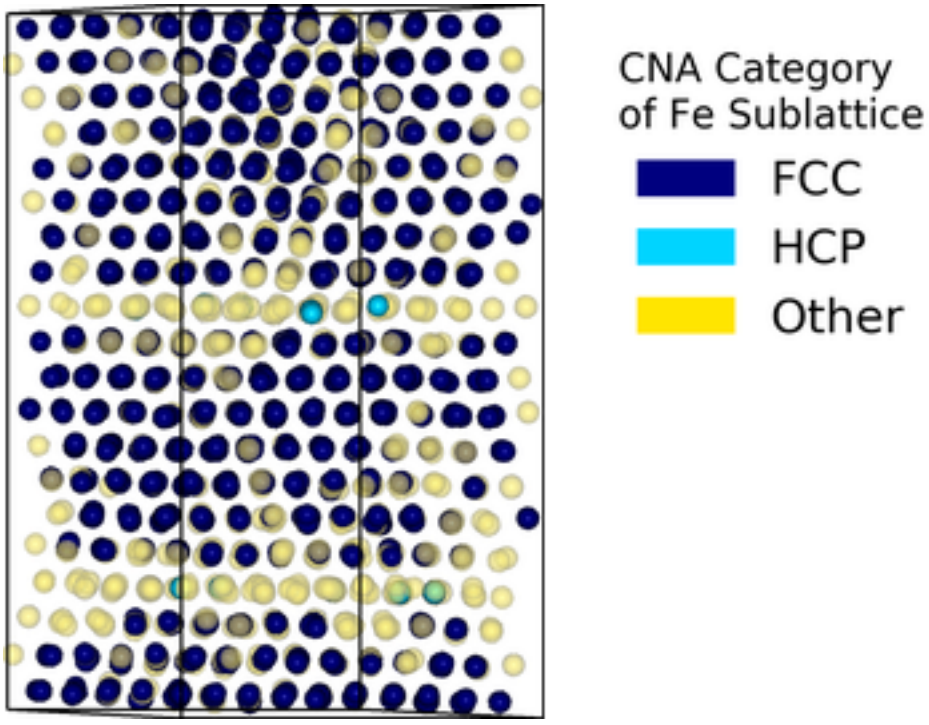
```
data.setup_data(lammps_path)
df = data.get_atom_data()
df = df[df.type==1]
plot = ipymd.atom_analysis.nearest_neighbour.cna_plot(df,repeat_meta=data.get_meta_data())
plot.display_plot()
```



A visualisation of the probable local character of each atom can also be created. Note the *accuracy* parameter in the `cna_categories` method allows for more robust fitting to the ideal signatures:

```
lammps_path = ipymd.get_data_path('thermalized_troilite.dump')
data = ipymd.data_input.lammps.LAMMPS_Output()
data.setup_data(lammps_path)
df = data.get_atom_data()
meta = data.get_meta_data()
df = df[df.type==1]
df = ipymd.atom_analysis.nearest_neighbour.cna_categories(
    df,repeat_meta=meta,accuracy=0.7)
manip = ipymd.atom_manipulation.Atom_Manipulation(df)
manip.color_by_categories('cna')
#manip.apply_colormap({'Other':'blue','FCC':'green','HCP':'red'}, type_col='cna')
manip.change_type_variable('Other','transparency',0.5,type_col='cna')
atom_df = manip.df

vis = ipymd.visualise_sim.Visualise_Sim()
vis.add_box(*meta[['a','b','c','origin']])
vis.add_atoms(atom_df)

img = vis.get_image(xrot=90)

img2 = ipymd.plotting.create_legend_image(atom_df.cna,atom_df.color,
                title='CNA Category\nof Fe Sublattice',size=150,colbytes=True)
```

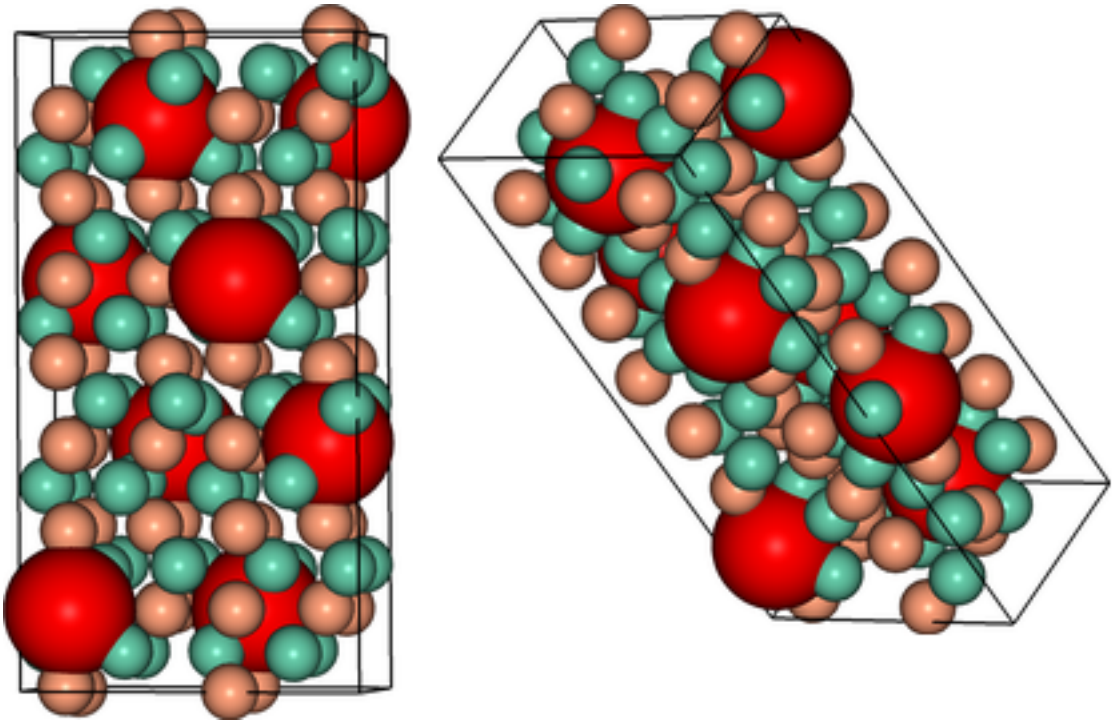
```
vis.visualise([img,img2],columns=2)
```



Vacany Identification

The `vacancy_identification` method finds grid sites with no atoms within a specified distance:

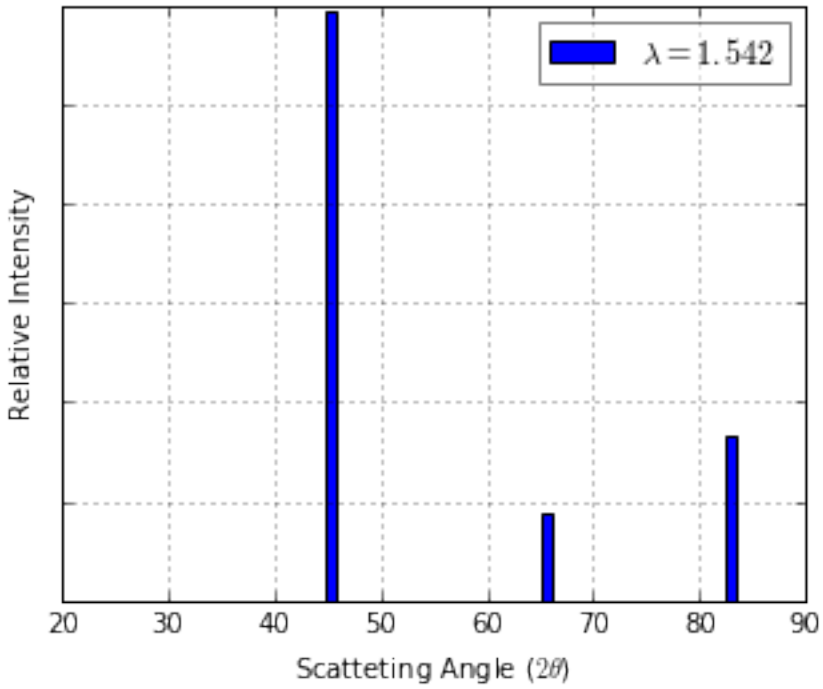
```
cif_path = ipymd.get_data_path('pyr_4C_monoclinic.cif')
data = ipymd.data_input.cif.CIF()
data.setup_data(cif_path)
cif4c_df, cif4c_meta = data.get_atom_data(), data.get_meta_data()
vac_df = ipymd.atom_analysis.nearest_neighbour.vacancy_identification(cif4c_df,0.2,2.3,cif4c_meta,
                                        radius=2.3,remove_dups=True)
vis = ipymd.visualise_sim.Visualise_Sim()
vis.add_atoms(vac_df)
vis.add_box(*cif4c_meta[['a','b','c','origin']])
vis.add_atoms(cif4c_df)
vis.visualise([vis.get_image(xrot=90,yrot=10),
               vis.get_image(xrot=45,yrot=45)],columns=2)
```



XRD Spectrum Prediction

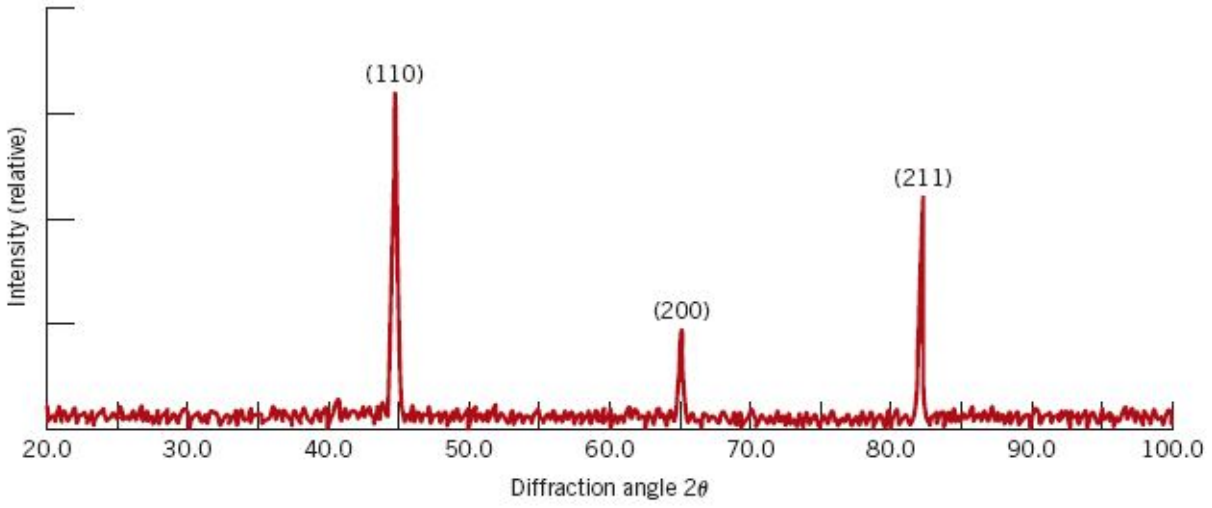
This is an implementation of the virtual x-ray diffraction pattern algorithm, from http://http://dx.doi.org/10.1007/s11837-013-0829-3.

```
data = ipymd.data_input.crystal.Crystal()
data.setup_data(
    [[0,0,0]], ['Fe'],
    229, cellpar=[2.866, 2.866, 2.866, 90, 90, 90],
    repetitions=[5,5,5])

meta = data.get_meta_data()
atoms_df = data.get_atom_data()

wlambda = 1.542 # Angstrom (Cu K-alpha)
thetas, Is = ipymd.atom_analysis.spectral.compute_xrd(atoms_df,meta,wlambda)
plot = ipymd.atom_analysis.spectral.plot_xrd_hist(thetas,Is,wlambda=wlambda,barwidth=1)
plot.axes.set_xlim(20,90)
plot.display_plot(True)
```



The predicted spectrum peaks (for alpha-Fe) shows good correlation with experimentally derived data:

```
from IPython.display import Image
exp_path = ipymd.get_data_path('xrd_fe_bcc_Cu_kalpha.png',
                               module=ipymd.atom_analysis)
Image(exp_path,width=380)
```

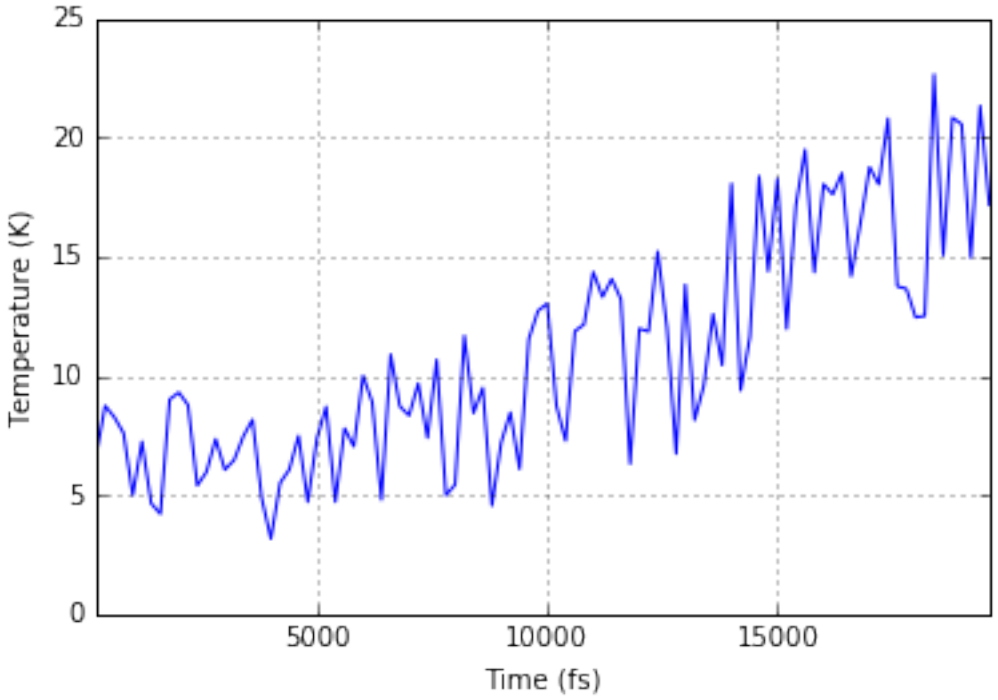


1.2.5 System Analysis

Within the `LAMMPS_Output` class there is also the option to read in a systems data file, with a log of global variables for each simulation timestep.

```
data = ipymd.data_input.lammps.LAMMPS_Output()
sys_path = ipymd.get_data_path('system.dump')
data.setup_data(sys_path=sys_path)
sys_data = data.get_meta_data_all()
sys_data.head()
```

```
ax = sys_data.plot('time','temp',legend=False)
ax.set_xlabel('Time (fs)')
ax.set_ylabel('Temperature (K)');
ax.grid()
```

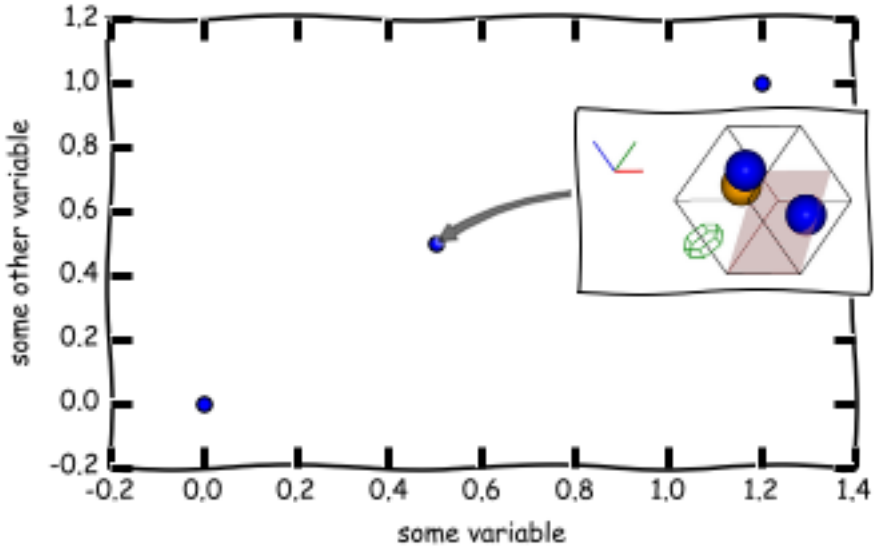


1.2.6 Plotting

Plotting is handled by the `Plotter` class, which is mainly a useful wrapper around `matplotlib`.

```
df = pd.DataFrame(
        [[2,3,4,1,[0, 0, 255],1],
         [1,3,3,1,'orange',1],
         [4,3,1,1,'blue',1]],
        columns=['x','y','z','radius','color','transparency'])
vis = ipymd.visualise_sim.Visualise_Sim()
vis.add_atoms(df)
vis.add_axes(length=0.2, offset=(-0.3,0))
vis.add_box([5,0,0],[0,5,0],[0,0,5])
vis.add_plane([[5,0,0],[0,5,2]],alpha=0.3)
vis.add_hexagon([[1,0,0],[0,0,.5]],[0,0,2],color='green')
img_ex1 = vis.get_image(xrot=45, yrot=45)

with ipymd.plotting.style('xkcd'):
    plot = ipymd.plotting.Plotter(figsize=(5,3))
    plot.axes.scatter([0,0.5,1.2],[0,0.5,1],s=30)
```

```
    plot.axes.set_xlabel('some variable')
    plot.axes.set_ylabel('some other variable')
    plot.add_image_annotation(img_ex1,(230,100),(0.5,0.5),zoom=0.5)
    plot.display_plot(tight_layout=True)
```



Matplotlib also has an animation capability:

```
import numpy as np
x_iter = [np.linspace(0, 2, 1000) for i in range(100)]
def y_iter(x_iter):
    for i,x in enumerate(x_iter):
        yield np.sin(2 * np.pi * (x - 0.01 * i))

with ipymd.plotting.style('ggplot'):
    line_anim = ipymd.plotting.animation_line(x_iter,y_iter(x_iter),
                                              xlim=(0,2),ylim=(-2,2),incl_controls=False)
line_anim
```

Recent IPython Notebook version have brought powerful new interactive features, such as Javascript widgets:

One could imagine using this feature to overlay time-dependant field information on to 2D visualiations of atomic configurations:

```
# visualise atoms
df = pd.DataFrame(
        [[2,3,4,1,'gray',0.6],
         [1,3,3,1,'gray',0.6],
         [4,3,1,1,'gray',0.6]],
        columns=['x','y','z','radius','color','transparency'])
vis = ipymd.visualise_sim.Visualise_Sim()
vis.add_atoms(df,illustrate=True)
img1 = vis.get_image(size=400,quality=5,xrot=90)

plot = ipymd.plotting.Plotter()
plot.add_image(img1,width=2,height=2,origin=(-1,-1))

# setup contour iterators
import numpy as np
from itertools import izip
```

```
from matplotlib.mlab import bivariate_normal

x_iter = [np.linspace(-1, 1, 1000) for i in range(100)]
y_iter = [np.linspace(-1, 1, 1000) for i in range(100)]
def z_iter(x_iter,y_iter):
    for i,(x,y) in enumerate(izip(x_iter,y_iter)):
        X, Y = np.meshgrid(x, y)
        yield bivariate_normal(X, Y, 0.005*(i+1), 0.005*(i+1),0.5,0.5)

# create contour visualisation
with ipymd.plotting.style('ggplot'):
    c_anim = ipymd.plotting.animation_contourf(x_iter,y_iter,z_iter(x_iter,y_iter),
                                       xlim=(-1,1),ylim=(-1,1),
                                       cmap='PuBu_r',alpha=0.5,plot=plot)
c_anim
```

1.3 Package API

1.3.1 ipymd package

Subpackages

ipymd.atom_analysis package

Subpackages

Submodules

ipymd.atom_analysis.basic module Created on Thu Jul 14 14:06:40 2016

@author: cjs14

functions to calculate basic properties of the atoms

ipymd.atom_analysis.basic.**density_bb**(*atoms_df, vectors=[[1, 0, 0], [0, 1, 0], [0, 0, 1]]*)
calculate density of the bounding box (assuming all atoms are inside)

ipymd.atom_analysis.basic.**lattparams_bb**(*vectors=[[1, 0, 0], [0, 1, 0], [0, 0, 1]], rounded=None, cells=(1, 1, 1)*)
calculate unit cell parameters of the bounding box

rounded [int] the number of decimal places to return

cells [(int,int,int)] how many unit cells the vectors represent in each direction

Returns

Return type a, b, c, alpha, beta, gamma (in degrees)

ipymd.atom_analysis.basic.**volume_bb**(*vectors=[[1, 0, 0], [0, 1, 0], [0, 0, 1]], rounded=None, cells=(1, 1, 1)*)
calculate volume of the bounding box

rounded [int] the number of decimal places to return

cells [(int,int,int)] how many unit cells the vectors represent in each direction

Returns volume

Return type float

ipymd.atom_analysis.basic.**volume_points**(*atoms_df*)
calculate volume of the shape encompasing all atom coordinates

ipymd.atom_analysis.nearest_neighbour module Created on Thu Jul 14 14:05:09 2016

@author: cjs14

functions based on nearest neighbour calculations

ipymd.atom_analysis.nearest_neighbour.**bond_lengths**(*atoms_df*, *coord_type*, *lattice_type*, *max_dist=4*, *max_coord=16*, *repeat_meta=None*, *rounded=2*, *min_dist=0.01*, *leafsize=100*)
calculate the unique bond lengths atoms in coords_atoms, w.r.t lattice_atoms

atoms_df [pandas.Dataframe] all atoms

coord_type [string] atoms to calcualte coordination of

lattice_type [string] atoms to act as lattice for coordination

max_dist [float] maximum distance for coordination consideration

max_coord [float] maximum possible coordination number

repeat_meta [pandas.Series] include consideration of repeating boundary idenfined by a,b,c in the meta data

min_dist [float] lattice points within this distance of the atom will be ignored (assumed self-interaction)

leafsize [int] points at which the algorithm switches to brute-force (kdtree specific)

Returns **distances** – list of unique distances

Return type set

ipymd.atom_analysis.nearest_neighbour.**cna_categories**(*atoms_df*, *accuracy=1.0*, *upper_bound=4*, *max_neighbours=24*, *repeat_meta=None*, *leafsize=100*, *ipython_progress=False*)
compute summed atomic environments of each atom in atoms_df

Based on Faken, Daniel and Jónsson, Hannes, 'Systematic analysis of local atomic structure combined with 3D computer graphics', March 1994, DOI: 10.1016/0927-0256(94)90109-0

signatures: - FCC = 12 x 4,2,1 - HCP = 6 x 4,2,1 & 6 x 4,2,2 - BCC = 6 x 6,6,6 & 8 x 4,4,4 - Diamond = 12 x 5,4,3 & 4 x 6,6,3 - Icosahedral = 12 x 5,5,5

Parameters

- **accuracy** (*float*) – 0 to 1 how accurate to fit to signature
- **repeat_meta** (*pandas.Series*) – include consideration of repeating boundary idenfined by a,b,c in the meta data
- **ipython_progress** (*bool*) – print progress to IPython Notebook

Returns **df** – copy of atoms_df with new column named cna

Return type pandas.Dataframe

ipymd.atom_analysis.nearest_neighbour.**cna_plot**(*atoms_df*, *upper_bound=4*, *max_neighbours=24*, *repeat_meta=None*, *leafsize=100*, *barwidth=1*, *ipython_progress=False*)

compute summed atomic environments of each atom in atoms_df

Based on Faken, Daniel and Jónsson, Hannes, 'Systematic analysis of local atomic structure combined with 3D computer graphics', March 1994, DOI: 10.1016/0927-0256(94)90109-0

common signatures: - FCC = 12 x 4,2,1 - HCP = 6 x 4,2,1 & 6 x 4,2,2 - BCC = 6 x 6,6,6 & 8 x 4,4,4 - Diamond = 12 x 5,4,3 & 4 x 6,6,3 - Icosahedral = 12 x 5,5,5

Parameters

- **repeat_meta** (*pandas.Series*) – include consideration of repeating boundary idenfined by a,b,c in the meta data
- **ipython_progress** (*bool*) – print progress to IPython Notebook

Returns **plot** – a matplotlib plot

Return type matplotlib.pyplot

ipymd.atom_analysis.nearest_neighbour.**cna_sum**(*atoms_df*, *upper_bound=4*, *max_neighbours=24*, *repeat_meta=None*, *leafsize=100*, *ipython_progress=False*)

compute summed atomic environments of each atom in atoms_df

Based on Faken, Daniel and Jónsson, Hannes, 'Systematic analysis of local atomic structure combined with 3D computer graphics', March 1994, DOI: 10.1016/0927-0256(94)90109-0

common signatures: - FCC = 12 x 4,2,1 - HCP = 6 x 4,2,1 & 6 x 4,2,2 - BCC = 6 x 6,6,6 & 8 x 4,4,4 - Diamond = 12 x 5,4,3 & 4 x 6,6,3 - Icosahedral = 12 x 5,5,5

Parameters

- **repeat_meta** (*pandas.Series*) – include consideration of repeating boundary idenfined by a,b,c in the meta data
- **ipython_progress** (*bool*) – print progress to IPython Notebook

Returns **counter** – a counter of cna signatures

Return type Counter

ipymd.atom_analysis.nearest_neighbour.**common_neighbour_analysis**(*atoms_df*, *upper_bound=4*, *max_neighbours=24*, *repeat_meta=None*, *leafsize=100*, *ipython_progress=False*)

compute atomic environment of each atom in atoms_df

Based on Faken, Daniel and Jónsson, Hannes, 'Systematic analysis of local atomic structure combined with 3D computer graphics', March 1994, DOI: 10.1016/0927-0256(94)90109-0

ideally: - FCC = 12 x 4,2,1 - HCP = 6 x 4,2,1 & 6 x 4,2,2 - BCC = 6 x 6,6,6 & 8 x 4,4,4 - icosahedral = 12 x 5,5,5

repeat_meta [pandas.Series] include consideration of repeating boundary idenfined by a,b,c in the meta data

ipython_progress [bool] print progress to IPython Notebook

Returns **df** – copy of atoms_df with new column named cna

Return type pandas.Dataframe

ipymd.atom_analysis.nearest_neighbour.**compare_to_lattice**(*atoms_df*, *lattice_atoms_df*, *max_dist=10*, *leafsize=100*)

calculate the minimum distance of each atom in atoms_df from a lattice point in lattice_atoms_df

atoms_df [pandas.Dataframe] atoms to calculate for

lattice_atoms_df [pandas.Dataframe] atoms to act as lattice points

max_dist [float] maximum distance for consideration in computation

leafsize [int] points at which the algorithm switches to brute-force (kdtree specific)

Returns **distances** – list of distances to nearest atom in lattice

Return type list

ipymd.atom_analysis.nearest_neighbour.**coordination**(*coord_atoms_df*, *lattice_atoms_df*, *max_dist=4*, *max_coord=16*, *repeat_meta=None*, *min_dist=0.01*, *leafsize=100*)

calculate the coordination number of each atom in coords_atoms, w.r.t lattice_atoms

coords_atoms_df [pandas.Dataframe] atoms to calcualte coordination of

lattice_atoms_df [pandas.Dataframe] atoms to act as lattice for coordination

max_dist [float] maximum distance for coordination consideration

max_coord [float] maximum possible coordination number

repeat_meta [pandas.Series] include consideration of repeating boundary idenfined by a,b,c in the meta data

min_dist [float] lattice points within this distance of the atom will be ignored (assumed self-interaction)

leafsize [int] points at which the algorithm switches to brute-force (kdtree specific)

Returns **coords** – list of coordination numbers

Return type list

ipymd.atom_analysis.nearest_neighbour.**coordination_bytype**(*atoms_df*, *coord_type*, *lattice_type*, *max_dist=4*, *max_coord=16*, *repeat_meta=None*, *min_dist=0.01*, *leafsize=100*)

returns dataframe with additional column for the coordination number of each atom of coord type, w.r.t lattice_type atoms

effectively an extension of calc_df_coordination

atoms_df [pandas.Dataframe] all atoms

coord_type [string] atoms to calcualte coordination of

lattice_type [string] atoms to act as lattice for coordination

max_dist [float] maximum distance for coordination consideration

max_coord [float] maximum possible coordination number

repeat_meta [pandas.Series] include consideration of repeating boundary idenfined by a,b,c in the meta data

min_dist [float] lattice points within this distance of the atom will be ignored (assumed self-interaction)

leafsize [int] points at which the algorithm switches to brute-force (kdtree specific)

Returns **df** – copy of atoms_df with new column named coord_{coord_type}_{lattice_type}

Return type pandas.Dataframe

ipymd.atom_analysis.nearest_neighbour.**guess_bonds**(*atoms_df*, *covalent_radii=None*, *threshold=0.1*, *max_length=5.0*, *radius=0.1*, *transparency=1.0*, *color=None*)

guess bonds between atoms, based on approximate covalent radii

Parameters

- **atoms_df** (*pandas.Dataframe*) – all atoms, requires colums ['x','y','z','type', 'color']
- **covalent_radii** (*dict or None*) – a dict of covalent radii for each atom type, if None then taken from ipymd.shared.atom_data
- **threshold** (*float*) – include bonds with distance +/- threshold of guessed bond length (Angstrom)
- **max_length** (*float*) – maximum bond length to include (Angstrom)
- **radius** (*float*) – radius of displayed bond cylinder (Angstrom)
- **transparency** (*float*) – transparency of displayed bond cylinder
- **color** (*str or tuple*) – color of displayed bond cylinder, if None then colored by atom color

Returns **bonds_df** – a dataframe with start/end indexes relating to atoms in atoms_df

Return type pandas.Dataframe

ipymd.atom_analysis.nearest_neighbour.**vacancy_identification**(*atoms_df*, *res=0.2*, *nn_dist=2.0*, *repeat_meta=None*, *remove_dups=True*, *color='red'*, *transparency=1.0*, *radius=1*, *type_name='Vac'*, *leafsize=100*, *n_jobs=1*, *ipython_progress=False*)

identify vacancies

atoms_df [pandas.Dataframe] atoms to calculate for

res [float] resolution of vacancy identification, i.e. spacing of reference lattice

nn_dist [float] maximum nearest-neighbour distance considered as a vacancy

repeat_meta [pandas.Series] include consideration of repeating boundary idenfined by a,b,c in the meta data

remove_dups [bool] only keep one vacancy site within the nearest-neighbour distance

leafsize [int] points at which the algorithm switches to brute-force (kdtree specific)

n_jobs [int, optional] Number of jobs to schedule for parallel processing. If -1 is given all processors are used.

ipython_progress [bool] print progress to IPython Notebook

Returns **vac_df** – new atom dataframe of vacancy sites as atoms

Return type pandas.DataFrame

ipymd.atom_analysis.spectral module Created on Tue Jul 12 20:18:04 2016

Derived from: LAMMPS - Large-scale Atomic/Molecular Massively Parallel Simulator http://lammps.sandia.gov, Sandia National Laboratories Steve Plimpton, sjplimp@sandia.gov Copyright (2003) Sandia Corporation. Under the terms of Contract DE-AC04-94AL85000 with Sandia Corporation, the U.S. Government retains certain rights in this software. This software is distributed under the GNU General Public License. https://github.com/lammps/lammps/tree/lammps-icms/src/USER-DIFFRACTION

This package contains the commands neeed to calculate x-ray and electron diffraction intensities based on kinematic diffraction theory. Detailed discription of the computation can be found in the following works:

Coleman, S.P., Spearot, D.E., Capolungo, L. (2013) Virtual diffraction analysis of Ni [010] symmetric tilt grain boundaries, Modelling and Simulation in Materials Science and Engineering, 21 055020. doi:10.1088/0965-0393/21/5/055020

Coleman, S.P., Sichani, M.M., Spearot, D.E. (2014) A computational algorithm to produce virtual x-ray and electron diffraction patterns from atomistic simulations, JOM, 66 (3), 408-416. doi:10.1007/s11837-013-0829-3

Coleman, S.P., Pamidighantam, S. Van Moer, M., Wang, Y., Koesterke, L. Spearot D.E (2014) Performance improvement and workflow development of virtual diffraction calculations, XSEDE14, doi:10.1145/2616498.2616552

@author: chris sewell

ipymd.atom_analysis.spectral.**get_sf_coeffs**()

ipymd.atom_analysis.spectral.**xrd_compute**(*atoms_df, meta_data, wlambda, min2theta=1.0, max2theta=179.0, lp=True, rspace=[1, 1, 1], manual=False, periodic=[True, True, True]*)

Compute predicted x-ray diffraction intensities for a given wavelength

atoms_df [pandas.DataFrame] a dataframe of info for each atom, including columns; x,y,z,type

meta_data [pandas.Series] data of a,b,c crystal vectors (as tuples, e.g. meta_data.a = (0,0,1))

wlambda [float] radiation wavelength (length units) typical values are Cu Ka = 1.54, Mo Ka = 0.71 Angstroms

min2theta [float] minimum 2 theta range to explore (degrees)

max2theta [float] maximum 2 theta range to explore (degrees)

lp [bool] switch to apply Lorentz-polarization factor

use_triclinic [bool] use_triclinic

rspace [list of floats] parameters to adjust the spacing of the reciprocal lattice nodes in the h, k, and l directions respectively

manual [bool] use manual spacing of reciprocal lattice points based on the values of the c parameters (good for comparing diffraction results from multiple simulations, but small c required).

periodic [list of bools] whether periodic boundary in the h, k, and l directions respectively

Returns **df** – theta (in degrees), I, h, k and l for each k-point

Return type pandas.DataFrame

Notes

This is an implementation of the virtual x-ray diffraction pattern algorithm by Coleman *et al*. *[ref1]*

The algorithm proceeds in the following manner:

1.Define a crystal structure by position (x,y,z) and atom/ion type.

2.Define the x-ray wavelength to use

3.Compute the full reciprocal lattice mesh

4.Filter reciprocal lattice points by those in the Eswald's sphere

5.Compute the structure factor at each reciprocal lattice point, for each atom type

6.Compute the x-ray diffraction intensity at each reciprocal lattice point

7.Group and sum intensities by angle

reciprocal points of the lattice are computed such that:

$$\mathbf{K} = m_1 \cdot \mathbf{b}_1 + m_2 \cdot \mathbf{b}_2 + m_3 \cdot \mathbf{b}_3$$

where,

$$\mathbf{b_1} = \frac{\mathbf{a_2} \times \mathbf{a_3}}{\mathbf{a_1} \cdot (\mathbf{a_2} \times \mathbf{a_3})}$$
$$\mathbf{b_2} = \frac{\mathbf{a_3} \times \mathbf{a_1}}{\mathbf{a_2} \cdot (\mathbf{a_3} \times \mathbf{a_1})}$$
$$\mathbf{b_3} = \frac{\mathbf{a_1} \times \mathbf{a_2}}{\mathbf{a_3} \cdot (\mathbf{a_1} \times \mathbf{a_2})}$$

The reciprocal k-point modulii of the x-ray is calculated from Bragg's law:

$$|\mathbf{K}_\lambda| = \frac{1}{d_{\text{hkl}}} = \frac{2\sin(\theta)}{\lambda}$$

This is used to construct an Eswald's sphere, and only reciprocal lattice ponts within are retained, as illustrated:

The atomic scattering factors, fj, accounts for the reduction in diffraction intensity due to Compton scattering, with coefficients based on the electron density around the atomic nuclei.

$$f_j\left(\frac{\sin\theta}{\lambda}\right) = \left[\sum_i^4 a_i \exp\left(-b_i \frac{\sin^2\theta}{\lambda^2}\right)\right] + c = \left[\sum_i^4 a_i \exp\left(-b_i \left(\frac{|\mathbf{K}|}{2}\right)^2\right)\right] + c$$

The relative diffraction intensity from x-rays is computed at each reciprocal lattice point through:

$$I_x(\mathbf{K}) = Lp(\theta)\frac{F(\mathbf{K})F^*(\mathbf{K})}{N}$$

such that:

$$F(\mathbf{K}) = \sum_{j=1}^{N} f_j . e^{(2\pi i\, \mathbf{K}\cdot\mathbf{r}_j)} = \sum_{j=1}^{N} f_j \cdot \left[\cos(2\pi\mathbf{K}\cdot\mathbf{r}_j) + i\sin(2\pi\mathbf{K}\cdot\mathbf{r}_j)\right]$$

and the Lorentz-polarization factor is:

$$Lp(\theta) = \frac{1 + \cos^2(2\theta)}{\cos(\theta)\sin^2(\theta)}$$

References

ipymd.atom_analysis.spectral.**xrd_group_i**(*df*, *tstep=None*, *sym_equiv_hkl='none'*)

group xrd intensities by theta

Parameters

- **df** (*pandas.DataFrame*) – containing columns; ['theta','I'] and optional ['h','k','l']
- **tstep** (*None or float*) – if not None, group thetas within ranges i to i+tstep
- **sym_equiv_hkl** (*str*) – group hkl by symmetry-equivalent refelections; ['none','mmm','m-3m']

Returns **df**

Return type pandas.DataFrame

Notes

if grouping by theta step, then the theta value for each group will be the intensity weighted average

Crystal System | Laue Class | Symmetry-Equivalent Reflections | Multiplicity ————— | ——— | ————————— | ———— Triclinic | -1 | hkl -h-k-l | 2 Monoclinic | 2/m | hkl -hk-l -h-k-l h-kl | 4 Orthorhombic | mmm | hkl h-k-l -hk-l -h-kl -h-k-l -hkl h-kl hk-l | 8 Tetragonal | 4/m | hkl -khl -h-kl k-hl -h-k-l k-h-l hk-l -kh-l | 8

4/mmm | hkl -khl -h-kl k-hl -h-k-l k-h-l hk-l -kh-l |
| khl -hkl -k-hl h-kl -k-h-l h-k-l kh-l -hk-l | 16

Cubic | m-3 | hkl -hkl h-kl hk-l -h-k-l h-k-l -hk-l -h-kl |

| klh -klh k-lh kl-h -k-l-h k-l-h -kl-h -k-lh |
| lhk -lhk l-hk lh-k -l-h-k l-h-k -lh-k -l-hk | 24

m-3m | hkl -hkl h-kl hk-l -h-k-l h-k-l -hk-l -h-kl |
| klh -klh k-lh kl-h -k-l-h k-l-h -kl-h -k-lh |
| lhk -lhk l-hk lh-k -l-h-k l-h-k -lh-k -l-hk |
| khl -khl k-hl kh-l -k-h-l k-h-l -kh-l -k-hl |
| lkh -lkh l-kh lk-h -l-k-h l-k-h -lk-h -l-kh |
| hlk -hlk h-lk hl-k -h-l-k h-l-k -hl-k -h-lk | 48

ipymd.atom_analysis.spectral.**xrd_plot**(*df, icol='I_norm', imin=0.01, barwidth=1.0, hkl_num=0, hkl_trange=[0.0, 180.0], incl_multi=False, label_inline=True, label_trange=[0.0, 180.0], ax=None, **kwargs*)

create plot of xrd pattern

df [pd.DataFrame] containing columns ['theta',icol] and optional ['hkl','multiplicity']

icol [str] column name for intensity data

imin [float] minimum intensity to display

barwidth [float or None] if None then the barwidth will be the bin width

hkl_num [int] number of k-point values to label

hkl_trange [[float,float]] theta range within which to label peaks with k-point values

label_inline [bool] place k-point labels inline or at top of plot

label_trange [[float,float]] if not inline, theta range within which to fit k-point labels

incl_multi [bool] add multiplicity to k-point label

kwargs [optional] additional arguments for bar plot (e.g. label, color, alpha)

Returns **plot** – a plot object

Return type ipymd.plotting.Plotting

Module contents

ipymd.data_input package

Subpackages

ipymd.data_input.spacegroup package

Submodules

ipymd.data_input.spacegroup.cell module

ipymd.data_input.spacegroup.cell.**cell_to_cellpar**(*cell*)

Returns the cell parameters [a, b, c, alpha, beta, gamma] as a numpy array.

ipymd.data_input.spacegroup.cell.**cellpar_to_cell**(*cellpar*, *ab_normal=(0, 0, 1)*, *a_direction=None*)

Return a 3x3 cell matrix from *cellpar* = [a, b, c, alpha, beta, gamma]. The returned cell is orientated such that a and b are normal to *ab_normal* and a is parallel to the projection of *a_direction* in the a-b plane.

Default *a_direction* is (1,0,0), unless this is parallel to *ab_normal*, in which case default *a_direction* is (0,0,1).

The returned cell has the vectors va, vb and vc along the rows. The cell will be oriented such that va and vb are normal to *ab_normal* and va will be along the projection of *a_direction* onto the a-b plane.

Example:

```
>>> cell = cellpar_to_cell([1, 2, 4,  10,  20, 30], (0,1,1), (1,2,3))
>>> np.round(cell, 3)
array([[ 0.816, -0.408,  0.408],
       [ 1.992, -0.13 ,  0.13 ],
       [ 3.859, -0.745,  0.745]])
```

ipymd.data_input.spacegroup.cell.**metric_from_cell**(*cell*)

Calculates the metric matrix from cell, which is given in the Cartesian system.

ipymd.data_input.spacegroup.spacegroup module Definition of the Spacegroup class.

This module only depends on NumPy and the space group database.

class ipymd.data_input.spacegroup.spacegroup.**Spacegroup**(*spacegroup*, *setting=1*, *datafile=None*)

Bases: object

Returns a new Spacegroup instance.

Parameters:

spacegroup [int | string | Spacegroup instance] The space group number in International Tables of Crystallography or its Hermann-Mauguin symbol. E.g. spacegroup=225 and spacegroup='F m -3 m' are equivalent.

setting [1 | 2] Some space groups have more than one setting. *setting* determines Which of these should be used.

datafile [None | string] Path to database file. If *None*, the the default database will be used.

__eq__(*other*)
Chech whether *self* and *other* refer to the same spacegroup number and setting.

__str__()
Return a string representation of the space group data in the same format as found the database.

centrosymmetric
Whether a center of symmetry exists.

equivalent_reflections(*hkl*)
Return all equivalent reflections to the list of Miller indices in hkl.

Example:

```
>>> from ase.lattice.spacegroup import Spacegroup
>>> sg = Spacegroup(225)  # fcc
>>> sg.equivalent_reflections([[0, 0, 2]])
array([[ 0,  0, -2],
       [ 0, -2,  0],
       [-2,  0,  0],
       [ 2,  0,  0],
       [ 0,  2,  0],
       [ 0,  0,  2]])
```

equivalent_sites(*scaled_positions*, *ondublicates='error'*, *symprec=0.001*)
Returns the scaled positions and all their equivalent sites.

Parameters:

scaled_positions: list | array List of non-equivalent sites given in unit cell coordinates.

ondublicates ['keep' | 'replace' | 'warn' | 'error'] Action if *scaled_positions* contain symmetry-equivalent positions:

'keep' ignore additional symmetry-equivalent positions

'replace' replace

'warn' like 'keep', but issue an UserWarning

'error' raises a SpacegroupValueError

symprec: float Minimum "distance" betweed two sites in scaled coordinates before they are counted as the same site.

Returns:

sites: array A NumPy array of equivalent sites.

kinds: list A list of integer indices specifying which input site is equivalent to the corresponding returned site.

Example:

```
>>> from ase.lattice.spacegroup import Spacegroup
>>> sg = Spacegroup(225)  # fcc
>>> sites, kinds = sg.equivalent_sites([[0, 0, 0], [0.5, 0.0, 0.0]])
>>> sites
array([[ 0. ,  0. ,  0. ],
       [ 0. ,  0.5,  0.5],
       [ 0.5,  0. ,  0.5],
       [ 0.5,  0.5,  0. ],
       [ 0.5,  0. ,  0. ],
       [ 0. ,  0.5,  0. ],
       [ 0. ,  0. ,  0.5],
       [ 0.5,  0.5,  0.5]])
>>> kinds
[0, 0, 0, 0, 1, 1, 1, 1]
```

get_op()
Returns all symmetry operations (including inversions and subtranslations), but unlike get_symop(), they are returned as two ndarrays.

get_rotations()
Return all rotations, including inversions for centrosymmetric crystals.

get_symop()
Returns all symmetry operations (including inversions and subtranslations) as a sequence of (rotation, translation) tuples.

lattice
Lattice type: P primitive I body centering, h+k+l=2n F face centering, h,k,l all odd or even A,B,C single face centering, k+l=2n, h+l=2n, h+k=2n R rhombohedral centering, -h+k+l=3n (obverse); h-k+l=3n (reverse)

no
Space group number in International Tables of Crystallography.

nsubtrans
Number of cell-subtranslation vectors.

nsymop
Total number of symmetry operations.

reciprocal_cell
Tree Miller indices that span all kinematically non-forbidden reflections as a matrix with the Miller indices along the rows.

rotations
Symmetry rotation matrices. The invertions are not included for centrosymmetrical crystals.

scaled_primitive_cell
Primitive cell in scaled coordinates as a matrix with the primitive vectors along the rows.

setting
Space group setting. Either one or two.

subtrans
Translations vectors belonging to cell-sub-translations.

symbol
Hermann-Mauguin (or international) symbol for the space group.

symmetry_normalised_reflections(*hkl*)
Returns an array of same size as *hkl*, containing the corresponding symmetry-equivalent reflections of lowest indices.

Example:

```
>>> from ase.lattice.spacegroup import Spacegroup
>>> sg = Spacegroup(225)  # fcc
>>> sg.symmetry_normalised_reflections([[2, 0, 0], [0, 2, 0]])
array([[ 0,  0, -2],
       [ 0,  0, -2]])
```

symmetry_normalised_sites(*scaled_positions*)
Returns an array of same size as *scaled_positions*, containing the corresponding symmetry-equivalent sites within the unit cell of lowest indices.

Example:

```
>>> from ase.lattice.spacegroup import Spacegroup
>>> sg = Spacegroup(225)  # fcc
>>> sg.symmetry_normalised_sites([[0.0, 0.5, 0.5], [1.0, 1.0, 0.0]])
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

tag_sites(*scaled_positions*, *symprec=0.001*)
Returns an integer array of the same length as *scaled_positions*, tagging all equivalent atoms with the same index.

Example:

```
>>> from ase.lattice.spacegroup import Spacegroup
>>> sg = Spacegroup(225)  # fcc
>>> sg.tag_sites([[0.0, 0.0, 0.0],
...               [0.5, 0.5, 0.0],
...               [1.0, 0.0, 0.0],
...               [0.5, 0.0, 0.0]])
array([0, 0, 0, 1])
```

translations
Symmetry translations. The invertions are not included for centrosymmetrical crystals.

unique_reflections(*hkl*)
Returns a subset *hkl* containing only the symmetry-unique reflections.

Example:

```
>>> from ase.lattice.spacegroup import Spacegroup
>>> sg = Spacegroup(225)  # fcc
>>> sg.unique_reflections([[ 2,  0,  0],
...                        [ 0, -2,  0],
...                        [ 2,  2,  0],
...                        [ 0, -2, -2]])
array([[2, 0, 0],
       [2, 2, 0]])
```

unique_sites(*scaled_positions*, *symprec=0.001*, *output_mask=False*)
Returns a subset of *scaled_positions* containing only the symmetry-unique positions. If *output_mask* is True, a boolean array masking the subset is also returned.

Example:

```
>>> from ase.lattice.spacegroup import Spacegroup
>>> sg = Spacegroup(225)  # fcc
>>> sg.unique_sites([[0.0, 0.0, 0.0],
...                  [0.5, 0.5, 0.0],
...                  [1.0, 0.0, 0.0],
...                  [0.5, 0.0, 0.0]])
array([[ 0. ,  0. ,  0. ],
       [ 0.5,  0. ,  0. ]])
```

Module contents Created on Sat Jul 16 05:26:00 2016

@author: cjs14

Submodules

ipymd.data_input.base module Created on Sun May 1 22:49:20 2016

@author: cjs14

class ipymd.data_input.base.**DataInput**

Bases: object

Data is divided into two levels; atomic and meta

- atom is a series of tables, one for each timestep, containing variables (columns) for each atom (rows)
- meta is a table containing variables (columns) for each configuration (rows)

count_configs()

return int of total number of atomic configurations

get_atom_data(*config=1*)

return pandas.DataFrame of atomic data

get_meta_data(*config=1*)

return pandas.Series of meta data for the atomic configuration

get_meta_data_all(*incl_bb=False*, ***kwargs*)

return pandas.DataFrame of meta data for the atomic configuration

incl_bb [bool] whether to include bounding box coordinates in DataFrame

kwargs [dict] kew word arguments relevant to specific input data

setup_data()

a method to setup the data and variables

ipymd.data_input.cif module Created on Wed May 18 22:24:10 2016

@author: cjs14

adapted from from http://physics.bu.edu/~erikl/research/tools/crystals/read_cif.py

class ipymd.data_input.cif.**CIF**

Bases: *ipymd.data_input.base.DataInput*

Data is divided into two levels; atomic and meta

- atom is a series of tables, one for each timestep, containing variables (columns) for each atom (rows)

•meta is a table containing variables (columns) for each configuration (rows)

setup_data (*file_path*, *override_abc=[]*, *ignore_overlaps=False*)
Build a crystal from a Crystallographic Information File (.cif)

Parameters **file_path** (*str*) – path to file override_abc : [] or [a,b,c]

if not empty, will override a, b, c length parameters given in file

Notes

here is a typical example of a CIF file:

_cell_length_a 4.916 _cell_length_b 4.916 _cell_length_c 5.4054 _cell_angle_alpha 90 _cell_angle_beta 90 _cell_angle_gamma 120 _cell_volume 113.131 _exptl_crystal_density_diffrn 2.646 _symmetry_space_group_name_H-M 'P 32 2 1' loop_ _space_group_symop_operation_xyz

'x,y,z' 'y,x,2/3-z' '-y,x-y,2/3+z' '-x,-x+y,1/3-z' '-x+y,-x,1/3+z' 'x-y,-y,-z'

loop_ _atom_site_label _atom_site_fract_x _atom_site_fract_y _atom_site_fract_z Si 0.46970 0.00000 0.00000 O 0.41350 0.26690 0.11910

ipymd.data_input.crystal module Created on Mon May 16 01:23:11 2016

@author: cjs14

Adapted from chemlab which, in turn, was adapted from ASE https://wiki.fysik.dtu.dk/ase/ Copyright (C) 2010, Jesper Friis

class ipymd.data_input.crystal.**Crystal**
Bases: *ipymd.data_input.base.DataInput*

Data is divided into two levels; atomic and meta

•atom is a series of tables, one for each timestep, containing variables (columns) for each atom (rows)

•meta is a table containing variables (columns) for each configuration (rows)

setup_data (*positions, atom_type, group, cellpar=[1.0, 1.0, 1.0, 90, 90, 90], repetitions=[1, 1, 1], mass_map={}, charge_map={}*)
Build a crystal from atomic positions, space group and cell parameters (in Angstroms)

Parameters

- **positions** (*list of coordinates*) – A list of the fractional atomic positions
- **atom_type** (*list of atom type*) – The atom types corresponding to the positions, the atoms will be translated in all the equivalent positions.
- **group** (*int | str*) – Space group given as its number in International Tables NB: to see mappings from Hermann–Mauguin notation, etc, use the get_spacegroup_df function in this module
- **repetitions** – Repetition of the unit cell in each direction
- **cellpar** – Unit cell parameters (in Angstroms and degrees)
- **mass_map** (*dict of atom masses*) – mapping of atom masses to atom types
- **charge_map** (*dict of atom charges*) – mapping of atom charges to atom types
- **function was taken and adapted from the spacegroup module** (*This*) –

:param found in ASE.: :param The module *spacegroup* module was originally developed by Jesper: :param Frills.:

Example

from ipymd.data_input import crystal c = crystal.Crystal([[0.0, 0.0, 0.0], [0.5, 0.5, 0.5]],

['Na', 'Cl'], 225, cellpar = [5.4, 5.4, 5.4, 90, 90, 90], repetitions = [5, 5, 5])

c.get_atom_data() c.get_simulation_box()

ipymd.data_input.crystal.**get_spacegroup_df**()
dataframe of spacegroup mappings

ipymd.data_input.lammps module Created on Mon May 16 01:15:56 2016

@author: cjs14

class ipymd.data_input.lammps.**LAMMPS_Input**
Bases: *ipymd.data_input.base.DataInput*

Data is divided into two levels; atomic and meta

- atom is a series of tables, one for each timestep, containing variables (columns) for each atom (rows)
- meta is a table containing variables (columns) for each configuration (rows)

setup_data(*atom_path=''*, *atom_style='atomic'*)
get data from file

Parameters **atom_style** (*'atomic', 'charge'*) – defines how atomic data is listed: atomic; atom-ID atom-type x y z charge; atom-ID atom-type q x y z

class ipymd.data_input.lammps.**LAMMPS_Output**
Bases: *ipymd.data_input.base.DataInput*

Data is divided into two levels; atomic and meta

- atom is a series of tables, one for each timestep, containing variables (columns) for each atom (rows)
- meta is a table containing variables (columns) for each configuration (rows)

setup_data(*atom_path=''*, *sys_path=''*, *sys_sep=' '*, *atom_map={}*, *incl_atom_step=False*, *incl_sys_data=True*)
Data divided into two levels; meta and atom

atom_map [dict] mapping of atom level variable names e.g. {'C_var[1]':'x'}

sys_sep [str] the separator between variables in the system data file

incl_atom_time [bool] include time according to atom file in column 'atom_step' of meta

incl_sys_data [bool] include system data in the single step meta data

Notes

Meta level data created with *fix print*, e.g.;

fix sys_info all print 100 "${t} ${natoms} ${temp}" & title "time natoms temp" file system.dump screen no

Atom level data created with *dump*, e.g.;

dump atom_info all custom 100 atom.dump id type x y z mass q OR (file per configuration) dump atom_info all custom 100 atom_*.dump id type xs ys zs mass q

ipymd.data_input.lammps.**atoi**(*text*)

ipymd.data_input.lammps.**natural_keys**(*text*)
alist.sort(key=natural_keys) sorts in human order, e.g. ['1','2','100'] instead of ['1','100','2'] http://nedbatchelder.com/blog/200712/human_sorting.html

Module contents

ipymd.plotting package

Subpackages

ipymd.plotting.JSAnimation package

Submodules

ipymd.plotting.JSAnimation.IPython_display module
ipymd.plotting.JSAnimation.IPython_display.**anim_to_html**(*anim*, *fps=None*, *embed_frames=True*, *default_mode='loop'*)
Generate HTML representation of the animation
ipymd.plotting.JSAnimation.IPython_display.**display_animation**(*anim*, ***kwargs*)
Display the animation with an IPython HTML object

ipymd.plotting.JSAnimation.examples module

ipymd.plotting.JSAnimation.html_writer module

Module contents

Submodules

ipymd.plotting.plotter module Created on Fri Jul 1 16:45:06 2016

@author: cjs14

class ipymd.plotting.plotter.**Plotter**(*nrows=1*, *ncols=1*, *figsize=(5, 4)*)
Bases: object

a class to deal with data plotting

figure
matplotlib.figure

the figure

axes
list or single matplotlib.axes

if more than one then returns a list (ordered in reading direction), else returns one instance

add_image(*image*, *axes=0*, *interpolation='bicubic'*, *hide_axes=True*, *width=1.0*, *height=1.0*, *origin=(0.0, 0.0)*, ***kwargs*)
add image to axes

add_image_annotation(*img*, *xy=(0, 0)*, *arrow_xy=None*, *axes=0*, *zoom=1*, *xytype='axes points'*, *arrow_xytype='data'*, *arrowprops={'connectionstyle': 'arc3, rad=0.2', 'alpha': 0.4, 'arrowstyle': 'simple', 'facecolor': 'black'}*)
add an image to the plot

coordtype:

argument	coordinate system
'figure points'	points from the lower left corner of the figure
'figure pixels'	pixels from the lower left corner of the figure
'figure fraction'	0,0 is lower left of figure and 1,1 is upper right
'axes points'	points from lower left corner of axes
'axes pixels'	pixels from lower left corner of axes
'axes fraction'	0,0 is lower left of axes and 1,1 is upper right
'data'	use the axes data coordinate system

for arrowprops see http://matplotlib.org/users/annotations_guide.html#annotating-with-arrow

axes

display_plot(*tight_layout=False*)
display plot in IPython

if tight_layout is True it may crop anything outside axes

figure

get_image(*size=300*, *dpi=300*, *tight_layout=False*)
return as PIL image

if tight_layout is True it may crop anything outside axes

resize_axes(*width=0.8*, *height=0.8*, *left=0.1*, *bottom=0.1*, *axes=0*)
resiaze axes, for instance to fit object outside of it

ipymd.plotting.plotter.**animation_contourf**(*x_iter*, *y_iter*, *z_iter*, *interval=20*, *xlim=(0, 1)*, *ylim=(0, 1)*, *zlim=(0, 1.0)*, *cmap='viridis'*, *cbar=True*, *incl_controls=True*, *plot=None*, *ax=0*, ***plot_kwargs*)
create an animation of multiple x,y data sets

x_iter [iterable] any iterable of x data sets, e.g. [[1,2,3],[4,5,6]]

y_iter [iterable] an iterable of y data sets, e.g. [[1,2,3],[4,5,6]]

y_iter [iterable] an iterable of z(x,y) data sets, each set must be of shape (len(x), len(y))

interval [int] draws a new frame every *interval* milliseconds

xlim [tuple] the x_limits for the axes (ignored if using existing plotter)

ylim [tuple] the y_limits for the axes (ignored if using existing plotter)

zlim [tuple] the z_limits for the colormap

cmap [str or matplotlib.cm] the colormap to use (see http://matplotlib.org/examples/color/colormaps_reference.html)

incl_controls [bool] include Javascript play controls

plot [ipymd.plotting.Plotter] an existing plotter object

ax [int] the id number of the axes on which to plot (if using existing plotter)

plot_kwargs [various] key word arguments to pass to plot method, e.g. marker='o', color='b', ...

Returns **html** – a html object

Return type IPython.core.display.HTML

Notes

x_iter and y_iter can be yield functions such as:

```
def y_iter(x_iter):
    for xs in x_iter:
        yield [i**2 for i in xs]
```

This means that the values do not have to be necessarily pre-computed.

ipymd.plotting.plotter.**animation_line**(*x_iter*, *y_iter*, *interval=20*, *xlim=(0, 1)*, *ylim=(0, 1)*, *incl_controls=True*, *plot=None*, *ax=0*, ***plot_kwargs*)

create an animation of multiple x,y data sets

x_iter [iterable] any iterable of x data sets, e.g. [[1,2,3],[4,5,6]]

y_iter [iterable] an iterable of y data sets, e.g. [[1,2,3],[4,5,6]]

interval [int] draws a new frame every *interval* milliseconds

xlim [tuple] the x_limits for the axes (ignored if using existing plotter)

ylim [tuple] the y_limits for the axes (ignored if using existing plotter)

incl_controls [bool] include Javascript play controls

plot [ipymd.plotting.Plotter] an existing plotter object

ax [int] the id number of the axes on which to plot (if using existing plotter)

plot_kwargs [various] key word arguments to pass to plot method, e.g. marker='o', color='b', ...

Returns **html** – a html object

Return type IPython.core.display.HTML

Notes

x_iter and y_iter can be yield functions such as:

```
def y_iter(x_iter):
    for xs in x_iter:
        yield [i**2 for i in xs]
```

This means that the values do not have to be necessarily pre-computed.

ipymd.plotting.plotter.**animation_scatter**(*x_iter*, *y_iter*, *interval=20*, *xlim=(0, 1)*, *ylim=(0, 1)*, *incl_controls=True*, *plot=None*, *ax=0*, ***plot_kwargs*)

create an animation of multiple x,y data sets

x_iter [iterable] any iterable of x data sets, e.g. [[1,2,3],[4,5,6]]

y_iter [iterable] an iterable of y data sets, e.g. [[1,2,3],[4,5,6]]

interval [int] draws a new frame every *interval* milliseconds

xlim [tuple] the x_limits for the axes (ignored if using existing plotter)

ylim [tuple] the y_limits for the axes (ignored if using existing plotter)

incl_controls [bool] include Javascript play controls

plot [ipymd.plotting.Plotter] an existing plotter object

ax [int] the id number of the axes on which to plot (if using existing plotter)

plot_kwargs [various] key word arguments to pass to plot method, e.g. marker='o', color='b', ...

Returns **html** – a html object

Return type IPython.core.display.HTML

Notes

x_iter and y_iter can be yield functions such as:

```
def y_iter(x_iter):
    for xs in x_iter:
        yield [i**2 for i in xs]
```

This means that the values do not have to be necessarily pre-computed.

ipymd.plotting.plotter.**style**(*style*)

A context manager to apply matplotlib style settings from a style specification.

Popular styles include; default, ggplot, xkcd, and are used in the the following manner:

```
with ipymd.plotting.style('default'):
    plot = ipymd.plotting.Plotter()
    plot.display_plot()
```

Parameters **style** (*str, dict, or list*) – A style specification. Valid options are:

str	The name of a style or a path/URL to a style file. For a list of available style names, see *style.available*.
dict	Dictionary with valid key/value pairs for *matplotlib.rcParams*.
list	A list of style specifiers (str or dict) applied from first to last in the list.

Module contents

ipymd.shared package

Subpackages

ipymd.shared.atomdata package

Module contents Created on Sun May 1 22:46:22 2016

@author: cjs14

ipymd.shared.fonts package

Module contents Created on Sun May 1 22:46:22 2016

@author: cjs14

Submodules

ipymd.shared.colors module Created on Wed Jun 29 01:56:51 2016

@author: cjs14

ipymd.shared.colors.**any_to_rgb**(*color*)
If color is an rgb tuple return it, if it is a string, parse it and return the respective rgb tuple.

ipymd.shared.colors.**available_colors**()

ipymd.shared.colors.**get**(*name*)
Given a string *color*, return the color as a tuple (r, g, b, a) where each value is between 0 and 255.

As for the color name follow the *HTML color names <http://www.w3schools.com/tags/ref_colornames.asp>* in lowescore style eg. *forest_green*.

ipymd.shared.colors.**hsl_to_rgb**(*arr*)
Converts HSL color array to RGB array

H = [0..360] S = [0..1] l = [0..1]

http://en.wikipedia.org/wiki/HSL_and_HSV#From_HSL

Returns R,G,B in [0..255]

ipymd.shared.colors.**html_to_rgb**(*colorstring*)
convert #RRGGBB to an (R, G, B) tuple

ipymd.shared.colors.**mix**(*a*, *b*, *ratio=0.5*)

ipymd.shared.colors.**parse_color**(*color*)
Return the RGB 0-255 representation of the current string passed.

It first tries to match the string with DVI color names.

ipymd.shared.colors.**rgb_to_hsl**(*a*)

ipymd.shared.colors.**rgb_to_hsl_hsv**(*a*, *isHSV=True*)
Converts RGB image data to HSV or HSL. :param a: 3D array. Retval of numpy.asarray(Image.open(...), int) :param isHSV: True = HSV, False = HSL :return: H,S,L or H,S,V array

ipymd.shared.colors.**rgb_to_hsv**(*a*)

ipymd.shared.transformations module Homogeneous Transformation Matrices and Quaternions.

A library for calculating 4x4 matrices for translating, rotating, reflecting, scaling, shearing, projecting, orthogonalizing, and superimposing arrays of 3D homogeneous coordinates as well as for converting between rotation matrices, Euler angles, and quaternions. Also includes an Arcball control object and functions to decompose transformation matrices.

Authors Christoph Gohlke, Laboratory for Fluorescence Dynamics, University of California, Irvine

Version 2012.10.14

Requirements

- CPython 2.7 or 3.2
- Numpy 1.6
- transformations.c 2012.01.01 (optional implementation of some functions in C)

Notes

The API is not stable yet and is expected to change between revisions.

This Python code is not optimized for speed. Refer to the transformations.c module for a faster implementation of some functions.

Documentation in HTML format can be generated with epydoc.

Matrices (M) can be inverted using numpy.linalg.inv(M), be concatenated using numpy.dot(M0, M1), or transform homogeneous coordinate arrays (v) using numpy.dot(M, v) for shape (4, *) column vectors, respectively numpy.dot(v, M.T) for shape (*, 4) row vectors ("array of points").

This module follows the "column vectors on the right" and "row major storage" (C contiguous) conventions. The translation components are in the right column of the transformation matrix, i.e. M[:3, 3]. The transpose of the transformation matrices may have to be used to interface with other graphics systems, e.g. with OpenGL's glMultMatrixd(). See also [16].

Calculations are carried out with numpy.float64 precision.

Vector, point, quaternion, and matrix function arguments are expected to be "array like", i.e. tuple, list, or numpy arrays.

Return types are numpy arrays unless specified otherwise.

Angles are in radians unless specified otherwise.

Quaternions w+ix+jy+kz are represented as [w, x, y, z].

A triple of Euler angles can be applied/interpreted in 24 ways, which can be specified using a 4 character string or encoded 4-tuple:

Axes 4-string: e.g. 'sxyz' or 'ryxy'

- first character : rotations are applied to 's'tatic or 'r'otating frame
- remaining characters : successive rotation axis 'x', 'y', or 'z'

Axes 4-tuple: e.g. (0, 0, 0, 0) or (1, 1, 1, 1)

- inner axis: code of axis ('x':0, 'y':1, 'z':2) of rightmost matrix.
- parity : even (0) if inner axis 'x' is followed by 'y', 'y' is followed by 'z', or 'z' is followed by 'x'. Otherwise odd (1).

- repetition : first and last axis are same (1) or different (0).
- frame : rotations are applied to static (0) or rotating (1) frame.

References

1. Matrices and transformations. Ronald Goldman. In "Graphics Gems I", pp 472-475. Morgan Kaufmann, 1990.
2. More matrices and transformations: shear and pseudo-perspective. Ronald Goldman. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.
3. Decomposing a matrix into simple transformations. Spencer Thomas. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.
4. Recovering the data from the transformation matrix. Ronald Goldman. In "Graphics Gems II", pp 324-331. Morgan Kaufmann, 1991.
5. Euler angle conversion. Ken Shoemake. In "Graphics Gems IV", pp 222-229. Morgan Kaufmann, 1994.
6. Arcball rotation control. Ken Shoemake. In "Graphics Gems IV", pp 175-192. Morgan Kaufmann, 1994.
7. Representing attitude: Euler angles, unit quaternions, and rotation vectors. James Diebel. 2006.
8. A discussion of the solution for the best rotation to relate two sets of vectors. W Kabsch. Acta Cryst. 1978. A34, 827-828.
9. Closed-form solution of absolute orientation using unit quaternions. BKP Horn. J Opt Soc Am A. 1987. 4(4):629-642.
10. Quaternions. Ken Shoemake. http://www.sfu.ca/~jwa3/cmpt461/files/quatut.pdf
11. From quaternion to matrix and back. JMP van Waveren. 2005. http://www.intel.com/cd/ids/developer/asmo-na/eng/293748.htm
12. Uniform random rotations. Ken Shoemake. In "Graphics Gems III", pp 124-132. Morgan Kaufmann, 1992.
13. Quaternion in molecular modeling. CFF Karney. J Mol Graph Mod, 25(5):595-604
14. New method for extracting the quaternion from a rotation matrix. Itzhack Y Bar-Itzhack, J Guid Contr Dynam. 2000. 23(6): 1085-1087.
15. Multiple View Geometry in Computer Vision. Hartley and Zissermann. Cambridge University Press; 2nd Ed. 2004. Chapter 4, Algorithm 4.7, p 130.
16. Column Vectors vs. Row Vectors. http://steve.hollasch.net/cgindex/math/matrix/column-vec.html

Examples

```
>>> alpha, beta, gamma = 0.123, -1.234, 2.345
>>> origin, xaxis, yaxis, zaxis = [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]
>>> I = identity_matrix()
>>> Rx = rotation_matrix(alpha, xaxis)
>>> Ry = rotation_matrix(beta, yaxis)
>>> Rz = rotation_matrix(gamma, zaxis)
>>> R = concatenate_matrices(Rx, Ry, Rz)
>>> euler = euler_from_matrix(R, 'rxyz')
>>> numpy.allclose([alpha, beta, gamma], euler)
True
>>> Re = euler_matrix(alpha, beta, gamma, 'rxyz')
>>> is_same_transform(R, Re)
True
```

```
>>> al, be, ga = euler_from_matrix(Re, 'rxyz')
>>> is_same_transform(Re, euler_matrix(al, be, ga, 'rxyz'))
True
>>> qx = quaternion_about_axis(alpha, xaxis)
>>> qy = quaternion_about_axis(beta, yaxis)
>>> qz = quaternion_about_axis(gamma, zaxis)
>>> q = quaternion_multiply(qx, qy)
>>> q = quaternion_multiply(q, qz)
>>> Rq = quaternion_matrix(q)
>>> is_same_transform(R, Rq)
True
>>> S = scale_matrix(1.23, origin)
>>> T = translation_matrix([1, 2, 3])
>>> Z = shear_matrix(beta, xaxis, origin, zaxis)
>>> R = random_rotation_matrix(numpy.random.rand(3))
>>> M = concatenate_matrices(T, R, Z, S)
>>> scale, shear, angles, trans, persp = decompose_matrix(M)
>>> numpy.allclose(scale, 1.23)
True
>>> numpy.allclose(trans, [1, 2, 3])
True
>>> numpy.allclose(shear, [0, math.tan(beta), 0])
True
>>> is_same_transform(R, euler_matrix(axes='sxyz', *angles))
True
>>> M1 = compose_matrix(scale, shear, angles, trans, persp)
>>> is_same_transform(M, M1)
True
>>> v0, v1 = random_vector(3), random_vector(3)
>>> M = rotation_matrix(angle_between_vectors(v0, v1), vector_product(v0, v1))
>>> v2 = numpy.dot(v0, M[:3,:3].T)
>>> numpy.allclose(unit_vector(v1), unit_vector(v2))
True
```

class ipymd.shared.transformations.**Arcball**(*initial=None*)

Bases: object

Initialize virtual trackball control.

initial : quaternion or rotation matrix

down(*point*)

Set initial cursor window coordinates and pick constrain-axis.

drag(*point*)

Update current cursor window coordinates.

getconstrain()

Return state of constrain to axis mode.

matrix()

Return homogeneous rotation matrix.

next(*acceleration=0.0*)

Continue rotation in direction of last drag.

place(*center*, *radius*)

Place Arcball, e.g. when window size changes.

center [sequence[2]] Window coordinates of trackball center.

radius [float] Radius of trackball in window coordinates.

setaxes(**axes*)
Set axes to constrain rotations.

setconstrain(*constrain*)
Set state of constrain to axis mode.

ipymd.shared.transformations.**affine_matrix_from_points**(*v0*, *v1*, *shear=True*, *scale=True*, *usesvd=True*)

Return affine transform matrix to register two point sets.

v0 and v1 are shape (ndims, *) arrays of at least ndims non-homogeneous coordinates, where ndims is the dimensionality of the coordinate space.

If shear is False, a similarity transformation matrix is returned. If also scale is False, a rigid/Eucledian transformation matrix is returned.

By default the algorithm by Hartley and Zissermann [15] is used. If usesvd is True, similarity and Eucledian transformation matrices are calculated by minimizing the weighted sum of squared deviations (RMSD) according to the algorithm by Kabsch [8]. Otherwise, and if ndims is 3, the quaternion based algorithm by Horn [9] is used, which is slower when using this Python implementation.

The returned matrix performs rotation, translation and uniform scaling (if specified).

```
>>> v0 = [[0, 1031, 1031, 0], [0, 0, 1600, 1600]]
>>> v1 = [[675, 826, 826, 677], [55, 52, 281, 277]]
>>> affine_matrix_from_points(v0, v1)
array([[   0.14549,    0.00062,  675.50008],
       [   0.00048,    0.14094,   53.24971],
       [   0.     ,    0.     ,    1.     ]])
>>> T = translation_matrix(numpy.random.random(3)-0.5)
>>> R = random_rotation_matrix(numpy.random.random(3))
>>> S = scale_matrix(random.random())
>>> M = concatenate_matrices(T, R, S)
>>> v0 = (numpy.random.rand(4, 100) - 0.5) * 20
>>> v0[3] = 1
>>> v1 = numpy.dot(M, v0)
>>> v0[:3] += numpy.random.normal(0, 1e-8, 300).reshape(3, -1)
>>> M = affine_matrix_from_points(v0[:3], v1[:3])
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
```

More examples in superimposition_matrix()

ipymd.shared.transformations.**angle_between_vectors**(*v0*, *v1*, *directed=True*, *axis=0*)

Return angle between vectors.

If directed is False, the input vectors are interpreted as undirected axes, i.e. the maximum angle is pi/2.

```
>>> a = angle_between_vectors([1, -2, 3], [-1, 2, -3])
>>> numpy.allclose(a, math.pi)
True
>>> a = angle_between_vectors([1, -2, 3], [-1, 2, -3], directed=False)
>>> numpy.allclose(a, 0)
True
>>> v0 = [[2, 0, 0, 2], [0, 2, 0, 2], [0, 0, 2, 2]]
>>> v1 = [[3], [0], [0]]
>>> a = angle_between_vectors(v0, v1)
>>> numpy.allclose(a, [0, 1.5708, 1.5708, 0.95532])
True
>>> v0 = [[2, 0, 0], [2, 0, 0], [0, 2, 0], [2, 0, 0]]
```

```
    >>> v1 = [[0, 3, 0], [0, 0, 3], [0, 0, 3], [3, 3, 3]]
    >>> a = angle_between_vectors(v0, v1, axis=1)
    >>> numpy.allclose(a, [1.5708, 1.5708, 1.5708, 0.95532])
    True
```

ipymd.shared.transformations.**arcball_constrain_to_axis**(*point*, *axis*)
Return sphere point perpendicular to axis.

ipymd.shared.transformations.**arcball_map_to_sphere**(*point*, *center*, *radius*)
Return unit sphere coordinates from window coordinates.

ipymd.shared.transformations.**arcball_nearest_axis**(*point*, *axes*)
Return axis, which arc is nearest to point.

ipymd.shared.transformations.**clip_matrix**(*left*, *right*, *bottom*, *top*, *near*, *far*, *perspective=False*)
Return matrix to obtain normalized device coordinates from frustrum.

The frustrum bounds are axis-aligned along x (left, right), y (bottom, top) and z (near, far).

Normalized device coordinates are in range [-1, 1] if coordinates are inside the frustrum.

If perspective is True the frustrum is a truncated pyramid with the perspective point at origin and direction along z axis, otherwise an orthographic canonical view volume (a box).

Homogeneous coordinates transformed by the perspective clip matrix need to be dehomogenized (divided by w coordinate).

```
    >>> frustrum = numpy.random.rand(6)
    >>> frustrum[1] += frustrum[0]
    >>> frustrum[3] += frustrum[2]
    >>> frustrum[5] += frustrum[4]
    >>> M = clip_matrix(perspective=False, *frustrum)
    >>> numpy.dot(M, [frustrum[0], frustrum[2], frustrum[4], 1])
    array([-1., -1., -1.,  1.])
    >>> numpy.dot(M, [frustrum[1], frustrum[3], frustrum[5], 1])
    array([ 1.,  1.,  1.,  1.])
    >>> M = clip_matrix(perspective=True, *frustrum)
    >>> v = numpy.dot(M, [frustrum[0], frustrum[2], frustrum[4], 1])
    >>> v / v[3]
    array([-1., -1., -1.,  1.])
    >>> v = numpy.dot(M, [frustrum[1], frustrum[3], frustrum[4], 1])
    >>> v / v[3]
    array([ 1.,  1., -1.,  1.])
```

ipymd.shared.transformations.**compose_matrix**(*scale=None*, *shear=None*, *angles=None*, *translate=None*, *perspective=None*)
Return transformation matrix from sequence of transformations.

This is the inverse of the decompose_matrix function.

Sequence of transformations: scale : vector of 3 scaling factors shear : list of shear factors for x-y, x-z, y-z axes angles : list of Euler angles about static x, y, z axes translate : translation vector along x, y, z axes perspective : perspective partition of matrix

```
    >>> scale = numpy.random.random(3) - 0.5
    >>> shear = numpy.random.random(3) - 0.5
    >>> angles = (numpy.random.random(3) - 0.5) * (2*math.pi)
    >>> trans = numpy.random.random(3) - 0.5
    >>> persp = numpy.random.random(4) - 0.5
    >>> M0 = compose_matrix(scale, shear, angles, trans, persp)
    >>> result = decompose_matrix(M0)
```

```
    >>> M1 = compose_matrix(*result)
    >>> is_same_transform(M0, M1)
    True
```

ipymd.shared.transformations.**concatenate_matrices**(**matrices*)

Return concatenation of series of transformation matrices.

```
    >>> M = numpy.random.rand(16).reshape((4, 4)) - 0.5
    >>> numpy.allclose(M, concatenate_matrices(M))
    True
    >>> numpy.allclose(numpy.dot(M, M.T), concatenate_matrices(M, M.T))
    True
```

ipymd.shared.transformations.**decompose_matrix**(*matrix*)

Return sequence of transformations from transformation matrix.

matrix [array_like] Non-degenerative homogeneous transformation matrix

Return tuple of: scale : vector of 3 scaling factors shear : list of shear factors for x-y, x-z, y-z axes angles : list of Euler angles about static x, y, z axes translate : translation vector along x, y, z axes perspective : perspective partition of matrix

Raise ValueError if matrix is of wrong type or degenerative.

```
    >>> T0 = translation_matrix([1, 2, 3])
    >>> scale, shear, angles, trans, persp = decompose_matrix(T0)
    >>> T1 = translation_matrix(trans)
    >>> numpy.allclose(T0, T1)
    True
    >>> S = scale_matrix(0.123)
    >>> scale, shear, angles, trans, persp = decompose_matrix(S)
    >>> scale[0]
    0.123
    >>> R0 = euler_matrix(1, 2, 3)
    >>> scale, shear, angles, trans, persp = decompose_matrix(R0)
    >>> R1 = euler_matrix(*angles)
    >>> numpy.allclose(R0, R1)
    True
```

ipymd.shared.transformations.**distance**(*x1*, *x2*)

Distance between two points in space

ipymd.shared.transformations.**euler_from_matrix**(*matrix*, *axes='sxyz'*)

Return Euler angles from rotation matrix for specified axis sequence.

axes : One of 24 axis sequences as string or encoded tuple

Note that many Euler angle triplets can describe one matrix.

```
    >>> R0 = euler_matrix(1, 2, 3, 'syxz')
    >>> al, be, ga = euler_from_matrix(R0, 'syxz')
    >>> R1 = euler_matrix(al, be, ga, 'syxz')
    >>> numpy.allclose(R0, R1)
    True
    >>> angles = (4*math.pi) * (numpy.random.random(3) - 0.5)
    >>> for axes in _AXES2TUPLE.keys():
    ...    R0 = euler_matrix(axes=axes, *angles)
    ...    R1 = euler_matrix(axes=axes, *euler_from_matrix(R0, axes))
    ...    if not numpy.allclose(R0, R1): print(axes, "failed")
```

ipymd.shared.transformations.**euler_from_quaternion**(*quaternion*, *axes='sxyz'*)
Return Euler angles from quaternion for specified axis sequence.

```
>>> angles = euler_from_quaternion([0.99810947, 0.06146124, 0, 0])
>>> numpy.allclose(angles, [0.123, 0, 0])
True
```

ipymd.shared.transformations.**euler_matrix**(*ai*, *aj*, *ak*, *axes='sxyz'*)
Return homogeneous rotation matrix from Euler angles and axis sequence.

ai, aj, ak : Euler's roll, pitch and yaw angles axes : One of 24 axis sequences as string or encoded tuple

```
>>> R = euler_matrix(1, 2, 3, 'syxz')
>>> numpy.allclose(numpy.sum(R[0]), -1.34786452)
True
>>> R = euler_matrix(1, 2, 3, (0, 1, 0, 1))
>>> numpy.allclose(numpy.sum(R[0]), -0.383436184)
True
>>> ai, aj, ak = (4*math.pi) * (numpy.random.random(3) - 0.5)
>>> for axes in _AXES2TUPLE.keys():
...    R = euler_matrix(ai, aj, ak, axes)
>>> for axes in _TUPLE2AXES.keys():
...    R = euler_matrix(ai, aj, ak, axes)
```

ipymd.shared.transformations.**identity_matrix**()
Return 4x4 identity/unit matrix.

```
>>> I = identity_matrix()
>>> numpy.allclose(I, numpy.dot(I, I))
True
>>> numpy.sum(I), numpy.trace(I)
(4.0, 4.0)
>>> numpy.allclose(I, numpy.identity(4))
True
```

ipymd.shared.transformations.**inverse_matrix**(*matrix*)
Return inverse of square transformation matrix.

```
>>> M0 = random_rotation_matrix()
>>> M1 = inverse_matrix(M0.T)
>>> numpy.allclose(M1, numpy.linalg.inv(M0.T))
True
>>> for size in range(1, 7):
...     M0 = numpy.random.rand(size, size)
...     M1 = inverse_matrix(M0)
...     if not numpy.allclose(M1, numpy.linalg.inv(M0)): print(size)
```

ipymd.shared.transformations.**is_same_transform**(*matrix0*, *matrix1*)
Return True if two matrices perform same transformation.

```
>>> is_same_transform(numpy.identity(4), numpy.identity(4))
True
>>> is_same_transform(numpy.identity(4), random_rotation_matrix())
False
```

ipymd.shared.transformations.**normalized**(*x*)
Return the x vector normalized

ipymd.shared.transformations.**orthogonalization_matrix**(*lengths*, *angles*)
Return orthogonalization matrix for crystallographic cell coordinates.

Angles are expected in degrees.

The de-orthogonalization matrix is the inverse.

```
>>> O = orthogonalization_matrix([10, 10, 10], [90, 90, 90])
>>> numpy.allclose(O[:3, :3], numpy.identity(3, float) * 10)
True
>>> O = orthogonalization_matrix([9.8, 12.0, 15.5], [87.2, 80.7, 69.7])
>>> numpy.allclose(numpy.sum(O), 43.063229)
True
```

ipymd.shared.transformations.**projection_from_matrix**(*matrix*, *pseudo=False*)

Return projection plane and perspective point from projection matrix.

Return values are same as arguments for projection_matrix function: point, normal, direction, perspective, and pseudo.

```
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> persp = numpy.random.random(3) - 0.5
>>> P0 = projection_matrix(point, normal)
>>> result = projection_from_matrix(P0)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
>>> P0 = projection_matrix(point, normal, direct)
>>> result = projection_from_matrix(P0)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
>>> P0 = projection_matrix(point, normal, perspective=persp, pseudo=False)
>>> result = projection_from_matrix(P0, pseudo=False)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
>>> P0 = projection_matrix(point, normal, perspective=persp, pseudo=True)
>>> result = projection_from_matrix(P0, pseudo=True)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
```

ipymd.shared.transformations.**projection_matrix**(*point*, *normal*, *direction=None*, *perspective=None*, *pseudo=False*)

Return matrix to project onto plane defined by point and normal.

Using either perspective point, projection direction, or none of both.

If pseudo is True, perspective projections will preserve relative depth such that Perspective = dot(Orthogonal, PseudoPerspective).

```
>>> P = projection_matrix([0, 0, 0], [1, 0, 0])
>>> numpy.allclose(P[1:, 1:], numpy.identity(4)[1:, 1:])
True
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> persp = numpy.random.random(3) - 0.5
>>> P0 = projection_matrix(point, normal)
>>> P1 = projection_matrix(point, normal, direction=direct)
>>> P2 = projection_matrix(point, normal, perspective=persp)
```

```
    >>> P3 = projection_matrix(point, normal, perspective=persp, pseudo=True)
    >>> is_same_transform(P2, numpy.dot(P0, P3))
    True
    >>> P = projection_matrix([3, 0, 0], [1, 1, 0], [1, 0, 0])
    >>> v0 = (numpy.random.rand(4, 5) - 0.5) * 20
    >>> v0[3] = 1
    >>> v1 = numpy.dot(P, v0)
    >>> numpy.allclose(v1[1], v0[1])
    True
    >>> numpy.allclose(v1[0], 3-v1[1])
    True
```

ipymd.shared.transformations.**quaternion_about_axis**(*angle*, *axis*)

Return quaternion for rotation about axis.

```
    >>> q = quaternion_about_axis(0.123, [1, 0, 0])
    >>> numpy.allclose(q, [0.99810947, 0.06146124, 0, 0])
    True
```

ipymd.shared.transformations.**quaternion_conjugate**(*quaternion*)

Return conjugate of quaternion.

```
    >>> q0 = random_quaternion()
    >>> q1 = quaternion_conjugate(q0)
    >>> q1[0] == q0[0] and all(q1[1:] == -q0[1:])
    True
```

ipymd.shared.transformations.**quaternion_from_euler**(*ai*, *aj*, *ak*, *axes='sxyz'*)

Return quaternion from Euler angles and axis sequence.

ai, aj, ak : Euler's roll, pitch and yaw angles axes : One of 24 axis sequences as string or encoded tuple

```
    >>> q = quaternion_from_euler(1, 2, 3, 'ryxz')
    >>> numpy.allclose(q, [0.435953, 0.310622, -0.718287, 0.444435])
    True
```

ipymd.shared.transformations.**quaternion_from_matrix**(*matrix*, *isprecise=False*)

Return quaternion from rotation matrix.

If isprecise is True, the input matrix is assumed to be a precise rotation matrix and a faster algorithm is used.

```
    >>> q = quaternion_from_matrix(numpy.identity(4), True)
    >>> numpy.allclose(q, [1, 0, 0, 0])
    True
    >>> q = quaternion_from_matrix(numpy.diag([1, -1, -1, 1]))
    >>> numpy.allclose(q, [0, 1, 0, 0]) or numpy.allclose(q, [0, -1, 0, 0])
    True
    >>> R = rotation_matrix(0.123, (1, 2, 3))
    >>> q = quaternion_from_matrix(R, True)
    >>> numpy.allclose(q, [0.9981095, 0.0164262, 0.0328524, 0.0492786])
    True
    >>> R = [[-0.545, 0.797, 0.260, 0], [0.733, 0.603, -0.313, 0],
    ...      [-0.407, 0.021, -0.913, 0], [0, 0, 0, 1]]
    >>> q = quaternion_from_matrix(R)
    >>> numpy.allclose(q, [0.19069, 0.43736, 0.87485, -0.083611])
    True
    >>> R = [[0.395, 0.362, 0.843, 0], [-0.626, 0.796, -0.056, 0],
    ...      [-0.677, -0.498, 0.529, 0], [0, 0, 0, 1]]
    >>> q = quaternion_from_matrix(R)
    >>> numpy.allclose(q, [0.82336615, -0.13610694, 0.46344705, -0.29792603])
```

```
    True
    >>> R = random_rotation_matrix()
    >>> q = quaternion_from_matrix(R)
    >>> is_same_transform(R, quaternion_matrix(q))
    True
```

ipymd.shared.transformations.**quaternion_imag**(*quaternion*)

Return imaginary part of quaternion.

```
    >>> quaternion_imag([3, 0, 1, 2])
    array([ 0.,  1.,  2.])
```

ipymd.shared.transformations.**quaternion_inverse**(*quaternion*)

Return inverse of quaternion.

```
    >>> q0 = random_quaternion()
    >>> q1 = quaternion_inverse(q0)
    >>> numpy.allclose(quaternion_multiply(q0, q1), [1, 0, 0, 0])
    True
```

ipymd.shared.transformations.**quaternion_matrix**(*quaternion*)

Return homogeneous rotation matrix from quaternion.

```
    >>> M = quaternion_matrix([0.99810947, 0.06146124, 0, 0])
    >>> numpy.allclose(M, rotation_matrix(0.123, [1, 0, 0]))
    True
    >>> M = quaternion_matrix([1, 0, 0, 0])
    >>> numpy.allclose(M, numpy.identity(4))
    True
    >>> M = quaternion_matrix([0, 1, 0, 0])
    >>> numpy.allclose(M, numpy.diag([1, -1, -1, 1]))
    True
```

ipymd.shared.transformations.**quaternion_multiply**(*quaternion1*, *quaternion0*)

Return multiplication of two quaternions.

```
    >>> q = quaternion_multiply([4, 1, -2, 3], [8, -5, 6, 7])
    >>> numpy.allclose(q, [28, -44, -14, 48])
    True
```

ipymd.shared.transformations.**quaternion_real**(*quaternion*)

Return real part of quaternion.

```
    >>> quaternion_real([3, 0, 1, 2])
    3.0
```

ipymd.shared.transformations.**quaternion_slerp**(*quat0*, *quat1*, *fraction*, *spin=0*, *shortestpath=True*)

Return spherical linear interpolation between two quaternions.

```
    >>> q0 = random_quaternion()
    >>> q1 = random_quaternion()
    >>> q = quaternion_slerp(q0, q1, 0)
    >>> numpy.allclose(q, q0)
    True
    >>> q = quaternion_slerp(q0, q1, 1, 1)
    >>> numpy.allclose(q, q1)
    True
    >>> q = quaternion_slerp(q0, q1, 0.5)
    >>> angle = math.acos(numpy.dot(q0, q))
```

```
    >>> numpy.allclose(2, math.acos(numpy.dot(q0, q1)) / angle) or          numpy.allclose(2, math.ac
    True
```

ipymd.shared.transformations.**random_quaternion**(*rand=None*)

Return uniform random unit quaternion.

rand: array like or None Three independent random variables that are uniformly distributed between 0 and 1.

```
    >>> q = random_quaternion()
    >>> numpy.allclose(1, vector_norm(q))
    True
    >>> q = random_quaternion(numpy.random.random(3))
    >>> len(q.shape), q.shape[0]==4
    (1, True)
```

ipymd.shared.transformations.**random_rotation_matrix**(*rand=None*)

Return uniform random rotation matrix.

rand: array like Three independent random variables that are uniformly distributed between 0 and 1 for each returned quaternion.

```
    >>> R = random_rotation_matrix()
    >>> numpy.allclose(numpy.dot(R.T, R), numpy.identity(4))
    True
```

ipymd.shared.transformations.**random_vector**(*size*)

Return array of random doubles in the half-open interval [0.0, 1.0).

```
    >>> v = random_vector(10000)
    >>> numpy.all(v >= 0) and numpy.all(v < 1)
    True
    >>> v0 = random_vector(10)
    >>> v1 = random_vector(10)
    >>> numpy.any(v0 == v1)
    False
```

ipymd.shared.transformations.**reflection_from_matrix**(*matrix*)

Return mirror plane point and normal vector from reflection matrix.

```
    >>> v0 = numpy.random.random(3) - 0.5
    >>> v1 = numpy.random.random(3) - 0.5
    >>> M0 = reflection_matrix(v0, v1)
    >>> point, normal = reflection_from_matrix(M0)
    >>> M1 = reflection_matrix(point, normal)
    >>> is_same_transform(M0, M1)
    True
```

ipymd.shared.transformations.**reflection_matrix**(*point*, *normal*)

Return matrix to mirror at plane defined by point and normal vector.

```
    >>> v0 = numpy.random.random(4) - 0.5
    >>> v0[3] = 1.
    >>> v1 = numpy.random.random(3) - 0.5
    >>> R = reflection_matrix(v0, v1)
    >>> numpy.allclose(2, numpy.trace(R))
    True
    >>> numpy.allclose(v0, numpy.dot(R, v0))
    True
    >>> v2 = v0.copy()
    >>> v2[:3] += v1
```

```
    >>> v3 = v0.copy()
    >>> v2[:3] -= v1
    >>> numpy.allclose(v2, numpy.dot(R, v3))
    True
```

ipymd.shared.transformations.**rotate_vectors**(*vector*, *axis*, *theta*)

rotate the vector v clockwise about the given axis vector by theta degrees.

e.g. rotate([0,1,0],[0,0,1],90) -> [1,0,0]

vector [iterable or list of iterables] vector to rotate [x,y,z] or [[x1,y1,z1],[x2,y2,z2]]

axis [iterable] axis to rotate around [x0,y0,z0]

theta [float] rotation angle in degrees

ipymd.shared.transformations.**rotation_from_matrix**(*matrix*)

Return rotation angle and axis from rotation matrix.

```
    >>> angle = (random.random() - 0.5) * (2*math.pi)
    >>> direc = numpy.random.random(3) - 0.5
    >>> point = numpy.random.random(3) - 0.5
    >>> R0 = rotation_matrix(angle, direc, point)
    >>> angle, direc, point = rotation_from_matrix(R0)
    >>> R1 = rotation_matrix(angle, direc, point)
    >>> is_same_transform(R0, R1)
    True
```

ipymd.shared.transformations.**rotation_matrix**(*angle*, *direction*)

Create a rotation matrix corresponding to the rotation around a general axis by a specified angle.

R = dd^T + cos(a) (I - dd^T) + sin(a) skew(d)

Parameters

- **angle** – float a
- **direction** – array d

ipymd.shared.transformations.**scale_from_matrix**(*matrix*)

Return scaling factor, origin and direction from scaling matrix.

```
    >>> factor = random.random() * 10 - 5
    >>> origin = numpy.random.random(3) - 0.5
    >>> direct = numpy.random.random(3) - 0.5
    >>> S0 = scale_matrix(factor, origin)
    >>> factor, origin, direction = scale_from_matrix(S0)
    >>> S1 = scale_matrix(factor, origin, direction)
    >>> is_same_transform(S0, S1)
    True
    >>> S0 = scale_matrix(factor, origin, direct)
    >>> factor, origin, direction = scale_from_matrix(S0)
    >>> S1 = scale_matrix(factor, origin, direction)
    >>> is_same_transform(S0, S1)
    True
```

ipymd.shared.transformations.**scale_matrix**(*factor*, *origin=None*, *direction=None*)

Return matrix to scale by factor around origin in direction.

Use factor -1 for point symmetry.

```
    >>> v = (numpy.random.rand(4, 5) - 0.5) * 20
    >>> v[3] = 1
    >>> S = scale_matrix(-1.234)
    >>> numpy.allclose(numpy.dot(S, v)[:3], -1.234*v[:3])
    True
    >>> factor = random.random() * 10 - 5
    >>> origin = numpy.random.random(3) - 0.5
    >>> direct = numpy.random.random(3) - 0.5
    >>> S = scale_matrix(factor, origin)
    >>> S = scale_matrix(factor, origin, direct)
```

ipymd.shared.transformations.**shear_from_matrix**(*matrix*)

Return shear angle, direction and plane from shear matrix.

```
    >>> angle = (random.random() - 0.5) * 4*math.pi
    >>> direct = numpy.random.random(3) - 0.5
    >>> point = numpy.random.random(3) - 0.5
    >>> normal = numpy.cross(direct, numpy.random.random(3))
    >>> S0 = shear_matrix(angle, direct, point, normal)
    >>> angle, direct, point, normal = shear_from_matrix(S0)
    >>> S1 = shear_matrix(angle, direct, point, normal)
    >>> is_same_transform(S0, S1)
    True
```

ipymd.shared.transformations.**shear_matrix**(*angle*, *direction*, *point*, *normal*)

Return matrix to shear by angle along direction vector on shear plane.

The shear plane is defined by a point and normal vector. The direction vector must be orthogonal to the plane's normal vector.

A point P is transformed by the shear matrix into P" such that the vector P-P" is parallel to the direction vector and its extent is given by the angle of P-P'-P", where P' is the orthogonal projection of P onto the shear plane.

```
    >>> angle = (random.random() - 0.5) * 4*math.pi
    >>> direct = numpy.random.random(3) - 0.5
    >>> point = numpy.random.random(3) - 0.5
    >>> normal = numpy.cross(direct, numpy.random.random(3))
    >>> S = shear_matrix(angle, direct, point, normal)
    >>> numpy.allclose(1, numpy.linalg.det(S))
    True
```

ipymd.shared.transformations.**simple_clip_matrix**(*scale*, *znear*, *zfar*, *aspectratio=1.0*)

Given the parameters for a frustum returns a 4x4 perspective projection matrix

Parameters

- **scale** (*float*) –
- **znear, zfar** (*float*) – near/far plane z, float

Return: a 4x4 perspective matrix

ipymd.shared.transformations.**superimposition_matrix**(*v0*, *v1*, *scale=False*, *usesvd=True*)

Return matrix to transform given 3D point set into second point set.

v0 and v1 are shape (3, *) or (4, *) arrays of at least 3 points.

The parameters scale and usesvd are explained in the more general affine_matrix_from_points function.

The returned matrix is a similarity or Eucledian transformation matrix. This function has a fast C implementation in transformations.c.

```
    >>> v0 = numpy.random.rand(3, 10)
    >>> M = superimposition_matrix(v0, v0)
    >>> numpy.allclose(M, numpy.identity(4))
    True
    >>> R = random_rotation_matrix(numpy.random.random(3))
    >>> v0 = [[1,0,0], [0,1,0], [0,0,1], [1,1,1]]
    >>> v1 = numpy.dot(R, v0)
    >>> M = superimposition_matrix(v0, v1)
    >>> numpy.allclose(v1, numpy.dot(M, v0))
    True
    >>> v0 = (numpy.random.rand(4, 100) - 0.5) * 20
    >>> v0[3] = 1
    >>> v1 = numpy.dot(R, v0)
    >>> M = superimposition_matrix(v0, v1)
    >>> numpy.allclose(v1, numpy.dot(M, v0))
    True
    >>> S = scale_matrix(random.random())
    >>> T = translation_matrix(numpy.random.random(3)-0.5)
    >>> M = concatenate_matrices(T, R, S)
    >>> v1 = numpy.dot(M, v0)
    >>> v0[:3] += numpy.random.normal(0, 1e-9, 300).reshape(3, -1)
    >>> M = superimposition_matrix(v0, v1, scale=True)
    >>> numpy.allclose(v1, numpy.dot(M, v0))
    True
    >>> M = superimposition_matrix(v0, v1, scale=True, usesvd=False)
    >>> numpy.allclose(v1, numpy.dot(M, v0))
    True
    >>> v = numpy.empty((4, 100, 3))
    >>> v[:, :, 0] = v0
    >>> M = superimposition_matrix(v0, v1, scale=True, usesvd=False)
    >>> numpy.allclose(v1, numpy.dot(M, v[:, :, 0]))
    True
```

ipymd.shared.transformations.**transform_from_crytal**(*coords, a, b, c, origin=[0, 0, 0]*)

transform from crystal fractional coordinates to cartesian

coords : numpy.array((N,3))

a : numpy.array(3)

b : numpy.array(3)

c : numpy.array(3)

origin : numpy.array(3)

Notes

From https://en.wikipedia.org/wiki/Fractional_coordinates

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} a & b\cos(\gamma) & c\cos(\beta) \\ 0 & b\sin(\gamma) & c\frac{\cos(\alpha)-\cos(\beta)\cos(\gamma)}{\sin(\gamma)} \\ 0 & 0 & c\frac{v}{\sin(\gamma)} \end{bmatrix} \begin{bmatrix} x_{frac} \\ y_{frac} \\ z_{frac} \end{bmatrix}$$

such that v is the volume of a unit parallelepiped defined as:

$$v = \sqrt{1 - \cos^2(\alpha) - \cos^2(\beta) - \cos^2(\gamma) + 2\cos(\alpha)\cos(\beta)\cos(\gamma)}$$

ipymd.shared.transformations.**transform_to_crystal**(*coords, a, b, c, origin=[0, 0, 0]*)

transform from cartesian to crystal fractional coordinates

coords : numpy.array((N,3)) a : numpy.array(3) b : numpy.array(3) c : numpy.array(3) origin : numpy.array(3)

Notes

From https://en.wikipedia.org/wiki/Fractional_coordinates

$$\begin{bmatrix} x_{frac} \\ y_{frac} \\ z_{frac} \end{bmatrix} = \begin{bmatrix} \frac{1}{a} & -\frac{\cos(\gamma)}{a\sin(\gamma)} & \frac{\cos(\alpha)\cos(\gamma)-\cos(\beta)}{av\sin(\gamma)} \\ 0 & \frac{1}{b\sin(\gamma)} & \frac{\cos(\beta)\cos(\gamma)-\cos(\alpha)}{bv\sin(\gamma)} \\ 0 & 0 & \frac{\sin(\gamma)}{cv} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

such that v is the volume of a unit parallelepiped defined as:

$$v = \sqrt{1 - \cos^2(\alpha) - \cos^2(\beta) - \cos^2(\gamma) + 2\cos(\alpha)\cos(\beta)\cos(\gamma)}$$

ipymd.shared.transformations.**translation_from_matrix**(*matrix*)

Return translation vector from translation matrix.

```
>>> v0 = numpy.random.random(3) - 0.5
>>> v1 = translation_from_matrix(translation_matrix(v0))
>>> numpy.allclose(v0, v1)
True
```

ipymd.shared.transformations.**translation_matrix**(*direction*)

Return matrix to translate by direction vector.

```
>>> v = numpy.random.random(3) - 0.5
>>> numpy.allclose(v, translation_matrix(v)[:3, 3])
True
```

ipymd.shared.transformations.**unit_vector**(*data*, *axis=None*, *out=None*)

Return ndarray normalized by length, i.e. eucledian norm, along axis.

```
>>> v0 = numpy.random.random(3)
>>> v1 = unit_vector(v0)
>>> numpy.allclose(v1, v0 / numpy.linalg.norm(v0))
True
>>> v0 = numpy.random.rand(5, 4, 3)
>>> v1 = unit_vector(v0, axis=-1)
>>> v2 = v0 / numpy.expand_dims(numpy.sqrt(numpy.sum(v0*v0, axis=2)), 2)
>>> numpy.allclose(v1, v2)
True
>>> v1 = unit_vector(v0, axis=1)
>>> v2 = v0 / numpy.expand_dims(numpy.sqrt(numpy.sum(v0*v0, axis=1)), 1)
>>> numpy.allclose(v1, v2)
True
>>> v1 = numpy.empty((5, 4, 3))
>>> unit_vector(v0, axis=1, out=v1)
>>> numpy.allclose(v1, v2)
True
>>> list(unit_vector([]))
[]
>>> list(unit_vector([1]))
[1.0]
```

ipymd.shared.transformations.**vector_norm**(*data*, *axis=None*, *out=None*)

Return length, i.e. eucledian norm, of ndarray along axis.

```
>>> v = numpy.random.random(3)
>>> n = vector_norm(v)
>>> numpy.allclose(n, numpy.linalg.norm(v))
True
>>> v = numpy.random.rand(6, 5, 3)
>>> n = vector_norm(v, axis=-1)
>>> numpy.allclose(n, numpy.sqrt(numpy.sum(v*v, axis=2)))
True
>>> n = vector_norm(v, axis=1)
>>> numpy.allclose(n, numpy.sqrt(numpy.sum(v*v, axis=1)))
True
>>> v = numpy.random.rand(5, 4, 3)
>>> n = numpy.empty((5, 3))
>>> vector_norm(v, axis=1, out=n)
>>> numpy.allclose(n, numpy.sqrt(numpy.sum(v*v, axis=1)))
True
>>> vector_norm([])
0.0
>>> vector_norm([1])
1.0
```

ipymd.shared.transformations.**vector_product**(*v0*, *v1*, *axis=0*)

Return vector perpendicular to vectors.

```
>>> v = vector_product([2, 0, 0], [0, 3, 0])
>>> numpy.allclose(v, [0, 0, 6])
True
>>> v0 = [[2, 0, 0, 2], [0, 2, 0, 2], [0, 0, 2, 2]]
>>> v1 = [[3], [0], [0]]
>>> v = vector_product(v0, v1)
>>> numpy.allclose(v, [[0, 0, 0, 0], [0, 0, 6, 6], [0, -6, 0, -6]])
True
>>> v0 = [[2, 0, 0], [2, 0, 0], [0, 2, 0], [2, 0, 0]]
>>> v1 = [[0, 3, 0], [0, 0, 3], [0, 0, 3], [3, 3, 3]]
>>> v = vector_product(v0, v1, axis=1)
>>> numpy.allclose(v, [[0, 0, 6], [0, -6, 0], [6, 0, 0], [0, -6, 6]])
True
```

Module contents

ipymd.shared.**atom_data**()

return a dataframe of atomic data

ipymd.shared.**get_data_path**(*data*, *check_exists=False*, *module=<module 'ipymd.test_data' from '/home/docs/checkouts/readthedocs.org/user_builds/ipymd/checkouts/latest/ipymd/test_data/__*

return a directory path to data within a module

data [str or list of str] file name or list of sub-directories and file name (e.g. ['lammps','data.txt'])

ipymd.test_data package

Subpackages

ipymd.test_data.atom_dump package

Module contents Created on Sun May 1 22:46:22 2016

@author: cjs14

Module contents Created on Sun May 1 22:46:22 2016

@author: cjs14

ipymd.visualise package

Subpackages

ipymd.visualise.opengl package

Subpackages

ipymd.visualise.opengl.postprocessing package

Subpackages

ipymd.visualise.opengl.postprocessing.shaders package

Module contents Created on Sun May 1 22:46:22 2016

@author: cjs14

Submodules

ipymd.visualise.opengl.postprocessing.base module

class ipymd.visualise.opengl.postprocessing.base.**AbstractEffect**(**args*, ***kwargs*)

Bases: object

Interface for a generic post processing effect.

A subclass of AbstractEffect can be used by a QChemlabWidget to provide post-processing effects such as outlines, gamma correction, approximate anti-aliasing, or screen space ambient occlusion.

on_resize(*w*, *h*)

Optionally, subclasses can override on_resize. This method is useful if the post-processing effect requires additional creation of textures that need to hold multiple passes.

render(*fb*, *textures*)

Subclasses should override this method to draw the post-processing effect by using the framebuffer *fb* (represented as an integer generated by glGenFramebuffers).

The textures corresponding to the model rendering and the previous post-processing effects are passed through the dictionary *textures*.

The textures passed by default are "color", "depth" and "normal" and are instances of chemlab.graphics.Texture.

set_options(***options*)
Subclasses should use this method to change the options of the effect

ipymd.visualise.opengl.postprocessing.noeffect module

class ipymd.visualise.opengl.postprocessing.noeffect.**NoEffect**(*widget*)
Bases: *ipymd.visualise.opengl.postprocessing.base.AbstractEffect*

Re-render the object without implementing any effect.

This renderer serves as an example, and can be used to access the textures used for the rendering through the *texture* attribute.

This texture can be used to dump the image being rendered.

render(*fb*, *textures*)

Module contents

ipymd.visualise.opengl.renderers package

Subpackages

ipymd.visualise.opengl.renderers.opengl_shaders package

Module contents Created on Sun May 1 22:46:22 2016

@author: cjs14

Submodules

ipymd.visualise.opengl.renderers.atom module Created on Sun May 15 20:10:20 2016

@author: cjs14

added patch to allow for transparent atoms when using 'impostors' backend & changed to have pre-processing of colors and radii

class ipymd.visualise.opengl.renderers.atom.**AtomRenderer**(*widget*, *r_array*, *radii*, *colorlist*, *backend='impostors'*, *shading='phong'*, *transparent=True*)
Bases: *ipymd.visualise.opengl.renderers.base.AbstractRenderer*

Render atoms by using different rendering methods.

Parameters

widget: The parent QChemlabWidget

r_array: np.ndarray((NATOMS, 3), dtype=float) The atomic coordinate array

backend: "impostors" | "polygons" | "points" You can choose the rendering method between the sphere impostors, polygonal sphere and points.

change_shading(*shd*)

draw()

hide(*mask*)

update_colors(*cols*)

update_positions(*r_array*)
Update the atomic positions

update_radii(*radii*)

ipymd.visualise.opengl.renderers.base module

class ipymd.visualise.opengl.renderers.base.**AbstractRenderer**(*widget*, **args*, ***kwargs*)

Bases: object

AbstractRenderer is the standard interface for renderers. Each renderer have to implement an initialization function __init__ and a draw method to do the actual drawing using OpenGL or by using other, more basic, renderers.

Usually the renderers have also some custom functions that they use to update themselves. For example a SphereRenderer implements the function update_positions to move the spheres around without having to regenerate all of the other properties.

See also:

/graphics for a tutorial on how to develop a simple renderer.

Parameters

widget: chemlab.graphics.QChemlabWidget The parent *QChemlabWidget*. Renderers can use the widget to access the camera, lights, and other informations.

args, kwargs: Any other argument that they may use.

draw()
Generic drawing function to be implemented by the subclasses.

class ipymd.visualise.opengl.renderers.base.**DefaultRenderer**(*widget*)

Bases: *ipymd.visualise.opengl.renderers.base.ShaderBaseRenderer*

Same as ShaderBaseRenderer with the default shaders.

You can find the shaders in chemlab/graphics/renderers/shaders/ under the names of default_persp.vert and default_persp.frag.

draw_vertices()
Subclasses should reimplement this method.

setup_shader()

class ipymd.visualise.opengl.renderers.base.**ShaderBaseRenderer**(*widget*, *vertex*, *fragment*)

Bases: *ipymd.visualise.opengl.renderers.base.AbstractRenderer*

Instruments OpenGL with a vertex and a fragment shader.

This renderer automatically binds light and camera information. Subclasses should not reimplement the draw method but the draw_vertices method where you can bind and draw the objects.

Parameters

widget: The parent QChemlabWidget

vertex: str Vertex program as a string

fragment: str Fragment program as a string

compile_shader()

draw()

draw_vertices()
Method to be reimplemented by the subclasses.

setup_shader()

ipymd.visualise.opengl.renderers.box module Created on Mon May 16 10:53:56 2016

@author: cjs14

added patch to allow for line width selection

class ipymd.visualise.opengl.renderers.box.**BoxRenderer**(*widget*, *vectors*, *origin=<Mock object>*, *color=(0, 0, 0, 255)*, *width=1.5*)

Bases: *ipymd.visualise.opengl.renderers.base.ShaderBaseRenderer*

Used to render one wireframed box.

Parameters

widget: The parent QChemlabWidget

vectors: np.ndarray((3,3), dtype=float) The three vectors representing the sides of the box.

origin: np.ndarray((3,3), dtype=float), default to zero The origin of the box.

color: 4 int tuple r,g,b,a color in the range [0,255]

width: float width of wireframe lines

draw_vertices()

update(*vectors*)
Update the box vectors.

ipymd.visualise.opengl.renderers.hexagon module Created on Mon May 16 12:41:12 2016

@author: cjs14

class ipymd.visualise.opengl.renderers.hexagon.**HexagonRenderer**(*widget*, *vectors*, *origin=<Mock object>*, *color=(0, 0, 0, 255)*, *width=1.5*)

Bases: *ipymd.visualise.opengl.renderers.base.ShaderBaseRenderer*

Used to render one wireframed hexagonal prism.

Parameters

widget: The parent QChemlabWidget

vectors: np.ndarray((2,3), dtype=float) The two vectors representing the orthogonal a,c crystal vectors.

origin: np.ndarray((3,), dtype=float), default to zero The origin of the box.

color: 4 int tuple r,g,b,a color in the range [0,255]

width: float width of wireframe lines

draw_vertices()

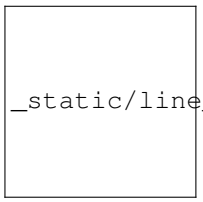
update(*vectors*)
Update the box vectors.

ipymd.visualise.opengl.renderers.line module

class ipymd.visualise.opengl.renderers.line.**LineRenderer**(*widget*, *startends*, *colors*, *width=1.5*)

Bases: *ipymd.visualise.opengl.renderers.base.ShaderBaseRenderer*

Render a set of lines.



Parameters

widget: The parent QChemlabWidget

startends: np.ndarray((NLINES, 2, 3), dtype=float) Start and end position of each line in the form of an array:

```
s1 = [0.0, 0.0, 0.0]
startends = [[s1, e1], [s2, e2], ..]
```

colors: np.ndarray((NLINES, 2, 4), dtype=np.uint8) The corresponding color of each extrema of each line.

draw_vertices()

update_colors(*colors*)
Update the colors

update_positions(*vertices*)
Update the line positions

ipymd.visualise.opengl.renderers.point module

class ipymd.visualise.opengl.renderers.point.**PointRenderer**(*widget*, *positions*, *colors*)

Bases: *ipymd.visualise.opengl.renderers.base.ShaderBaseRenderer*

Render colored points.

Parameters

widget: The parent QChemlabWidget

positons: np.ndarray((NPOINTS, 3), dtype=np.float32) Positions of the points to draw.

colors: np.ndarray((NPOINTS, 4), dtype=np.uint8) or list of tuples Color of each point in the (r,g,b,a) format in the interval [0, 255]

draw_vertices()

update_colors(*colors*)
Update the colors

update_positions(*vertices*)
Update the point positions

ipymd.visualise.opengl.renderers.sphere module

class ipymd.visualise.opengl.renderers.sphere.**Sphere**(*radius, center, parallels=20, meridians=15, color=[0.0, 0.0, 0.0, 0.0]*)

Bases: object

Create a Sphere object specifying its radius its center point. You can modulate its smoothness using the parallel and meridians settings.

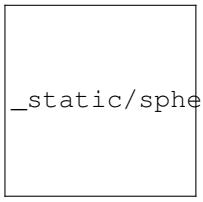
rotate(*axis*, *angle*)

class ipymd.visualise.opengl.renderers.sphere.**SphereRenderer**(*widget*, *poslist*, *radiuslist*, *colorlist*, *shading='phong'*)

Bases: *ipymd.visualise.opengl.renderers.base.AbstractRenderer*

Renders a set of spheres.

The method used by this renderer is approximating a sphere by using triangles. While this is reasonably fast, for best performance and animation you should use SphereImpostorRenderer



Parameters

widget: The parent QChemlabWidget

poslist: np.ndarray((NSPHERES, 3), dytpe=float) A position array. While there aren't dimensions, in the context of chemlab 1 unit of space equals 1 nm.

radiuslist: np.ndarray(NSPHERES, dtype=float) An array with the radius of each sphere.

colorlist: np.ndarray(NSPHERES, 4) or list of tuples An array with the color of each sphere. Suitable colors are those found in chemlab.graphics.colors or any tuple with values (r, g, b, a) in the range [0, 255]

draw()

update_colors(*colorlist*)

update_positions(*positions*)
Update the sphere positions.

ipymd.visualise.opengl.renderers.sphere_imp module

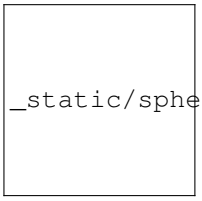
class ipymd.visualise.opengl.renderers.sphere_imp.**SphereImpostorRenderer**(*viewer*, *poslist*, *radiuslist*, *colorlist*, *transparent=False*, *shading='phong'*)

Bases: *ipymd.visualise.opengl.renderers.base.ShaderBaseRenderer*

The interface is identical to `SphereRenderer` but uses a different drawing method.

The spheres are squares that always face the user. Each point of the sphere, along with the lighting, is calculated in the fragment shader, resulting in a perfect sphere.

SphereImpostorRenderer is an extremely fast rendering method, it is perfect for rendering a lot of spheres (> 50000) and for animations.



change_shading(*shd_typ*)

draw()

hide(*mask*)

setup_shader()

update_colors(*colorlist*)

update_positions(*rarray*)

update_radii(*radiuslist*)

ipymd.visualise.opengl.renderers.triangle module Created on Mon May 16 09:55:43 2016

@author: cjs14

added patch to allow for transparent surface

class ipymd.visualise.opengl.renderers.triangle.**TriangleRenderer**(*widget*, *vertices*, *normals*, *colors*, *shading='phong'*, *transparent=False*, *wireframe=False*)

Bases: *ipymd.visualise.opengl.renderers.base.DefaultRenderer*

Renders an array of triangles.

A lot of renderers are built on this, for example `SphereRenderer`. The implementation is relatively fast since it's based on VertexBuffers.

Parameters

widget: The parent QChemlabWidget

vertices: np.ndarray((NTRIANGLES*3, 3), dtype=float) The triangle vertices, keeping in mind the unwinding order. If the face of the triangle is pointing outwards, the vertices should be provided in clokckwise order.

normals: np.ndarray((NTRIANGLES*3, 3), dtype=float) The normals to each of the triangle vertices, used for lighting calculations.

colors: np.ndarray((NTRIANGLES*3, 4), dtype=np.uint8) Color for each of the vertices in (r,g,b,a) values in the interval [0, 255]

draw_vertices()

setup_shader()

update_colors(*colors*)
Update the triangle colors.

update_normals(*normals*)
Update the triangle normals.

update_vertices(*vertices*)
Update the triangle vertices.

ipymd.visualise.opengl.renderers.triangles module TriangleRenderer is the basics for other shapes, we pass just triangle vertices and we got the result.

class ipymd.visualise.opengl.renderers.triangles.**TriangleRenderer**(*widget*, *vertices*, *normals*, *colors*, *shading='phong'*)

Bases: *ipymd.visualise.opengl.renderers.base.DefaultRenderer*

Renders an array of triangles.

A lot of renderers are built on this, for example `SphereRenderer`. The implementation is relatively fast since it's based on VertexBuffers.

Parameters

widget: The parent QChemlabWidget

vertices: np.ndarray((NTRIANGLES*3, 3), dtype=float) The triangle vertices, keeping in mind the unwinding order. If the face of the triangle is pointing outwards, the vertices should be provided in clokckwise order.

normals: np.ndarray((NTRIANGLES*3, 3), dtype=float) The normals to each of the triangle vertices, used for lighting calculations.

colors: np.ndarray((NTRIANGLES*3, 4), dtype=np.uint8) Color for each of the vertices in (r,g,b,a) values in the interval [0, 255]

draw_vertices()

setup_shader()

update_colors(*colors*)
Update the triangle colors.

update_normals(*normals*)
Update the triangle normals.

update_vertices(*vertices*)
Update the triangle vertices.

Module contents

Submodules

ipymd.visualise.opengl.buffers module

class ipymd.visualise.opengl.buffers.**VertexBuffer**(*data*, *usage*)
Bases: object

bind()

bind_attrib(*attribute*, *size*, *type*, *normalized=<Mock object>*, *stride=0*)

bind_colors(*size*, *type*, *stride=0*)

bind_edgeflags(*stride=0*)

bind_indexes(*type*, *stride=0*)

bind_normals(*type*, *stride=0*)

bind_texcoords(*size*, *type*, *stride=0*)

bind_vertexes(*size*, *type*, *stride=0*)

set_data(*data*)

unbind()

ipymd.visualise.opengl.camera module Module to provide a nice camera for 3d applications

class ipymd.visualise.opengl.camera.**Camera**
Our viewpoint on the 3D world. The Camera class can be used to access and modify from which point we're seeing the scene.

It also handle the projection matrix (the matrix we apply to project 3d points onto our 2d screen).

position

Type np.ndarray(3, float)

Default np.array([0.0, 0.0, 5.0])

The position of the camera. You can modify this attribute to move the camera in various directions using the absoule x, y and z coordinates.

a, b, c

Type np.ndarray(3), np.ndarray(3), np.ndarray(3) dtype=float

Default a: np.ndarray([1.0, 0.0, 0.0]) b: np.ndarray([0.0, 1.0, 0.0]) c: np.ndarray([0.0, 0.0, -1.0])

Those three vectors represent the camera orientation. The `a` vector points to our right, the `b` points upwards and `c` in front of us.

By default the camera points in the negative z-axis direction.

pivot

Type np.ndarray(3, dtype=float)

Default np.array([0.0, 0.0, 0.0])

The point we will orbit around by using *Camera.orbit_x()* and *Camera.orbit_y()*.

matrix

Type np.ndarray((4,4), dtype=float)

Camera matrix, it contains the rotations and translations needed to transform the world according to the camera position. It is generated from the `a`,''b'',''c'' vectors.

projection

Type np.ndarray((4, 4),dtype=float)

Projection matrix, generated from the projection parameters.

z_near, z_far

Type float, float

Near and far clipping planes. For more info refer to: http://www.lighthouse3d.com/tutorials/view-frustum-culling/

fov

Type float

field of view in degrees used to generate the projection matrix.

aspectratio

Type float

Aspect ratio for the projection matrix, this should be adapted when the application window is resized.

autozoom(*points*)

Fit the current view to the correct zoom level to display all *points*.

The camera viewing direction and rotation pivot match the geometric center of the points and the distance from that point is calculated in order for all points to be in the field of view. This is currently used to provide optimal visualization for molecules and systems

Parameters

points: np.ndarray((N, 3)) Array of points.

matrix

mouse_rotate(*dx*, *dy*)

Convenience function to implement the mouse rotation by giving two displacements in the x and y directions.

mouse_zoom(*inc*)

Convenience function to implement a zoom function.

This is achieved by moving `Camera.position` in the direction of the `Camera.c` vector.

orbit_x(*angle*)
Same as `orbit_y()` but the axis of rotation is the `Camera.b` vector.

We rotate around the point like if we sit on the side of a salad spinner.

orbit_y(*angle*)
Orbit around the point `Camera.pivot` by the angle *angle* expressed in radians. The axis of rotation is the camera "right" vector, `Camera.a`.

In practice, we move around a point like if we were on a Ferris wheel.

orbit_z(*angle*)

projection

restore(*state*)
Restore the camera state, passed as a *state* dictionary. You can obtain a previous state from the method *Camera.state*.

state()
Return the current camera state as a dictionary, it can be restored with *Camera.restore*.

unproject(*x*, *y*, *z=-1.0*)
Receive x and y as screen coordinates and returns a point in world coordinates.

This function comes in handy each time we have to convert a 2d mouse click to a 3d point in our space.

Parameters

x: float in the interval [-1.0, 1.0] Horizontal coordinate, -1.0 is leftmost, 1.0 is rightmost.

y: float in the interval [1.0, -1.0] Vertical coordinate, -1.0 is down, 1.0 is up.

z: float in the interval [1.0, -1.0] Depth, -1.0 is the near plane, that is exactly behind our screen, 1.0 is the far clipping plane.

Return type np.ndarray(3,dtype=float)

Returns The point in 3d coordinates (world coordinates).

ipymd.visualise.opengl.camera.**fequal**(*a*, *b*, *tol*)

ipymd.visualise.opengl.qchemlabwidget module

ipymd.visualise.opengl.qchemlabwidget.**create_color_texture**(*fb*, *width*, *height*)
ipymd.visualise.opengl.qchemlabwidget.**create_depth_texture**(*fb*, *width*, *height*)

ipymd.visualise.opengl.qchemlabwidget.**create_normal_texture**(*fb*, *width*, *height*)

ipymd.visualise.opengl.qtviewer module

class ipymd.visualise.opengl.qtviewer.**FpsDraw**(*parent*)
Bases: `object`

draw()

ipymd.visualise.opengl.shaders module

ipymd.visualise.opengl.shaders.**compileShader**(*source*, *shaderType*)
Compile shader source of given type

source – GLSL source-code for the shader shaderType – GLenum GL_VERTEX_SHADER, GL_FRAGMENT_SHADER, etc,

returns GLuint compiled shader reference raises RuntimeError when a compilation failure occurs

ipymd.visualise.opengl.shaders.**set_uniform**(*prog*, *uni*, *typ*, *value*)

ipymd.visualise.opengl.textures module Texture data structures

class ipymd.visualise.opengl.textures.**Texture**(*kind*, *width*, *height*, *intformat*, *format*, *dtype*, *data=None*)

Bases: object

bind()

delete()

empty()

set_parameter(*par*, *value*)

Module contents

Submodules

ipymd.visualise.visualise_sim module Created on Sun May 1 23:47:03 2016

@author: cjs14

class ipymd.visualise.visualise_sim.**Visualise_Sim**(*units='real'*)

Bases: object

For units *real*, these are the units:

mass = grams/mole distance = Angstroms time = femtoseconds energy = Kcal/mole velocity = Angstroms/femtosecond force = Kcal/mole-Angstrom torque = Kcal/mole temperature = Kelvin pressure = atmospheres dynamic viscosity = Poise charge = multiple of electron charge (1.0 is a proton) dipole = charge*Angstroms electric field = volts/Angstrom density = gram/cm^dim

add_atoms(*atoms_df*, *spheres=True*, *illustrate=False*)

add atoms to visualisation

atoms_df [pandas.DataFrame] a table of atom data, must contain columns; x, y, z, radius, color and transparency

spheres [bool] whether the atoms are rendered as spheres or points

illustrate [str] if True, atom shading is more indicative of an illustration

add_axes(*axes=[[1, 0, 0], [0, 1, 0], [0, 0, 1]], length=1.0, offset=(-1.2, 0.2), colors=('red', 'green', 'blue'), width=1.5*)

add axes

axes [np.array(3,3)] to turn off axes, set to None

axes_offset [tuple] x, y offset from top top-left atom

add_bonds(*atoms_df*, *bonds_df*, *cylinders=False*, *illustrate=False*, *linewidth=5*)

add bonds to visualisation

atoms_df [pandas.DataFrame] a table of atom data, must contain columns; x, y, z

bonds_df [list] a table of bond data, must contain; start, end, radius, color, transparency to/from refer to the atoms_df iloc (not necessarily the index number!)

cylinders [bool] whether the bonds are rendered as cylinders or lines

illustrate [str] if True, atom shading is more indicative of an illustration

add_box(*a, b, c, origin=[0, 0, 0], color='black', width=1*)
add wireframed box to visualisation

a [np.ndarray(3, dtype=float)] The a vectors representing the sides of the box.

b [np.ndarray(3, dtype=float)] The b vectors representing the sides of the box.

c [np.ndarray(3, dtype=float)] The c vectors representing the sides of the box.

origin [np.ndarray((3,3), dtype=float), default to zero]

The origin of the box.

color [str] the color of the wireframe, in chemlab colors

add_box_from_meta(*meta*, *color='black'*, *width=1*)
a shortcut for adding boxes using a panda.Series containing a,b,c,origin

add_hexagon(*vectors*, *origin=<Mock object>*, *color='black'*, *width=1*)
add wireframed hexagonal prism to visualisation

vectors [np.ndarray((2,3), dtype=float)] The two vectors representing the orthogonal a,c directions.

origin [np.ndarray((3,3), dtype=float), default to zero]

The origin of the hexagon (representing center of hexagon)

color [str] the color of the wireframe, in chemlab colors

add_plane(*vectors*, *origin=<Mock object>*, *rev_normal=False*, *color='red'*, *alpha=1.0*)
add square plane to visualisation

vectors [np.ndarray((2,3), dtype=float)] The two vectors representing the edges of the plane.

origin [np.ndarray((3,3), dtype=float), default to zero] The origin of the plane.

rev_normal [bool] whether to reverse direction of normal (for lighting calculations)

color [str] the color of the plane, in chemlab colors

basic_vis(*atoms_df=None*, *meta=None*, *spheres=True*, *illustrate=False*, *xrot=0*, *yrot=0*, *zrot=0*, *fov=10.0*, *axes=<Mock object>*, *axes_length=1.0*, *axes_offset=(-1.2, 0.2)*, *size=400*, *quality=5*)
basic visualisation shortcut

invoking add_atoms, add_box (if meta), add_axes, get_image and visualise functions

create_textline_image(*text*, *fontsize=10*, *color=(0, 0, 0)*, *background=(255, 255, 255)*, *boxsize=(1000, 20)*)
create a PIL image from a line of text

get_image(*xrot=0*, *yrot=0*, *zrot=0*, *fov=5.0*, *size=400*, *quality=5*, *zoom_extents=None*, *trim_whitespace=True*)
get image of visualisation

NB: x-axis horizontal, y-axis vertical, z-axis out of page

Parameters

- **rotx** (*float*) – rotation about x (degrees)
- **roty** (*float*) – rotation about y (degrees)
- **rotz** (*float*) – rotation about z (degrees)

- **fov** (*float*) – field of view angle (degrees)
- **size** (*float*) – size of image
- **quality** (*float*) – quality of image (pixels per point), note: higher quality will take longer to render
- **zoom_extents** (*None or np.ndarray((N, 3))*) – define an array of points to autozoom image, if None then computed automatically
- **trim_whitespace** (*bool*) – whether to trim whitspace around image

Returns image

Return type PIL.Image

open_qtview(*xrot=0*, *yrot=0*, *zrot=0*, *fov=5.0*, *thickness=1*, *zoom_extents=None*)
open a qt viewer of the objects

Parameters

- **rotx** (*float*) – rotation about x (degrees)
- **roty** (*float*) – rotation about y (degrees)
- **rotz** (*float*) – rotation about z (degrees)
- **fov** (*float*) – field of view angle (degrees)
- **thickness** (*float*) – multiplier for thickness of lines for some objects
- **zoom_extents** (*None or np.ndarray((N, 3))*) – define an array of points to autozoom image, if None then computed automatically

Returns **viewer** – the qt viewing window

Return type PyQt4.QtGui.QMainWindow

remove_all_objects()

remove_atoms(*n=1*)
remove the last n sets of atoms to be added

remove_bonds(*n=1*)
remove the last n sets of bonds to be added

remove_boxes(*n=1*)
remove the last n boxes to be added

remove_hexagons(*n=1*)
remove the last n boxes to be added

remove_planes(*n=1*)
remove the last n planes to be added

visualise(*images*, *columns=1*, *width=None*, *height=None*)
visualise image(s) in IPython

When this object is returned by an input cell or passed to the display function, it will result in the image being displayed in the frontend.

Parameters

- **images** (*list/single PIL.Image or (x,y)*) – (x,y) denotes a blank space of size x,y e.g. [img1,(100,0),img2]
- **columns** (*int*) – number of image columns

- **width** (*int*) – Width to which to constrain the image in html
- **height** (*int*) – Height to which to constrain the image in html

Returns image

Return type IPython.display.Image

Module contents

Submodules

ipymd.atom_manipulation module

Created on Mon May 16 08:15:13 2016

@author: cjs14

class ipymd.atom_manipulation.**Atom_Manipulation**(*atom_df*, *meta_series=None*, *undos=1*)
Bases: object

a class to manipulate atom data

atom_df [pandas.DataFrame] containing columns; x, y, z, type

meta_series [pandas.Series] containing columns; origin, a, b, c to define unit cell if none it will be constructed from the min/max x, y, z values

undos [int] number of past dataframes to save

apply_map(*vmap*, *column*, *default=False*, *type_col='type'*)
change values in a column, according to a mapping of another column

vmap [dict or str] A dictionary mapping values, or a string associated with a column in the ipymd.shared.atom_data() dataframe (e.g. color and RVdW)

column [str] the column to change

default [various] the default value to put when the type key cannot be found, if False then the original value will not be overwritten

change_type_variable(*atom_type*, *variable*, *value*, *type_col='type'*)
change particular variable for one atom type

change_variables(*map_dict*, *vtype='type'*)
change particular variables according to the map_dict

color_by_categories(*colname*, *cmap='jet'*, *sort=True*)
change colors to map

colname [string] a column of the dataframe that contains categories by which to color

cmap [string] the colormap to apply, see available at http://matplotlib.org/examples/color/colormaps_reference.html

color_by_index(*cmap='jet'*, *minv=None*, *maxv=None*)
change colors to map index values

cmap [string] the colormap to apply, see available at http://matplotlib.org/examples/color/colormaps_reference.html

minv, maxv [float] optional min, max cmap value, otherwise take min, max value found in column

color_by_variable(*colname*, *cmap='jet'*, *minv=None*, *maxv=None*)
change colors to map

colname [string] a coloumn of the dataframe that contains numbers by which to color

cmap [string] the colormap to apply, see available at http://matplotlib.org/examples/color/colormaps_reference.html

minv, maxv [float] optional min, max cmap value, otherwise take min, max value found in column

df

filter_inside_box(*vectors*, *origin=<Mock object>*)
return only atoms inside box

vectors [np.array((3,3))] a, b, c vectors

origin : np.array((1,3))

filter_inside_hexagon(*vectors*, *origin=<Mock object>*)
return only atoms inside hexagonal prism

vectors [np.array((2,3))] a, c vectors

origin : np.array((1,3))

filter_inside_pts(*points*)
return only atoms inside the bounding shape of a set of points

points : np.array((N,3))

filter_variables(*values*, *vtype='type'*)

group_atoms_as_mols(*atom_ids*, *name*, *remove_atoms=True*, *mean_xyz=True*, *color='red'*, *transparency=1.0*, *radius=1.0*)
combine atoms into a molecule atom_ids : list of lists

list of dataframe indexes for each molecule

name [string] name of molecule

remove_atoms [bool] remove the grouped atoms from the dataframe

mean_xyz [bool] use the mean coordinate of atoms for molecule, otherwise use coordinate of first atom

meta

repeat_cell(*a=1*, *b=1*, *c=1*, *original_first=False*)
repeat atoms along a, b, c directions (and update unit cell)

a [int or tuple] repeats in 'a' direction, if tuple then defines repeats in -/+ direction

b [int or tuple] repeats in 'b' direction, if tuple then defines repeats in -/+ direction

c [int or tuple] repeats in 'c' direction, if tuple then defines repeats in -/+ direction

original_first: bool if True, the original atoms will be first in the DataFrame

revert_to_original()
revert to original atom_df

rotate(*angle, vector=[1, 0, 0], update_uc=True*)
rotate the clockwise about the given axis vector by theta degrees.

e.g. for rotate_atoms(90,[0,0,1]); [0,1,0] -> [1,0,0]

angle [float] rotation angle in degrees

vector [iterable] vector to rotate around [x0,y0,z0]

update_uc [bool] update unit cell (a,b,c,origin) to match rotation

slice_absolute(*amin=0*, *amax=None*, *bmin=0*, *bmax=None*, *cmin=0*, *cmax=None*, *incl_max=False*, *update_uc=True*, *delta=0.01*)
slice along a,b,c directions (from origin) given absolute vector length

if amax, bmax or cmax is None, then will use the vector length

update_uc [bool] update unit cell (a,b,c,origin) to match slice

incl_max [bool] whether to slice < (False) <= (True) max values

delta [float] retain atoms within 'delta' fraction outside of slice plane)

slice_fraction(*amin=0*, *amax=1*, *bmin=0*, *bmax=1*, *cmin=0*, *cmax=1*, *incl_max=False*, *update_uc=True*, *delta=0.01*)
slice along a,b,c directions (from origin) as fraction of vector length

incl_max [bool] whether to slice < (False) <= (True) max values

update_uc [bool] update unit cell (a,b,c,origin) to match slice

delta [float] retain atoms within 'delta' fraction outside of slice plane)

translate(*vector*, *update_uc=True*)
translate atoms by vector

vector [list] x, y, z translation

update_uc [bool] update unit cell (a,b,c,origin) to match translation

undo_last()

ipymd.data_output module

Created on Mon May 23 17:55:14 2016

@author: cjs14

class ipymd.data_output.**Data_Output**(*atom_df*, *abc*, *origin=<Mock object>*)
Bases: object

save_lammps(*outpath='out.lammps'*, *overwrite=False*, *atom_type='atomic'*, *header=''*, *mass_map={}*)
to adhere to http://lammps.sandia.gov/doc/read_data.html?highlight=read_data

Parameters

- **outpath** (*string*) – the output file name
- **overwrite** (*bool*) – whether to raise an error if the file already exists
- **atom_type** (*str*) – the lammps atom style, currently supports atomic or charge
- **header** (*str*) – text to put in the header
- **mass_map** (*dict*) – a mapping of atom types to mass

Example

In [1]: import pandas as pd In [2]: df = pd.DataFrame([['Fe',2,3,4,1],

['Cr',2,3,3,-1], ['Fe',4,3,1,1]],columns=['type','xs','ys','zs','q'])

In [3]: from ipymd.data_output import Data_Output as data_out In [4]: data = data_out(df, [[1,0,0],[0,1,0],[0,0,1]]) In [5]: data.save_lammps('test.lammps', atom_type='charge', overwrite=True,

header='my header')

In [6]: cat test.lammps # This file was created by ipymd (v0.0.1) on 2016-05-23 20:51:16 # type map: {'Cr': 2, 'Fe': 1} # my header

3 atoms 2 atom types

simulation box boundaries 0.0000 1.0000 xlo xhi 0.0000 1.0000 ylo yhi 0.0000 1.0000 zlo zhi 0.0000 0.0000 0.0000 xy xz yz

Atoms

1 1 1.0000 2.0000 3.0000 4.0000 2 2 -1.0000 2.0000 3.0000 3.0000 3 1 1.0000 4.0000 3.0000 1.0000

Module contents

Created on Sun May 1 22:46:22 2016

@author: cjs14

ipymd.**version**()

CHAPTER 2

License

ipymd is released under the GNU GPL3 or GNU LGPL license, if the PyQt parts are omitted (in ipymd.visualise.opengl) and the ipymd.data_input.spacegroup package is omitted, ipyMD is released under the GNU GPLv3 and its main developer is Chris Sewell.

[ref1] 1.Coleman, S. P., Sichani, M. M. & Spearot, D. E. A Computational Algorithm to Produce Virtual X-ray and Electron Diffraction Patterns from Atomistic Simulations. JOM 66, 408–416 (2014).

i

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

X