Visualisation of 3D atomic and electronic data in the Jupyter Notebook

Author:
Chris Sewell

Imperial College London
South Kensington, London

8th August 2017

Contents

List of Figures

List of Tables

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

The notebook and code used for this demonstration can be found here.

2 A Quick Introduction to JSON

A recommended practice for structuring hierarchical data (which will be used throughout this demonstration) is to use the JSON format because it is:

- applicable for any (non-relational) data structure
- lightweight and easy to read and edit
- has a simple read/write mapping to python objects
- widely used (especially in web technologies)

A good example of how this structure can being applied to quantum chamical data can be found here and, in fact, this Jupyter Notebook itself is stored in JSON format:

```
cells:           [...]
metadata:        {...}
nbformat:        4
nbformat_minor: 2
```

Note that Python has a built-in `json` package, but we shall use the jsonextended package, which has extended functionality for data parsing, manipulation and visualisation.

2.1 JSON Schema

Rather than strictly controlling the entire data structure from the outset, before acting on the data, we can simply validate that it contains the required data keys and types required. This is in keeping with the interpreted (as opposed to declarative) nature of Python.

Another advantage of json is that we can utilise the standard json schema approach to achieve this validation. For example, a basic Jupyter Notebook schema would look like this:

```
properties:
  cells:
    items:
      properties:
        cell_type:
          enum: [code, markdown]
          type: string
      required: [cell_type]
      type: object
    type: array
required: [cells, metadata]
type: object
```

JSON Schema basics are outlined here and a cheat sheet of standard keys can be found here.

3 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.

```
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 : 2.4820288072462011 2.4820288072462011 2.4820288072462011
 angles : 109.47122063449069 109.47122063449069 109.47122063449069
 volume : 11.770598948
      A : -1.4329999999999996 -1.4330000000000001 1.4329999999999998
      B : -1.4330000000000001 1.4330000000000001 -1.4330000000000001
      C : 1.4330000000000001 -1.4330000000000001 -1.4330000000000001
PeriodicSite: Fe (0.0000, 0.0000, 0.0000) [0.0000, 0.0000, 0.0000]
```

Note that the underlying structure of the pymatgen structure is JSON.

```
@class: Structure
@module: pymatgen.core.structure
lattice:
  a: 2.48
  alpha: 109.0
  b: 2.48
  beta: 109.0
  c: 2.48
  gamma: 109.0
  matrix: [[-1.43, -1.43, 1.43], [-1.43,
           1.43, -1.43], [1.43, -1.43,
           -1.43]]
  volume: 11.8
sites:
  - abc: [0.0, 0.0, 0.0]
  label: Fe
  species:
      - element: Fe
      occu: 1
  xyz: [0.0, 0.0, 0.0]
```

4 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 data structure (i.e. independendant of any specific graphics package) of all elements we wish to visualise. The key aspects of our structure is:

1. **elements**: A list of elements in the scene. For now we only have one type but more will be added later.
2. **transforms**: A list of geometric transforms we will apply globally (to all elements) and locally (to individual elements) before visualisation.

We shall create a number of elements and transforms (and associated validation schema) for this demonstration, but any number can be specified.

```
elements:
  - cell_vectors:
      a: [3.67, 0.0, 2.25e-16]
      b: [-2.25e-16, 3.67, 2.25e-16]
      c: [0.0, 0.0, 5.03]
  centre: [1.84, 1.84, 2.52]
  color: #e06633
  coords: [[0.0, 0.0, 0.0], [1.84, 1.84, 2.25e-16]]
  label: Fe
  radius: 1.32
  sname: mackinawite
  transforms: []
  transparency: 1.0
  type: repeat_cell
  - cell_vectors:
      a: [3.67, 0.0, 2.25e-16]
      b: [-2.25e-16, 3.67, 2.25e-16]
      c: [0.0, 0.0, 5.03]
  centre: [1.84, 1.84, 2.52]
  color: #b2b200
  coords: [[-1.12e-16, 1.84, 1.31], [1.84, 0.0, 3.72]]
  label: S
  radius: 1.05
  sname: mackinawite
  transforms: []
  transparency: 1.0
  type: repeat_cell
transforms: []
```

To create the repeat cells, we have deconstructed the pymatgen structure and applied a mapping of atomic number to radius/color, using a pre-constructed csv table (table 4.1).

Table 4.1: 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

We have grouped the atoms with the similar visual representations, rather than specifying each atom separately for simplicity. Here we do this by atomic number, but equally it could be done by symmetry equivalence or another metric.

4.1 Geometric Transforms

Geometric transforms which we likely want to perform include:

- creating a supercell of the configuration
- orientating the configuration in a convenient manner in the cartesian coordinate space
- slicing into the configuration

```
elements:
  - transforms: []
  - transforms: []
transforms:
  - cvector: b
  direction: (1, 0, 0)
  type: local_align
  - cvector: a
  recentre: True
  rep: 2
  type: local_repeat
  - centre: (0.0, 0.0, 0.0)
  type: recentre
  - centre: None
  lbound: None
  normal: (0, 0, 1)
  type: slice
  ubound: 1.0
```

Note we haven't actually performed any of these transforms yet. This is done done by creating a transformed copy of the data at visualisation time.

```
elements:
  - cell_vectors:
      a: [14.7, 0.0, 9e-16]
      b: [-9e-16, 14.7, 9e-16]
      c: [0.0, 0.0, 20.1]
  centre: [0.0, 0.0, 0.0]
  color: #e06633
  coords: [[-7.35, -7.35, -10.1], [-5.51, -5.51, -10.1], [-3.67, -7.35, -
10.1], ...(x61)]
  label: Fe
  radius: 1.32
  sname: mackinawite
  transforms: []
  transparency: 1.0
  type: repeat_cell
  - cell_vectors:
      a: [14.7, 0.0, 9e-16]
      b: [-9e-16, 14.7, 9e-16]
      c: [0.0, 0.0, 20.1]
  centre: [0.0, 0.0, 0.0]
  color: #b2b200
```

```
  coords: [[-7.35, -5.51, -8.76], [-5.51, -7.35, -6.34], [-3.67, -5.51, -
8.76], ...(x29)]
  label: S
  radius: 1.05
  sname: mackinawite
  transforms: []
  transparency: 1.0
  type: repeat_cell
transforms: []
```

5 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 9 how this can be utilised to distribute the analysis.

```
A Jupyter Widget
```

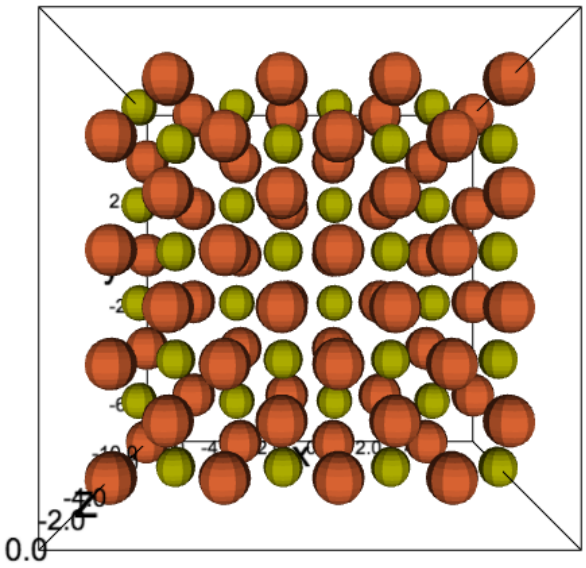

Figure 5.1: an example of an ipyvolume scatter plot

6 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
```

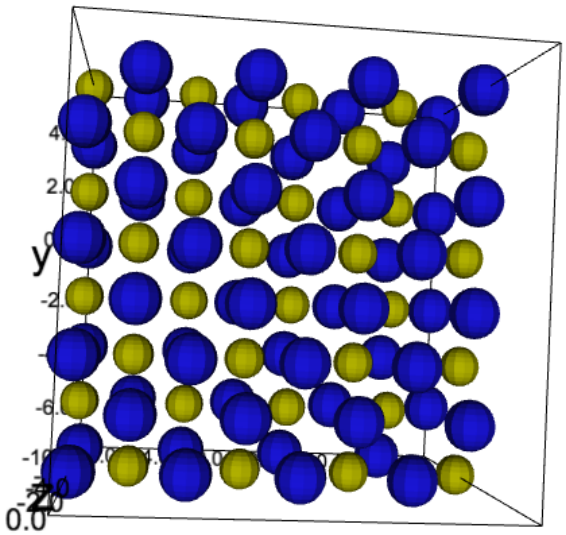

Figure 6.1: an example of an ipyvolume scatter plot (with bespoke controls)

7 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.

7.1 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]
```

7.2 Visualisation Preparation

We then, follow the same process as for atoms; converting to a common structure and adding geometric transforms. Note that we apply the slices locally, so that we can first resize the density array. We do this because, after the `repeat` transforms, the array size is now; (400,400,400), which would be costly to compute the slice for. Resizing by 0.25 reduces the array back to (100,100,100).

```
elements:
  - transforms:
      - type: slice
  type: repeat_cell
  - transforms:
      - sfraction: 0.25
      type: resize
      - type: slice
  type: repeat_density
transforms:
  - rep: 3
  type: local_repeat
  - rep: 3
```

```
  type: local_repeat
  - rep: 3
  type: local_repeat
  - type: recentre
```

7.3 Visualisation

Before parsing to ipyvolume, 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. In the example, we have thus created a really nice representation of the covalent bonding in bulk silicon crystals.

```
A Jupyter Widget
```

```
A Jupyter Widget
```

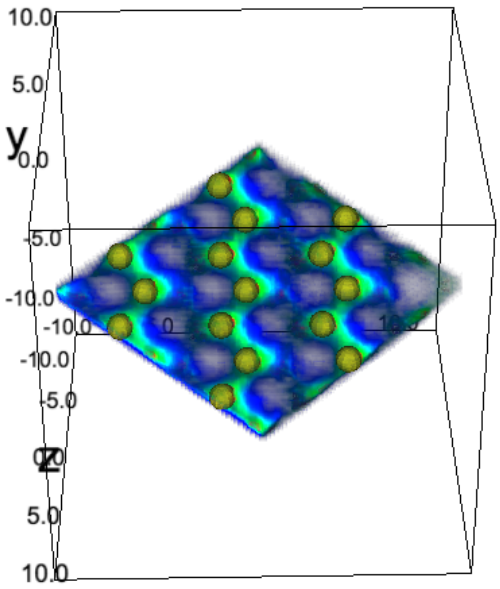

Figure 7.1: an example of an ipyvolume volume and scatter plot (with bespoke controls)

8 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.

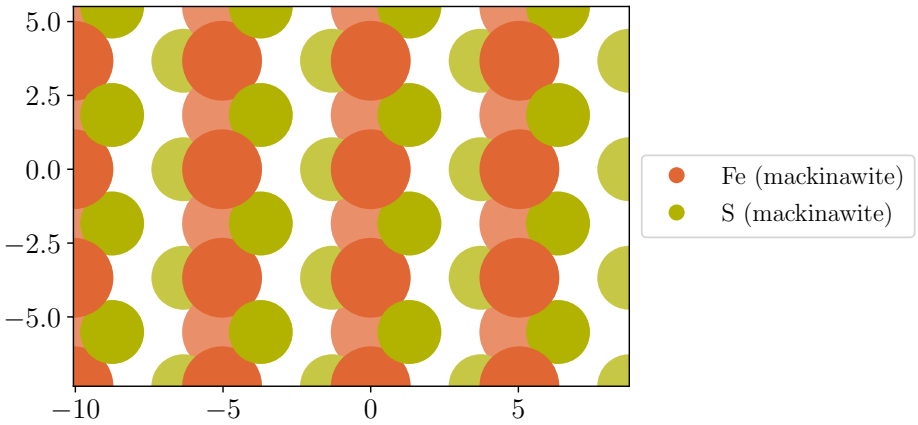

Figure 8.1: an example of a 2D representation of the atomic data

9 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 Reveal.JS 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"
```

10 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. 10.1)

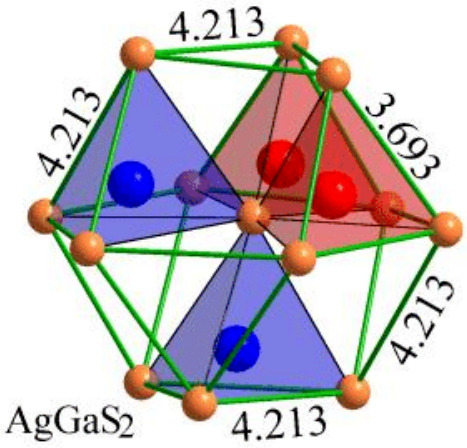

Figure 10.1: an example of nearest-neighbour polygons

- functional (browser side) controls, e.g. slider to translate/rotate point set. Not yet implemented in ipywidgets, see this issue for current status.
- volumes:
  - RuntimeWarning: 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