

Visualisation of 3D atomic and electronic data

Author:
Chris Sewell

Imperial College London
South Kensington, London

1st August 2017

Contents

1	Introduction	3
2	Creating Atomic Configurations	4
3	Preparation for visualisation	5
3.1	Geometry Manipulation	6
4	Visualising in the Jupyter Notebook	7
5	Adding Dynamic Controls	8
6	Overlaying electronic level data	9
6.1	Data Parsing	9
6.2	Visualisation Preparation	9
6.3	Visualisation	10
7	2D Representations	12
8	Publishing and Distributing Analysis	13
9	TODO	14

List of Figures

4.1	an example of an ipyvolume scatter plot	7
5.1	an example of an ipyvolume scatter plot (with bespoke controls)	8
6.1	an example of an ipyvolume volume and scatter plot (with bespoke controls)	11
7.1	an example of an 2D scatter plot (with depth perception)	12
9.1	an example of nearest-neighbour polygons	14

List of Tables

3.1	The first rows of the atomic data lookup.	5
-----	---	---

1 Introduction

With the improvements in Jupyter Notebook, allowing for the synergy of browser-side javascript and client-side python coding, it is becoming possible to replicate the functionality of standalone atomic visualisation packages (such as [ovito](#)). The added benefits this approach brings is:

- Greater control and flexibility in the analysis and visualisation process
- Fully autonomous replication of the analysis and visualisation
- Better documentation of the analysis and visualisation

The goal of this notebook is to show a method for:

1. reading/creating atomic configurations
2. visualising these in a Jupyter Notebook
3. Adding dynamic controls
4. Overlaying electronic level data (probability/spin densities)
5. Distributing the output

2 Creating Atomic Configurations

The [pymatgen](#) package offers a means to create/manipulate atomic configurations with repeating boundary conditions.

For this demonstration, we would like to select structures from a folder of cif (crystallographic information files). This is achieved by writing a *plugin* for [jsonextended](#), which is a package parsing file types into a json format and subsequent manipulation.

```
FeS_greigite.cif: {...}
FeS_mackinawite.cif: {...}
FeS_marcasite.cif: {...}
FeS_pyrite.cif: {...}
FeS_pyrrhotite_4C_c2c.cif: {...}
FeS_troilite.cif: {...}
Fe_bcc.cif: {...}
S_8alpha_fddd.cif: {...}
S_8beta_p21c.cif: {...}
```

For each cif, we can now access a pymatgen structure:

```
Structure Summary
Lattice
  abc : 3.6735000000000002 3.6735000000000002 5.0327999999999999
  angles : 90.0 90.0 90.0
  volume : 67.91563420380001
    A : 3.6735000000000002 0.0 2.2493700083339009e-16
    B : -2.2493700083339009e-16 3.6735000000000002 2.2493700083339009
e-16
    C : 0.0 0.0 5.0327999999999999
PeriodicSite: Fe (0.0000, 0.0000, 0.0000) [0.0000, 0.0000, 0.0000]
PeriodicSite: Fe (1.8367, 1.8368, 0.0000) [0.5000, 0.5000, 0.0000]
PeriodicSite: S (-0.0000, 1.8368, 1.3095) [0.0000, 0.5000, 0.2602]
PeriodicSite: S (1.8368, 0.0000, 3.7233) [0.5000, 0.0000, 0.7398]
```

3 Preparation for visualisation

A visualisation requires the configuration to contain some additional information, including the atom shape (e.g. sphere radius) and texture (e.g. sphere color).

Therefore, it will be helpful to create a view agnostic (i.e. independendant of any specific graphics package) representation of all elements we wish to visualise. We do this by deconstructing the pymatgen structure and applying a mapping of atomic number to radius/color, using a pre-constructed csv table.

Table 3.1: The first rows of the atomic data lookup.

	Blue	ElAffinity	ElNeg	Green	Ionization	Mass	Name	RBO	RCov	RVdW	Red	Symbol
1	0.75	0.75	2.20	0.75	13.60	1.01	Hydrogen	0.31	0.31	1.10	0.75	H
2	1.00	0.00	0.00	1.00	24.59	4.00	Helium	0.28	0.28	1.40	0.85	He
3	1.00	0.62	0.98	0.50	5.39	6.94	Lithium	1.28	1.28	1.81	0.80	Li
4	0.00	0.00	1.57	1.00	9.32	9.01	Beryllium	0.96	0.96	1.53	0.76	Be
5	0.71	0.28	2.04	0.71	8.30	10.81	Boron	0.84	0.84	1.92	1.00	B

```
mackinawite_Fe:
  cell_vectors:
    a: [ 3.67350000e+00 0.00000000e+00 2.24937001e-16]
    b: [ -2.24937001e-16 3.67350000e+00 2.24937001e-16]
    c: [ 0. 0. 5.0328]
  centre: [ 1.83675 1.83675 2.5164 ]
  color: rgb(224,102,51)
  coords: [[ 0. 0. 0.], [ 1.83675000e+00 1.83675000e+00
    2.24937001e-16]]
  label: Fe
  radius: 1.32
  transparency: 1.0
  type: scatter
  visible: [True, True]
mackinawite_S:
  cell_vectors:
    a: [ 3.67350000e+00 0.00000000e+00 2.24937001e-16]
    b: [ -2.24937001e-16 3.67350000e+00 2.24937001e-16]
    c: [ 0. 0. 5.0328]
  centre: [ 1.83675 1.83675 2.5164 ]
  color: rgb(178,178,0)
  coords: [[ -1.12468500e-16 1.83675000e+00 1.30953456e+00], [
    1.83675 0. 3.72326544]]
  label: S
  radius: 1.05
  transparency: 1.0
  type: scatter
  visible: [True, True]
```

Since the representation is in a JSON format, it makes it very easy to extend to new types of elements. Note that we group atoms with the same visual representations, rather than specifying each atom separately. This is because it will be more efficient for the rendering process (see [here](#) for an explanation). Here we do this by atomic number, but equally it could be done by symmetry equivalence or another metric.

3.1 Geometry Manipulation

We will also likely want to:

- create a supercell of the configuration
- orientate the configuration in a convenient manner in the cartesian coordinate space
- slice into the configuration

We can group these operations into a class, which is extensible to more geometric operation and element types.

```
mackinawite_Fe:
  cell_vectors:
    a: [ -6.74811003e-16 -1.10205000e+01 -1.47911420e-31]
    b: [ 1.10205000e+01 -4.13202567e-32 -1.47911420e-31]
    c: [ 9.24510362e-16 -9.24510362e-16 1.50984000e+01]
  centre: [ 0. 0. 0.]
  color: rgb(224,102,51)
  coords: [[-5.51025 5.51025 -7.5492 ], {\ldots}(x53)]
  label: Fe
  radius: 1.32
  transparency: 1.0
  type: scatter
  visible: [True, {\ldots}(x53)]
mackinawite_S:
  cell_vectors:
    a: [ -6.74811003e-16 -1.10205000e+01 -1.47911420e-31]
    b: [ 1.10205000e+01 -4.13202567e-32 -1.47911420e-31]
    c: [ 9.24510362e-16 -9.24510362e-16 1.50984000e+01]
  centre: [ 0. 0. 0.]
  color: rgb(178,178,0)
  coords: [[-3.6735 5.51025 -6.23966544], {\ldots}(x53)]
  label: S
  radius: 1.05
  transparency: 1.0
  type: scatter
  visible: [True, {\ldots}(x53)]
```

4 Visualising in the Jupyter Notebook

To create 3D renderings of the configuration, we will use [ipyvolume](#) and its implementation of the model/view pattern.

A Jupyter Widget

The rendering can also be captured as a screenshot or saved as an image/html. We shall discuss in Section ?? how this can be utilised for to distribute the analysis.

A Jupyter Widget

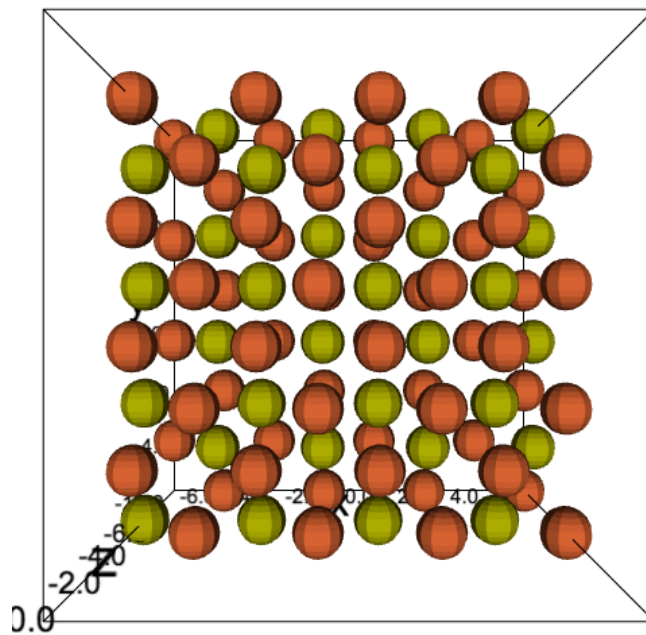


Figure 4.1: an example of an ipyvolume scatter plot

5 Adding Dynamic Controls

`ipyvolume` utilises the `ipywidgets` framework and thus it is relatively trivial to set up dynamic controls.

```
A Jupyter Widget
```

We can bundle these in with the original container to create a bespoke GUI.

```
A Jupyter Widget
```

```
A Jupyter Widget
```

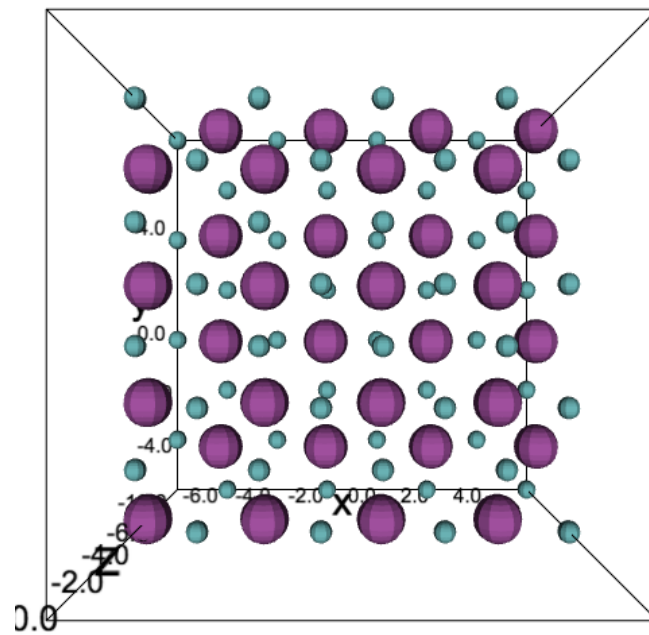


Figure 5.1: an example of an `ipyvolume` scatter plot (with bespoke controls)

6 Overlaying electronic level data

Ab initio quantum simulation packages can compute electronic/spin densities (to accompany the nuclei positions) in the form of a discretized 3D cube. These can be overlayed onto the nuclei, by [volume rendering](#) or [isosurface](#) methods.

6.1 Data Parsing

Taking the [CRYSTAL](#) program as an example, output from the electronic density is principally output into two files; one that contains the lattice vectors and nuclei coordinates and one that contains a data cube of the electronic density, with axis relating to the cell vectors. We can write parser plugins for both these files:

```
ech3.out:
  structure: Full Formula (Si2)
            Reduced Formula: Si
            abc      :   3.832519   3.832519   3.832519
            angles:  60.000000  60.000000  60.000000
            Sites (2)
              #  SP      a      b      c
              ---  ---  -
              0  Si      0.125092  0.125092  0.125092
              1  Si      0.874908  0.875277  0.874908
ech3_dat.prop3d:
  charge_density: np.array((100, 100, 100), min=2.68E-03, max=5.36E
+02)
  da_vec: [ 0. 0.051729 0.051729]
  db_vec: [ 0.051729 0. 0.051729]
  dc_vec: [ 0.051729 0.051729 0. ]
  na:      100
  nb:      100
  nc:      100
  o_vec: [0.0, 0.0, 0.0]
```

6.2 Visualisation Preparation

We then, follow the same process as for atoms; converting to a common structure and adding geometric manipulation functions for this data type.

```
Silicon Charge:
  cell_vectors:
    a: [ 0. 2.71 2.71]
    b: [ 2.71 0. 2.71]
    c: [ 2.71 2.71 0. ]
  centre: [ 2.71 2.71 2.71]
  dcube: np.array((100, 100, 100), min=2.68E-03, max=5.36E+02)
  slices: []
  type: volume
Silicon_Si:
  cell_vectors:
    a: [ 0. 2.71 2.71]
    b: [ 2.71 0. 2.71]
    c: [ 2.71 2.71 0. ]
```

```

centre: [ 2.71 2.71 2.71]
color: rgb(127,153,153)
coords: [[ 0.678 0.678 0.678], [ 4.743 4.742 4.743]]
label: Si
radius: 1.11
transparency: 1.0
type: scatter
visible: [True, True]

```

```

Silicon Charge:
  cell_vectors:
    a: [ 0. 10.84 10.84]
    b: [ 10.84 0. 10.84]
    c: [ 10.84 10.84 0. ]
  centre: [ 0. 0. 0.]
  dcube: np.array((400, 400, 400), min=2.68E-03, max=5.36E+02)
  slices: [[([ 0. 0. 1.], ...(x3))]]
  type: volume
Silicon_Si:
  cell_vectors:
    a: [ 0. 10.84 10.84]
    b: [ 10.84 0. 10.84]
    c: [ 10.84 10.84 0. ]
  centre: [ 0. 0. 0.]
  color: rgb(127,153,153)
  coords: [[-10.162 -10.162 -10.162], ...(x127)]
  label: Si
  radius: 1.11
  transparency: 1.0
  type: scatter
  visible: [True, ...(x127)]

```

6.3 Visualisation

For ipyvolume, at present, the volume data must be a cube of equal dimensions. Therefore, we use the cell vectors to transform the data cube into cartesian coordinates, such that voxels (cube sections) outside of the cell volume are set as `np.nan` values. We also resize the discretisation of the cube to an appropriate size for the renderer to handle.

Additionally, for ipyvolume (in its current state), there can only be one volume rendering per scene and it is assumed that the volumes bottom left corner is at (0,0,0).

```

cell_vectors:
  a: [ 0. 10.84 10.84]
  b: [ 10.84 0. 10.84]
  c: [ 10.84 10.84 0. ]
centre: [ 0. 0. 0.]
dcube: np.array((400, 400, 400), min=2.68E-03, max=5.36E+02)
slices: [[([ 0. 1. 1.], -2, 0, None)]]
type: volume

```

A Jupyter Widget

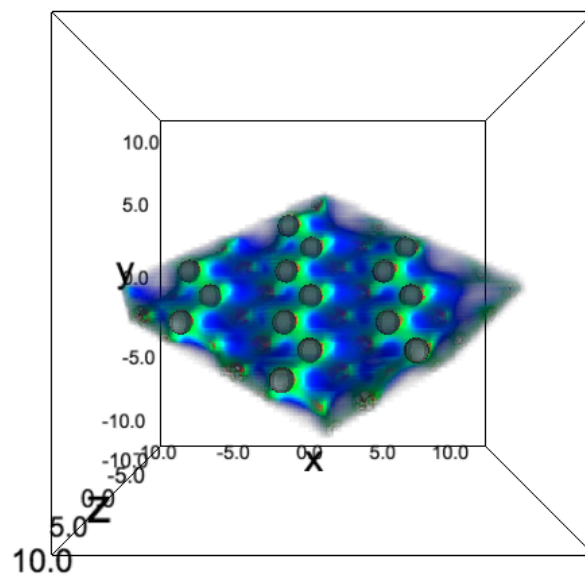


Figure 6.1: an example of an ipyvolume volume and scatter plot (with bespoke controls)

7 2D Representations

Because the data is stored in a representation agnostic manner, this allows for the possibility of displaying the data in multiple ways. In particular, for publication quality images we may want to create a 2D-representation of the scene. Below we plot the atoms with a depth perception effect, created by lightening the color of the atoms w.r.t their depth into the page.

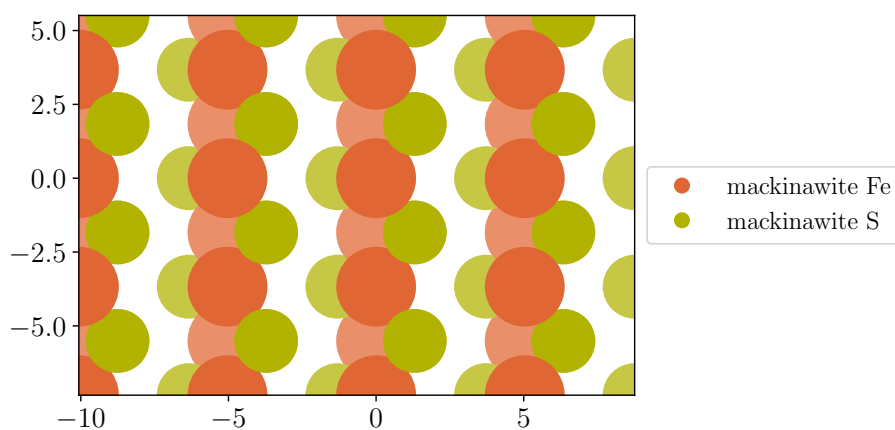


Figure 7.1: an example of an 2D scatter plot (with depth perception)

8 Publishing and Distributing Analysis

As discussed above, individual visualisations can be saved individually as images or HTML. But a more complete solution is to use [ipypublish](#) to convert the entire Jupyter Notebook to a document and/or presentation. `ipypublish` utilises notebook/cell/output level metadata attributes to define a greater level of control as to how elements in the notebook are converted.

This entire document is a single notebook which is available as a [Notebook](#), [PDF](#), [HTML](#) or [RevealJS slideshow](#) document (click the hyperlinks to view them). This was achieved by only the following command line commands:

```
$ nbpublish -pdf -ptemp -f latex_ipypublish_nocode "3D Atomic Visualisation.ipynb"
$ nbpublish -f html_ipypublish_all "3D Atomic Visualisation.ipynb"
$ nbpresent "3D Atomic Visualisation.ipynb"
```

9 TODO

- Orthographic camera. Not yet implemented in ipyvolume, see [this issue](#) for current status.
- better control of spheres
 - exact control of radii (radius rather than scaling size)
 - more segments (either direct control of segments, or a "sphere_hi_res" type)
 - transparency level
 - should color allow (r,g,b) tuple/array? because at the moment that doesn't work
- creation of array of arbitrary lines (like scatter but with; x0,y0,z0,x1,y1,z1)
 - show lattice bounding boxes: parallelepiped wire frames
 - show bonds (i.e. connections) between different scatters
- show nearest-neighbour coordination: polygons with vertices at nearest-neighbour positions (as shown in fig. 9.1)

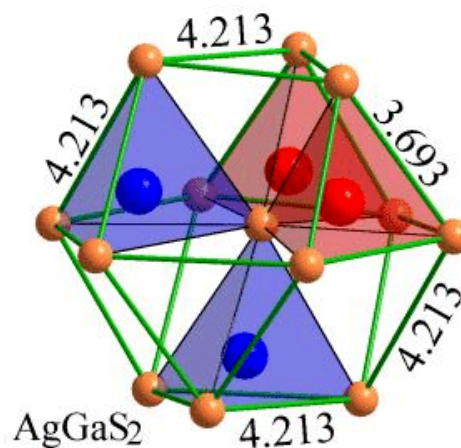


Figure 9.1: an example of nearest-neighbour polygons

- functional (browser side) controls, e.g. slider to translate/rotate point set. Not yet implemented in ipywidgets, see [this issue](#) for current status.
- volumes:
 - RuntimeError: invalid value encountered in true_divide (serialize.py:43) presumably for (0,0,0) gradients
 - rarely get artifact rendering
 - isosurface rendering
 - multiple volumes in single plot
 - volumes with arbitrary centres
 - rotating volumes
- fullscreen
 - fails to open if multiple views instantiated
 - if volume is present, then the rendering becomes very low resolution and, sometimes, the volume disappears completely on exit
- 2d volume representation
 - define slice into cube and use matplotlib.contour