chemlab Documentation

Release 0.1

Gabriele Lanaro

April 08, 2013

CONTENTS

Webpage https://chemlab.github.com/chemlab

Project Page https://github.com/chemlab/chemlab

Mailing List python-chemlab.googlegroups.com

Downloads https://chemlab.github.com/chemlab

Chemlab is a library that can help the user with chemistry-relevant calculations using the flexibility and power of the python programming language. It aims to be well-designed and pythonic, taking inspiration from project such as numpy and scipy.

Chemlab long term goal is to be:

- **General** Chemistry is a huge field, chemlab wants to provide a general ground from where to build domain-specific tools and apps.
- **Array oriented** most operations and data structures are based on numpy arrays. This let you write compact and efficient code.
- **Graphic** chemlab integrates a 3D molecular viewer that is easily extendable and lets you write your own visualization tools.
- **Interoperable** chemlab wants to be interoperable with other chemistry programs by reading and writing different file formats and using flexible data structures.
- **Fast** Even if python is known to be slow every effort should be made to make chemlab 'fast enough', by using effectively numpy arrays and efficient data structures. When everything else fails we can still write the hard bits in C with the help of cython.

CHAPTER

ONE

CURRENT STATUS

Computational and theoretical chemistry is a huge field, and providing a program that encompasses all aspect of it is an impossible task. The spirit of chemlab is to provide a common ground from where you can build specific programs. For this reason it includes an *fully programmable* molecular viewer.

Chemlab is in its early developement and it provides the most basic data structures. The molecular viewer has a solid ground and can actually draw and play trajectories in an efficient way. To get started be sure to check the *User Manual*.

Chemlab is developer-friendly, it provides good documentation and has an easy structure to get in. Feel free to send me anything that you may do with chemlab, like supporting a new file format, a new graphic renderer, a nice example, even if you don'think it's perfect. Send an email to the mailing list or file an issue on the github page to discuss any idea that comes to your mind. Get involved!

CHAPTER

TWO

USER MANUAL

Table of Contents

2.1 Installation and Quickstart

chemlab is currently tested on Ubuntu 12.10 and python 2.7. First install the dependencies:

```
$ sudo apt-get install python-numpy python-scipy python-matplotlib python-pyside python-opengl cythor
```

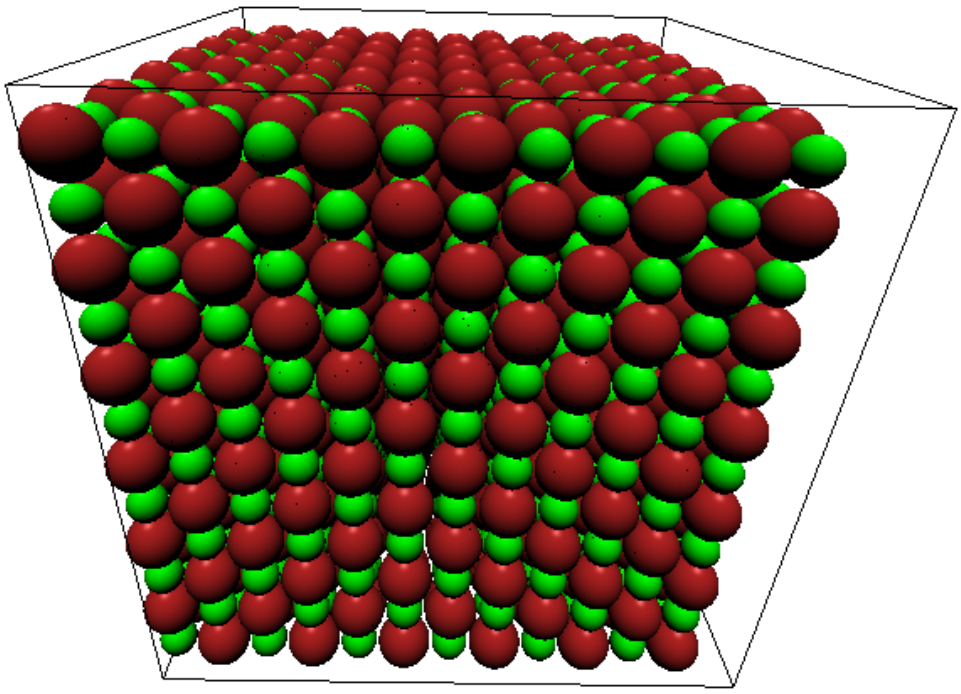
Download unpack and install chemalb from the setup.py included in the package:

```
$ wget https://pypi.python.org/packages/source/c/chemlab/chemlab-0.1.tar.gz
$ tar xvzf chemlab-0.1.tar.gz
$ cd chemlab-0.1
$ sudo python setup.py install
```

Test the newly installed package by typing:

```
$ chemlab view tests/data/cry.gro
```

The molecular viewer should display a crystal, if not, file an issue on github.



Once you're setup you're ready to to dig in chemlab's features contained in the *User Manual*.

2.1.1 Developement

After installing the dependencies, grab the chemlab source from git:

```
$ git clone --recursive https://github.com/chemlab/chemlab.git
```

Complile the included extensions:

```
$ python setup.py build_ext --inplace
```

Just add the chemlab directory to the PYTHONPATH in your .bashrc:

```
export PYTHONPATH=$PYTHONPATH:/path/to/chemlab
```

2.2 Atoms, Molecules and Systems

In chemlab, atoms can be represented using the `chemlab.core.Atom` data structure that contains some common information about our particles like type, mass and position. Atom instances are easily created by initializing them with data

```
>>> from chemlab.core import Atom
>>> ar = Atom('Ar', [0.0, 0.0, 0.0])
>>> ar.type
'Ar'
>>> ar.r
np.array([0.0, 0.0, 0.0])
```

Note: for the atomic coordinates you should use nanometers

A `chemlab.core.Molecule` is an entity composed of more atoms and most of the Molecule properties are inherited from the constituent atoms. To initialize a Molecule you can, for example pass a list of atom instances to its constructor:

```
>>> from chemlab.core import Molecule
>>> mol = Molecule([at1, at2, at3])
```

2.2.1 Manipulating Molecules

Molecules are easily and efficiently manipulated through the use of numpy arrays. One of the most useful arrays contained in Molecule is the array of coordinates `Molecule.r_array`. The array of coordinates is a numpy array of shape `(NA,3)` where `NA` is the number of atoms in the molecule. According to the numpy broadcasting rules, if you sum two arrays with shapes `(NA,3)` and `(3,)`, each row of the first array get summed by the second array. Let's say we have a water molecule and we want to displace it randomly in a box, this is easily accomplished by initializing a Molecule at the origin and summing its coordinates by a random displacement:

```
import numpy as np

wat = Molecule([Atom("H", [0.0, 0.0, 0.0]),
                Atom("H", [0.0, 1.0, 0.0]),
                Atom("O", [0.0, 0.0, 1.0])])

# Shapes (NA, 3) and (3,)
wat.r_array += np.random.rand(3)
```

Using the same principles you can also apply other kinds of transformations such as matrices. You can for example rotate the molecule by 90 degrees around the z-axis:

```
from chemlab.graphics.transformations import rotation_matrix

# The transformation module returns 4x4 matrices
M = rotation_matrix(np.pi/2, np.array([0.0, 0.0, 1.0]))[:3,:3]

# slow, readable way
for i,r in enumerate(wat.r_array):
    wat.r_array[i] = np.dot(M,r)

# numpy efficient way to do the same:
# wat.r_array = np.dot(wat.r_array, M.T)
```

The array-based API provides a massive increase in performance and a more straightforward integration with C libraries thanks to the numpy arrays. This feature comes at a cost: the data is copied between atoms and molecules, in other words the changes in the costituents atoms are not reflected in the Molecule and viceversa. Even if it may look a bit unnatural, this approach limits side effects making the code more predictable and easy to follow.

2.2.2 Systems

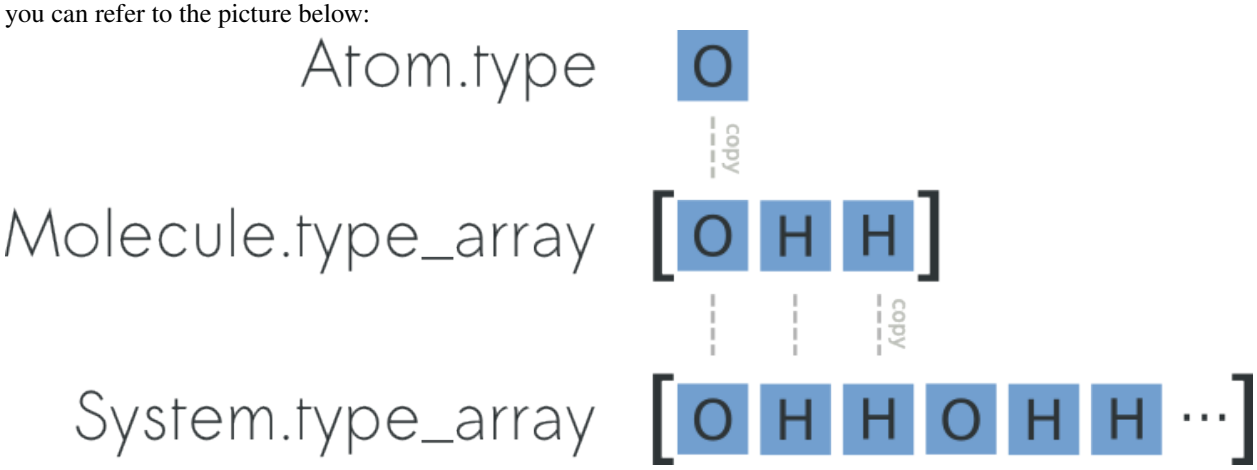
In context such as molecular simulations it is customary to introduce a new data structure called `System`. A *System* represents a collection of molecules, and optionally (but recommended) you can pass also periodic box information:

```
>>> from chemlab.core import System
# molecule = a list of Molecule instances
>>> s = System(molecules, boxsize=2.0)
```

System do not take directly *Atom* instances as its constituents, therefore if you need to simulate a system made of single atoms (say, a box of liquid Ar) you need to wrap the atoms into a Molecule:

```
>>> ar = Atom('Ar', [0.0, 0.0, 0.0])
>>> mol = Molecule([ar])
```

System, similarly to Molecule can expose data by using arrays and it inherits atomic data from the constituent molecules. For instance, you can easily and efficiently access all the atomic coordinates by using the attribute `System.r_array`. To understand the relation between `Atom.r`, `Molecule.r_array` and `System.r_array` you can refer to the picture below:



You can preallocate a *System* by using the classmethod `System.empty` (pretty much like you can preallocate numpy arrays with *np.empty* or *np.zeros*) and then add the molecules one by one:

```
import numpy as np
from chemlab.core import Atom, Molecule, System
from chemlab.graphics import display_system

# Template molecule
wat = Molecule([Atom('O', [0.00, 0.00, 0.01]),
                Atom('H', [0.00, 0.08,-0.05]),
                Atom('H', [0.00,-0.08,-0.05])])

# Initialize a system with four water molecules.
s = System.empty(4, 12) # 4 molecules, 12 atoms

for i in range(4):
    wat.move_to(np.random.rand(3)) # randomly displace the water molecule
    s.add(wat) # data gets copied each time
```

```
display_system(s)
```

Since the data is copied, the `wat` molecule act as a *template* so you can move it around and keep adding it to the *System*.

Preallocating and adding molecules is a pretty fast way to build a *System*, but the fastest way (in terms of processing time) is to build the system by passing ready-made arrays, this is done by using `chemlab.core.System.from_arrays()`.

Building Crystals

chemlab provides an handy way to build crystal structures from the atomic coordinates and the space group information. If you have the crystallographic data, you can easily build a crystal:

```
from chemlab.core import Atom, Molecule, crystal
from chemlab.graphics import display_system

# Molecule templates
na = Molecule([Atom('Na', [0.0, 0.0, 0.0])])
cl = Molecule([Atom('Cl', [0.0, 0.0, 0.0])])

s = crystal([[0.0, 0.0, 0.0], [0.5, 0.5, 0.5]], # Fractional Positions
            [na, cl], # Molecules
            225, # Space Group
            cellpar = [.54, .54, .54, 90, 90, 90], # unit cell parameters
            repetitions = [5, 5, 5]) # unit cell repetitions in each direction

display_system(s)
```

See Also:

`chemlab.core.crystal()`

Note: If you'd like to implement a .cif file reader, you're welcome! Drop a patch on github.

Manipulating Systems

Selections

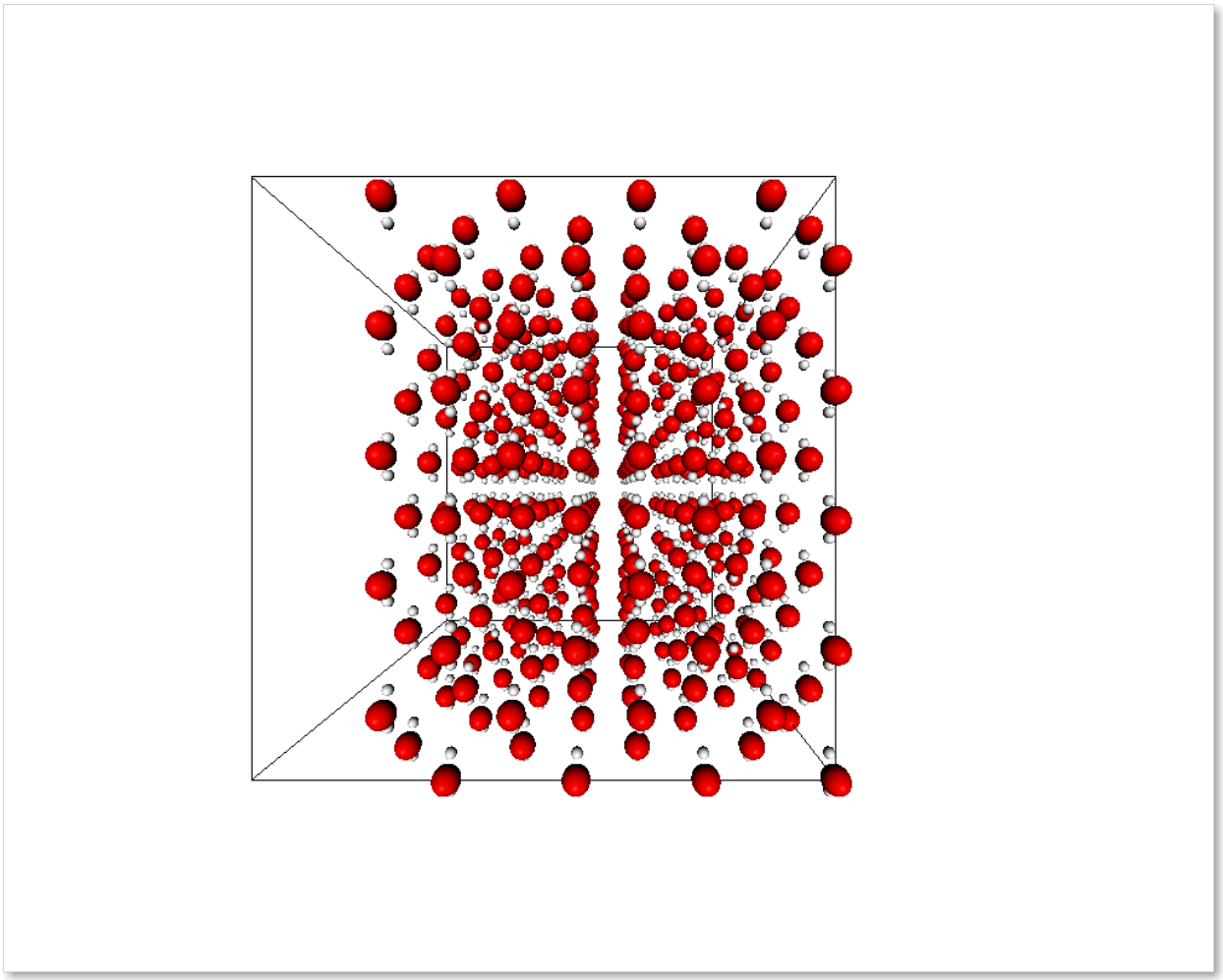
You can manipulate systems by using some simple but flexible functions. It is really easy to generate a system by selecting a part from a bigger system, this is implemented in the functions `chemlab.core.subsystem_from_atoms()` and `chemlab.core.subsystem_from_molecules()`.

Those two functions take as first argument the original *System*, and as the second argument a *selection*. A *selection* is either a boolean array that is True when we want to select that element and False otherwise or an integer array containing the elements that we want to select. By using those two functions we can create subsystem by building those selections.

The following example shows an easy way to take the molecules that contain atoms in the region of space $x > 0.5$ by employing `subsystem_from_atoms()`:

```
import numpy as np
from chemlab.core import crystal, Molecule, Atom, subsystem_from_atoms
from chemlab.graphics import display_system
```

```
# Template molecule
wat = Molecule([Atom('O', [0.00, 0.00, 0.01]),
                Atom('H', [0.00, 0.08,-0.05]),
                Atom('H', [0.00,-0.08,-0.05])])

s = crystal([[0.0, 0.0, 0.0]], [wat], 225,
     cellpar = [.54, .54, .54, 90, 90, 90], # unit cell parameters
     repetitions = [5, 5, 5]) # unit cell repetitions in each direction

selection = s.r_array[:, 0] > 0.5
sub_s = subsystem_from_atoms(s, selection)

display_system(sub_s)
```



It is also possible to select a subsystem by selecting specific molecules, in the following example we select the first 10 water molecules by using `subsystem_from_molecules()`:

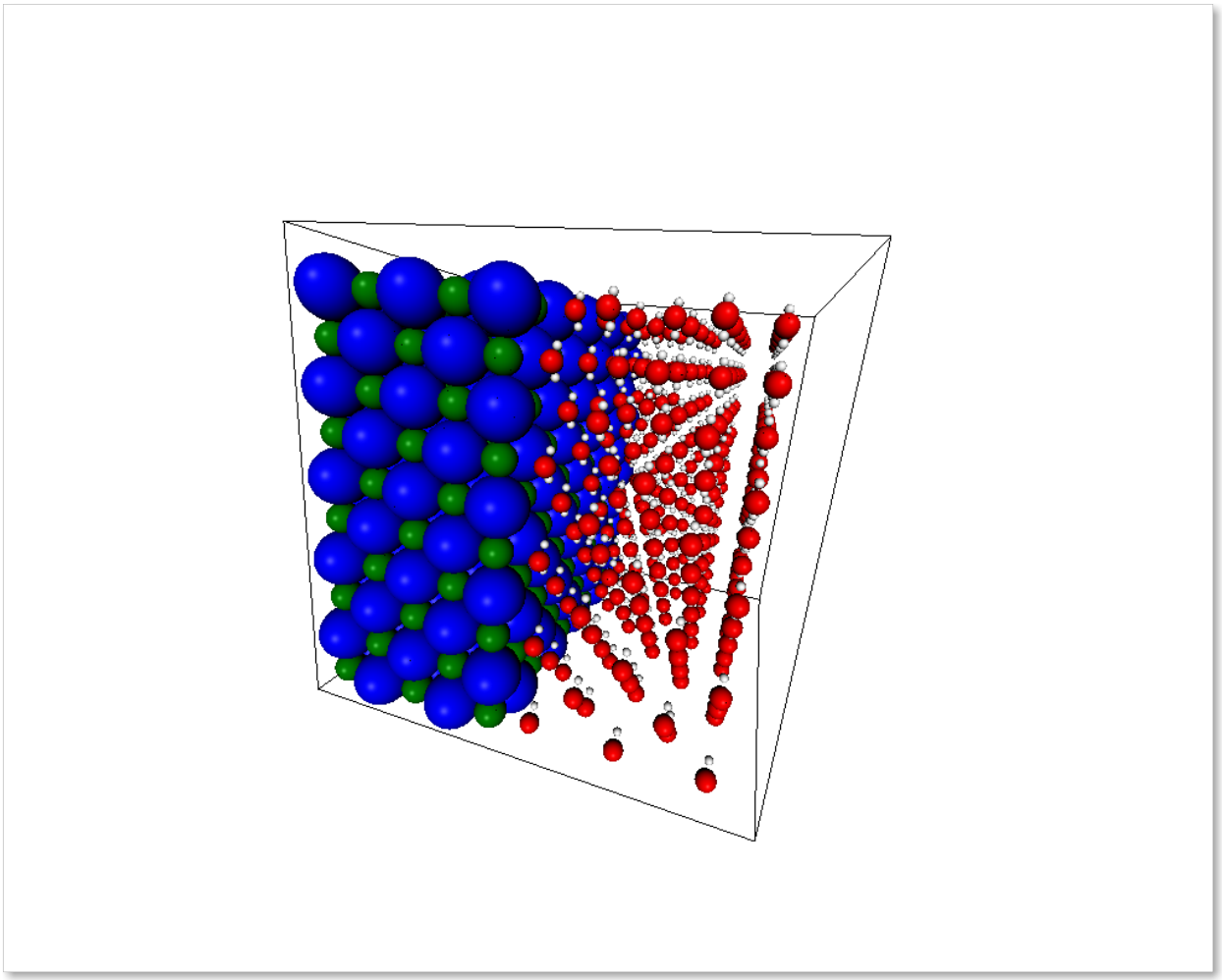
```
from chemlab.core import subsystem_from_molecules

selection = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
sub_s = subsystem_from_molecules(s, selection)
```

Note: chemlab will provide other selection utilities in the future, if you have a specific request, file an issue on github

Merging systems

You can also create a system by merging two different systems. In the following example we will see how to make a NaCl/H2O interface by using `chemlab.core.merge_systems()`:

```
import numpy as np
from chemlab.core import Atom, Molecule, crystal
from chemlab.core import subsystem_from_atoms, merge_systems
from chemlab.graphics import display_system

# Make water crystal
wat = Molecule([Atom('O', [0.00, 0.00, 0.01]),
      Atom('H', [0.00, 0.08,-0.05]),
      Atom('H', [0.00,-0.08,-0.05])])

water_crystal = crystal([[0.0, 0.0, 0.0]], [wat], 225,
     cellpar = [.54, .54, .54, 90, 90, 90], # unit cell parameters
     repetitions = [5, 5, 5]) # unit cell repetitions in each direction

# Make nacl crystal
na = Molecule([Atom('Na', [0.0, 0.0, 0.0])])
cl = Molecule([Atom('Cl', [0.0, 0.0, 0.0])])

nacl_crystal = crystal([[0.0, 0.0, 0.0], [0.5, 0.5, 0.5]], [na, cl], 225,
      cellpar = [.54, .54, .54, 90, 90, 90],
      repetitions = [5, 5, 5])

water_half = subsystem_from_atoms(water_crystal,
                water_crystal.r_array[:,0] > 1.2)
nacl_half = subsystem_from_atoms(nacl_crystal,
                nacl_crystal.r_array[:,0] < 1.2)

interface = merge_systems(water_half, nacl_half)
display_system(interface)
```



At the present time, the merging will avoid overlapping by creating a bounding box around the two systems and removing the molecules of the first system that are inside the second system bounding box. In the future there will be more clever ways to handle this overlaps.

Sorting

If you use chemlab in conjunction with GROMACS, you may use the `chemlab.core.System.sort()` to sort the molecules according to their molecular formulas before exporting. The topology file expect to have a file with the same molecule type ordererd.

2.3 Input and Output Routines

2.3.1 The jungle of file formats

There are *a lot* of file formats used and produced by chemistry applications. Each program has his way to store geometries, trajectories, energies and properties etc. chemlab tries to encompass all of those different properties by using a lightweight way to handle such differences.

2.3.2 Reading and writing data

The classes responsible for the I/O are subclasses of `chemlab.io.handlers.IOHandler`. These handlers work all in the same way, here is an example of GroHandler:

```
from chemlab.io import GroIO

infile = GroIO('waterbox.gro')
system = infile.read('system')

# Modify system as you wish...

outfile = GroIO('waterbox_out.gro')
outfile.write('system', system)
```

You first create the handler instance for a certain format and then you can read a certain *feature* provided by the handler. In this example we read and write the *system* feature.

Some file formats may have some extra data for each atom, molecule or system. For example the ".gro" file formats have his own way to call the atoms in a water molecule: `OW`, `HW1`, `HW2`. To handle such issues, you can write this information in the *export* arrays contained in the data structures, such as `Atom.export`, `Molecule.export`, and their array-based counterparts `Molecule.atom_export_array`, `System.mol_export` and `System.atom_export_array`.

Those attributes are especially important where you write in some data format, since you may have to provide those attribute when you initialize your Atom, Molecule and System.

You can easily open a data file without even having to search his format handler by using the utility function `chemlab.io.datafile()`:

```
from chemlab.io import datafile

sys = datafile('waterbox.gro').read('system')
t, coords = datafile('traj.xtc').read('trajectory')
```

See Also:

Supported File Formats

2.3.3 Implementing your own IOHandler

Implementing or improving an existing IOHandler is a great way to partecipate in chemlab development. Fortuately, it's extremely easy to setup one of them.

It boils down to a few steps:

1. Subclass `IOHandler`;
2. Define the class attributes *can_read* and *can_write*;
3. Implement the *write* and *read* methods for the features that you added in *can_read* and *can_write*;
4. Write the documentation for each feature.

Here is an example of the *xyz* handler:

```
import numpy as np
from chemlab.io.handlers import IOHandler
from chemlab.core import Molecule

class XyzIO(IOHandler):
```

```
    '''The XYZ format is described in this wikipedia article
    http://en.wikipedia.org/wiki/XYZ_file_format.

    **Features**

    .. method:: read("molecule")

       Read the coordinates in a :py:class:`~chemlab.core.Molecule` instance.

    .. method:: write("molecule", mol)

       Writes a :py:class:`~chemlab.core.Molecule` instance in the XYZ format.
    '''

    can_read = ['molecule']
    can_write = ['molecule']

    def __init__(self, filename):
        self.filename = filename

    def read(self, feature):
        self.check_feature(feature, "read")
        lines = open(self.filename).readlines()

        num = int(lines[0])
        title = lines[1]

        if feature == 'title':
            return title

        if feature == 'molecule':
            type_array = []
            r_array = []
            for l in lines[2:]:
                type, x, y, z = l.split()
                r_array.append([float(x),float(y),float(z)])
                type_array.append(type)

            r_array = np.array(r_array)/10 # To nm
            type_array = np.array(type_array)

            return Molecule.from_arrays(r_array=r_array, type_array=type_array)


    def write(self, feature, mol):
        self.check_feature(feature, "write")
        lines = []
        if feature == 'molecule':
            lines.append(str(mol.n_atoms))

            lines.append('Generated by chemlab')
            for t, (x, y, z) in zip(mol.type_array, mol.r_array):
                lines.append('    %s       %.6f      %.6f      %.6f' %
                             (t, x*10, y*10, z*10))

            open(self.filename, 'w').write('\n'.join(lines))
```

A few remarks:

- **It is recommended to use the method** `check_feature()` before performing read/write. This will check that the feature is present in the can_read/can_write list;
- **If you want to squeeze out performance you should use** `Molecule.from_arrays()` and `System.from_arrays()`;
- **You can read whatever data you wish, for example the** `EdrIO` handler does not read Molecule or System at all;
- You can definitely take inspiration from the handlers included in chemlab, *Supported File Formats*.

2.4 Graphics and Visualization

2.4.1 Intro

The `chemlab.graphics` package is one of the most interesting aspects of chemlab, that sets him apart from similar programs.

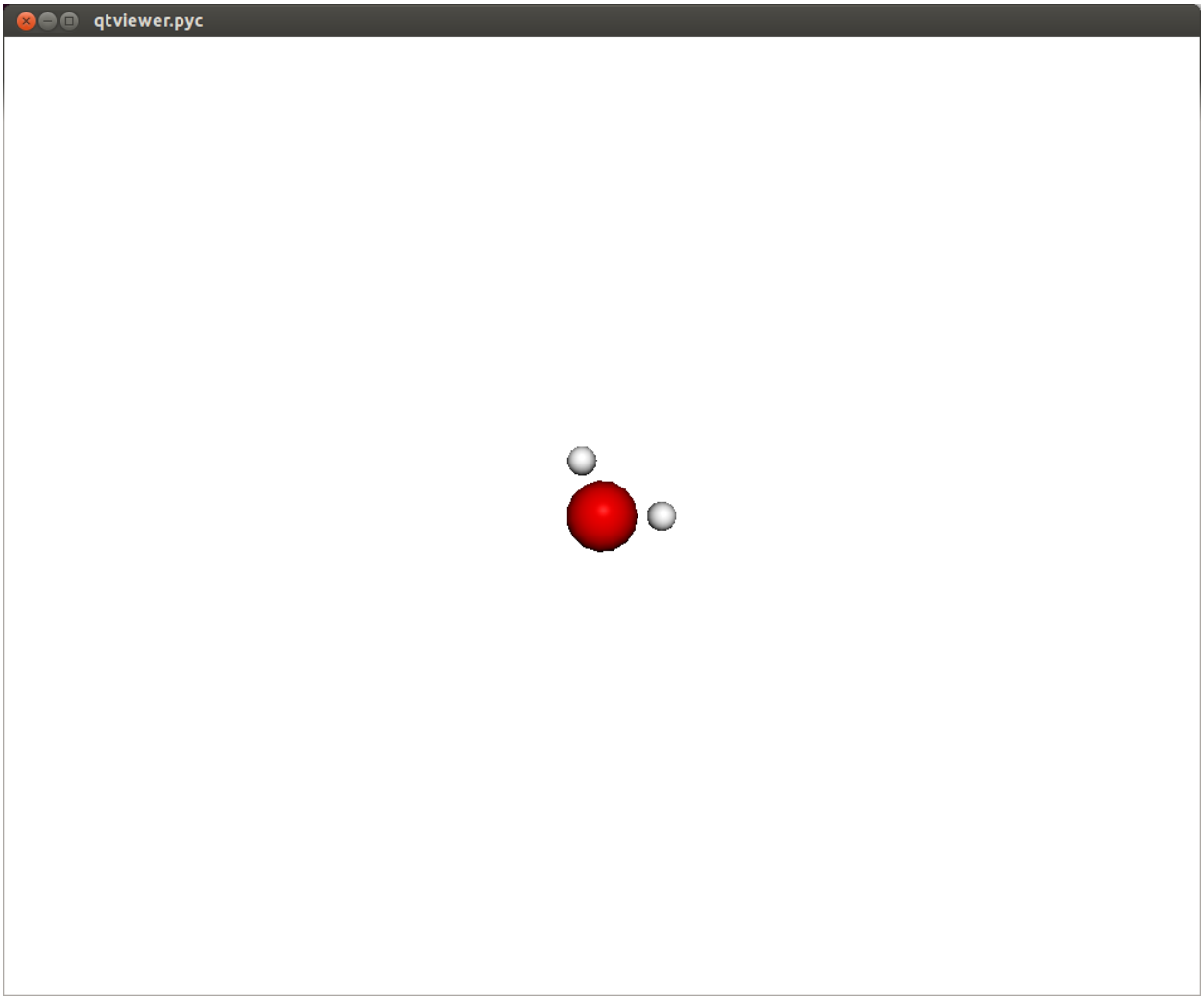
The purpose of the package is to provide a solid library to develop 3D applications to display chemical data in an flexible way. For example it's extremely easy to build a molecular viewer and add a bunch of custom features to it.

The typical approach when developing a graphics application is to create a `QtViewer` instance and add 3D features to it:

```
>>> from chemlab.graphics import QtViewer
>>> v = QtViewer()
```

now let's define a molecule. We can use the *moldb* module to get a water template.

```
>>> from chemlab.graphics.renderers import SphereRenderer
>>> from chemlab.data.moldb import water
>>> ar = v.add_renderer(AtomRenderer, water.r_array, water.type_array)
>>> v.run()
```



In this way you should be able to visualize a molecule where each atom is represented as a sphere. There are also a set of viewing controls:

- **Mouse Drag (Left Click) or Left/Right/Up/Down**: Rotate the molecule
- **Mouse Drag (Right Click)**: Pan the view
- **Mouse Wheel or +/-**: Zoom in/out

In a similar fashion it is possible to display other features, such as boxes, arrows, lines, etc. It is useful to notice that with `Viewer.add_renderer` we are not passing an *instance* of the renderer, but we're passing the renderer *class* and its respective constructor arguments. The method `Viewer.add_renderer` returns the actual instance.

It is possible as well to overlay 2D elements to a scene in a similar fashion, this will display a string at the screen position 300, 300:

```
from chemlab.graphics.uis import TextUI
tui = v.add_ui(TextUI, 300, 300, "Hello, World!")
```

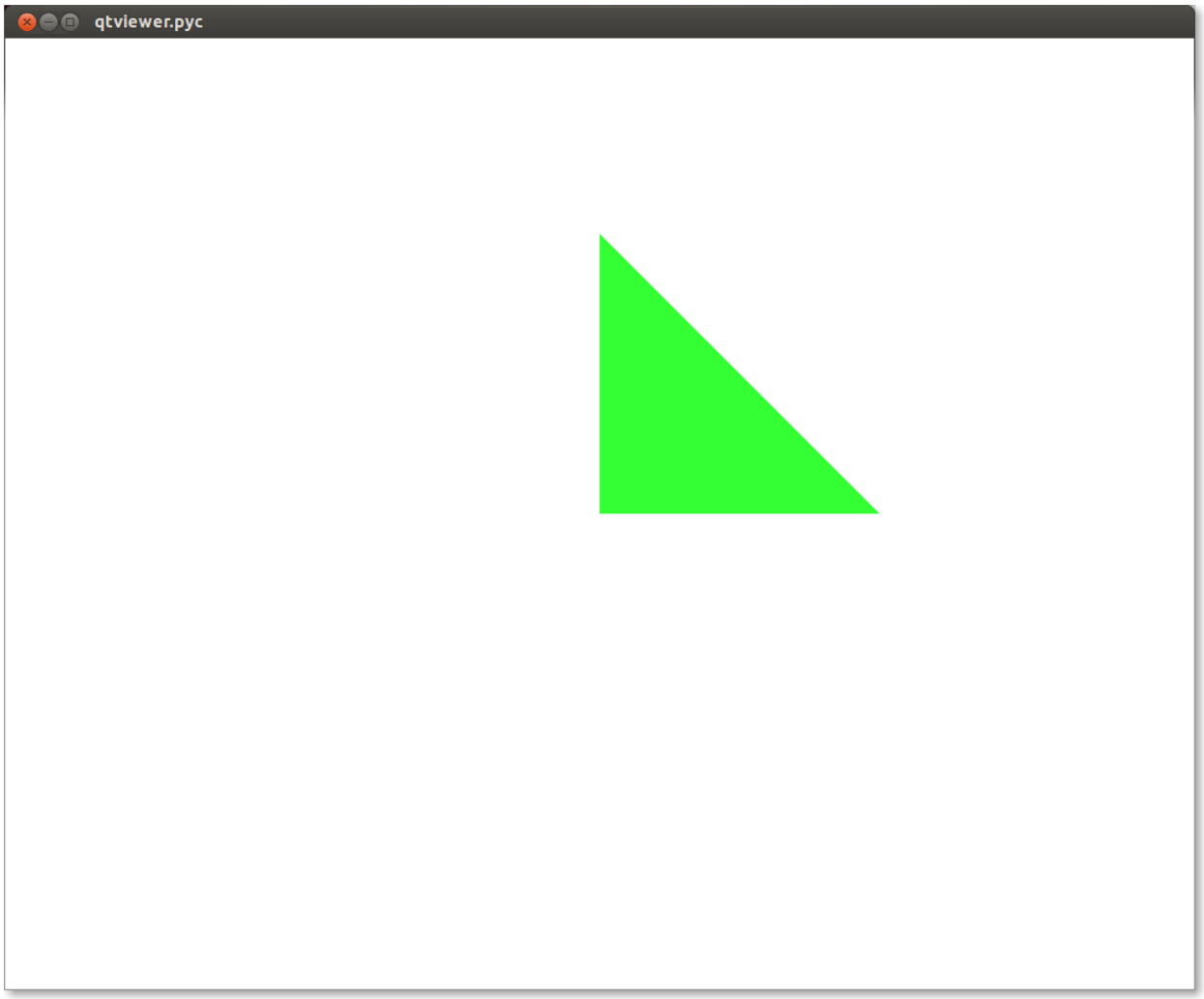
Anyway, I encourage you to use the powerful Qt framework to provide interaction and widgets to your application.

2.4.2 Renderers

Renderers are simply classes used to draw 3D objects. They are tecnically required to provide just one method, *draw* and they must take an instance of `QChemlabWidget` as their first argument (check out the `AbstractRenderer` class). In this way they provide the maximum flexibility required to build efficient opengl routines. Renderers may be subclass other renderers as well as use other renderers.

A very useful renderer is TriangleRenderer, used to render efficiently a list of triangles, it constitutes a base for writing other renderers. TriangleRenderer works basically like this, you pass the vertices, normals and colors of the triangle and it will display a triangle in the world:

```
from chemlab.graphics import QtViewer
from chemlab.graphics.renderers import TriangleRenderer
from chemlab.graphics.colors import green
import numpy as np

vertices = np.array([[0.0, 0.0, 0.0], [0.0, 1.0, 0.0], [1.0, 0.0, 0.0]])
normals = np.array([[0.0, 0.0, 1.0], [0.0, 0.0, 1.0], [0.0, 0.0, 1.0]])
colors = np.array([green, green, green])

v = QtViewer()
v.add_renderer(TriangleRenderer, vertices, normals, colors)
v.run()
```



If you pass 6 vertices/normals/colors, he will display 2 triangles and so on. As a sidenote, he is very efficient and in fact `chemlab.graphics.renderers.TriangleRenderer` is used as a backend for a lot of other renderers such as `SphereRenderer` and `CylinderRenderer`. If you can reduce a shape in triangles, you can easily write a renderer for it.

In addition to that, `TriangleRenderer` provides also a method to update vertices, normals and colors. We can demonstrate that from the last example by defining an update function that rotates our triangle:

```
from chemlab.graphics.transformations import rotation_matrix

def update():
    y_axis = np.array([0.0, 1.0, 0.0])

    # We take the [:3,:3] part because rotation_matrix can be used to
    # rotate homogeneous (4D) coordinates.
    rot = rotation_matrix(3.14/32, y_axis)[:3, :3]

    # This is the numpy-efficient way of applying rot to each coordinate
    vertices[:] = np.dot(vertices, rot.T)
    normals[:] = np.dot(vertices, rot.T)

    tr.update_vertices(vertices)
    tr.update_normals(normals)
    v.widget.repaint()

v.schedule(update, 10)
v.run()
```

On this ground we can develop a TetrahedronRenderer based on our TriangleRenderer. To do that we first need to understand how a tetrahedron is made, and how can we define the vertices that make the tetrahedron.

2.4.3 Tutorial: TetrahedronRenderer

First of all, we need to have the 4 coordinates that represents a tetrahedron. Without even trying to visualize it, just pick the values straight from Wikipedia:

```
import numpy as np
v1 = np.array([1.0, 0.0, -1.0/np.sqrt(2)])
v2 = np.array([-1.0, 0.0, -1.0/np.sqrt(2)])
v3 = np.array([0.0, 1.0, 1.0/np.sqrt(2)])
v4 = np.array([0.0, -1.0, 1.0/np.sqrt(2)])
```

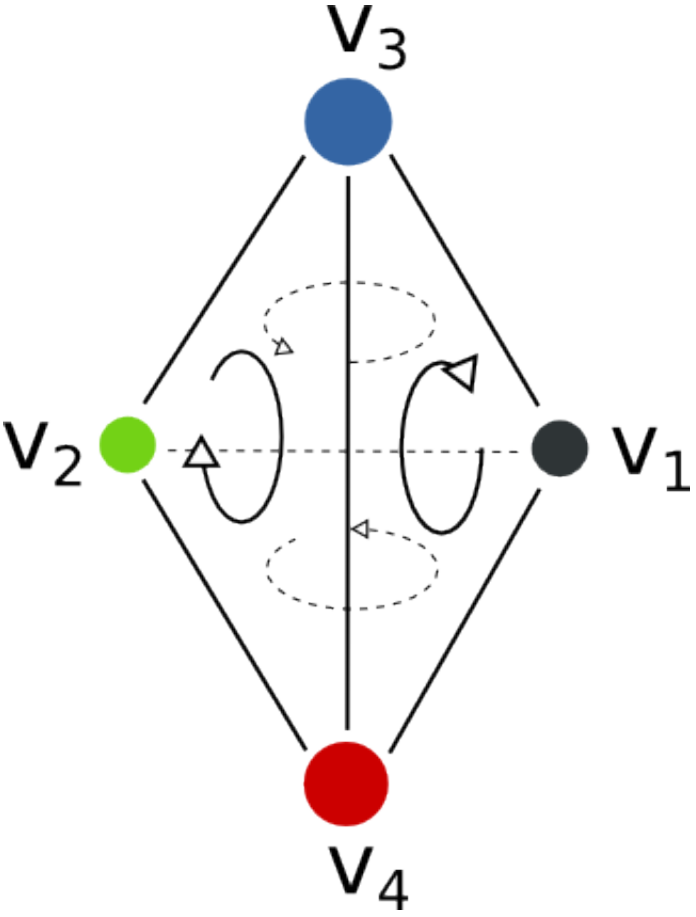
We can quickly verify if this is correcty by using a `PointRenderer`:

```
from chemlab.graphics import QtViewer
from chemlab.graphics.renderers import PointRenderer
from chemlab.graphics.colors import black, green, blue, red

colors = [black, green, blue, red]
v = QtViewer()
v.add_renderer(PointRenderer, np.array([v1, v2, v3, v4]), colors)
v.run()
```

We've got 4 boring points that look like they're at the vertices of a tetrahedron. Most importantly we learned that we can use PointRenderer to quickly test shapes.

Now let's define the four triangles (12 vertices) that represent a solid tetrahedron. It is good practice to put the triangle vertices in a certain order to estabilish which face is pointing outside and which one is pointing inside for optimization

reasons. The convention is that if we specify 3 triangle vertices in clockwise order this means that the face points outwards from the solid:



We can therefore write our vertices and colors:

```
vertices = np.array([
    v1, v4, v3,
    v3, v4, v2,
    v1, v3, v2,
    v2, v4, v1
])

colors = [green] * 12
```

All is left to do is write the normals to the surface at each vertex. This is easily done by calculating the cross product of the vectors constituting two sides of a triangle, (remember that the normals should point outward):

```
n1 = -np.cross(v4 - v1, v3 - v1)
n2 = -np.cross(v4 - v3, v2 - v3)
n3 = -np.cross(v3 - v1, v2 - v1)
n4 = -np.cross(v4 - v2, v1 - v2)

normals = [n1, n1, n1,
           n2, n2, n2,
           n3, n3, n3,
           n4, n4, n4]
```

```
from chemlab.graphics.renderers import TriangleRenderer
```

```
v.add_renderer(TriangleRenderer, vertices, normals, colors)
v.run()
```

Now that we've got the basic shape in place we can code the actual Renderer class to be used directly with the viewer. We will make a renderer that, given a set of coordinates will display many tetrahedra.

We can start by defining a Renderer class, inheriting from AbstractRenderer, the main thing you should notice is that you need an additional argument *widget* that will be passed when you use the method *QtViewer.add_renderer*:

```
from chemlab.graphics.renderers import AbstractRenderer

class TetrahedraRenderer(AbstractRenderer):
    def __init__(self, widget, positions):
        super(TetrahedraRenderer, self).__init__(widget)
        ...
```

The strategy to implement a multiple-tetrahedron renderer will be like this:

- store the triangle vertices, and normals of a single tetrahedra.
- for each position that we pass, translate the vertices of the single tetrahedra and accumulate the obtained vertices in a big array.
- repeat the normals of a single tetrahedra for the number of tetrahedra we're going to render.
- generate the per-vertex colors (green for simplicity)
- create a TriangleRenderer as an attribute and initialize him with the accumulated vertices, normals, and colors
- reimplement the *draw* method by calling the draw method of our trianglerenderer.

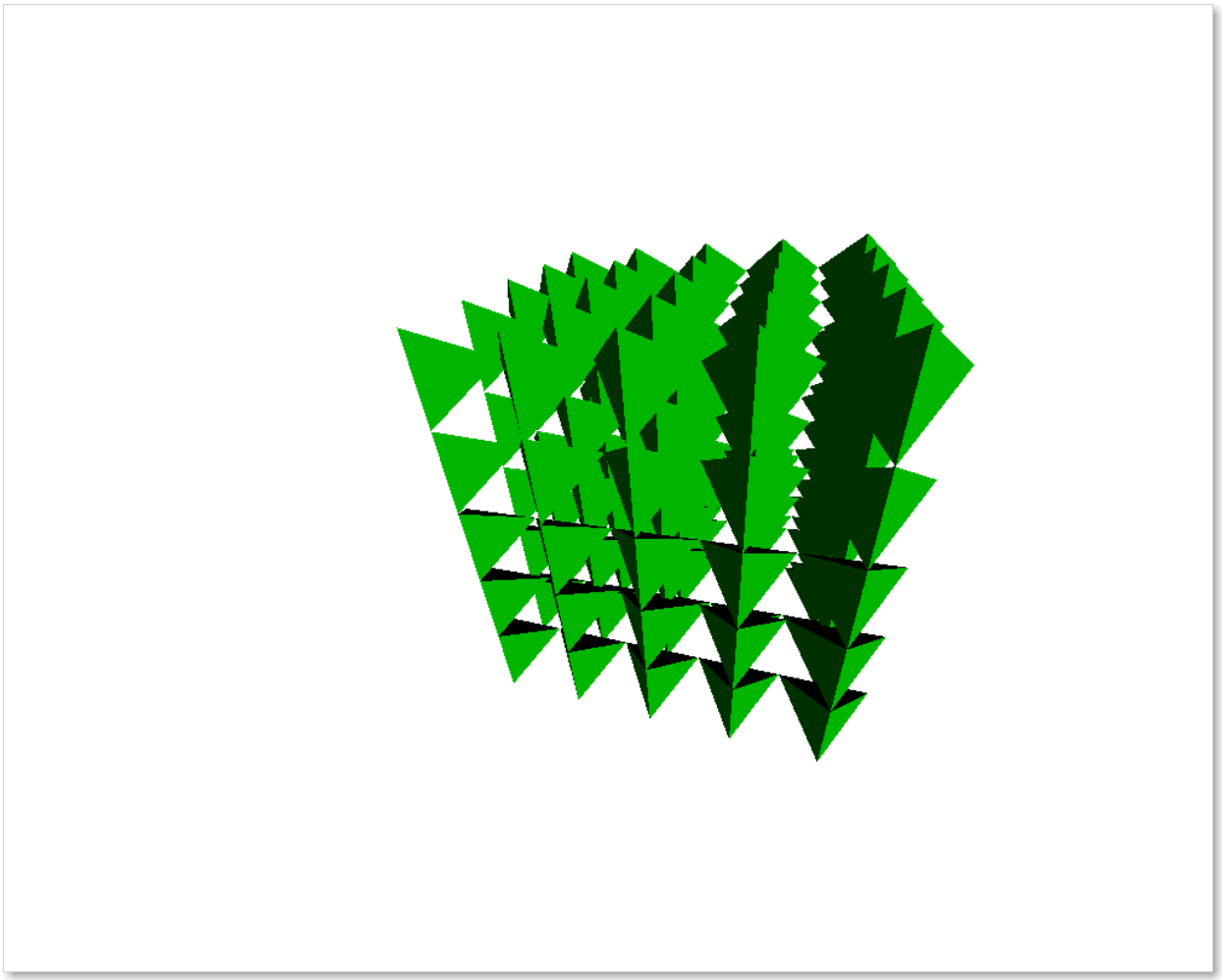
You can see the code in this snippet:

```
class TetrahedraRenderer(AbstractRenderer):
    def __init__(self, widget, positions):
        super(TetrahedraRenderer, self).__init__(widget)

        v1 = np.array([1.0, 0.0, -1.0/np.sqrt(2)])
        v2 = np.array([-1.0, 0.0, -1.0/np.sqrt(2)])
        v3 = np.array([0.0, 1.0, 1.0/np.sqrt(2)])
        v4 = np.array([0.0, -1.0, 1.0/np.sqrt(2)])

        positions = np.array(positions)

        # Vertices of a single tetrahedra
        self._th_vertices = np.array([
            v1, v4, v3,
            v3, v4, v2,
            v1, v3, v2,
            v2, v4, v1
        ])

        self._th_normals = np.array([
            n1, n1, n1,
            n2, n2, n2,
            n3, n3, n3,
            n4, n4, n4])

        self.n_tetra = len(positions)

        tot_vertices = []
```

```
        for pos in positions:
            tot_vertices.extend(self._th_vertices + pos)

        # Refer to numpy.tile, this simply repeats the elements
        # of the array in an efficient manner.
        tot_normals = np.tile(self._th_normals, (self.n_tetra, 1))
        tot_colors = [green] * self.n_tetra * 12

        # !NOTICE! that we have to pass widget as the first argument
        self.tr = TriangleRenderer(widget, tot_vertices,
                                   tot_normals, tot_colors)

    def draw(self):
        self.tr.draw()
```

To demostrate let's draw a grid of 125 tetrahedra:

```
positions = []

for x in range(5):
    for y in range(5):
        for z in range(5):
            positions.append([float(x)*2, float(y)*2, float(z)*2])

v.add_renderer(TetrahedraRenderer, positions)
v.widget.camera.position = np.array([0.0, 0.0, 20.0])
v.run()
```



If you had any problem with the tutorial or you want to implement other kind of renderers don't exitate to contact me. The full code of this tutorial is in *chemlab/examples/tetrahedra_tutorial.py*.

2.5 Using GROMACS with chemlab

GROMACS is one of the most used packages for molecular simulations, chemlab can provide a modern and intuitive interface to generate input and analyze the output of GROMACS calculations. To illustrate the concepts we'll perform a very simple simulation of liquid water.

2.5.1 Installing GROMACS

This depends on the system you're using but I believe that GROMACS is already packaged for most linux distributions and also for other operating systems.

In Ubuntu:

```
$ sudo apt-get install gromacs
```

2.5.2 What GROMACS needs

In order to run a minimum simulation GROMACS requires to know some basic properties of the system we intend to simulate. This boils down to basically 3 ingredients:

1. The starting composition and configuration of our system. This is provided by a ".gro" file that contains the atom and molecule types, and their position in space.
2. Information about the connectivity and interactions between our particles. This is called topology file and it is provided by writing a ".top" file.
3. Simulation method. This will require us to give parameters on how we want to make the system evolve. This is provided by an ".mdp" file.

chemlab can help us to build any system that we want and we'll use it to write a ".gro" file. Then we will use chemlab to visualize and analyze the result of the GROMACS simulation.

2.5.3 Crafting a box of water

There are many ways to generate a box of water, in our example we will place 512 water molecules in a cubic grid. The advantages of doing that is the simplicity of the approach and the fact that we are naturally avoid any overlap between adiacent molecules.

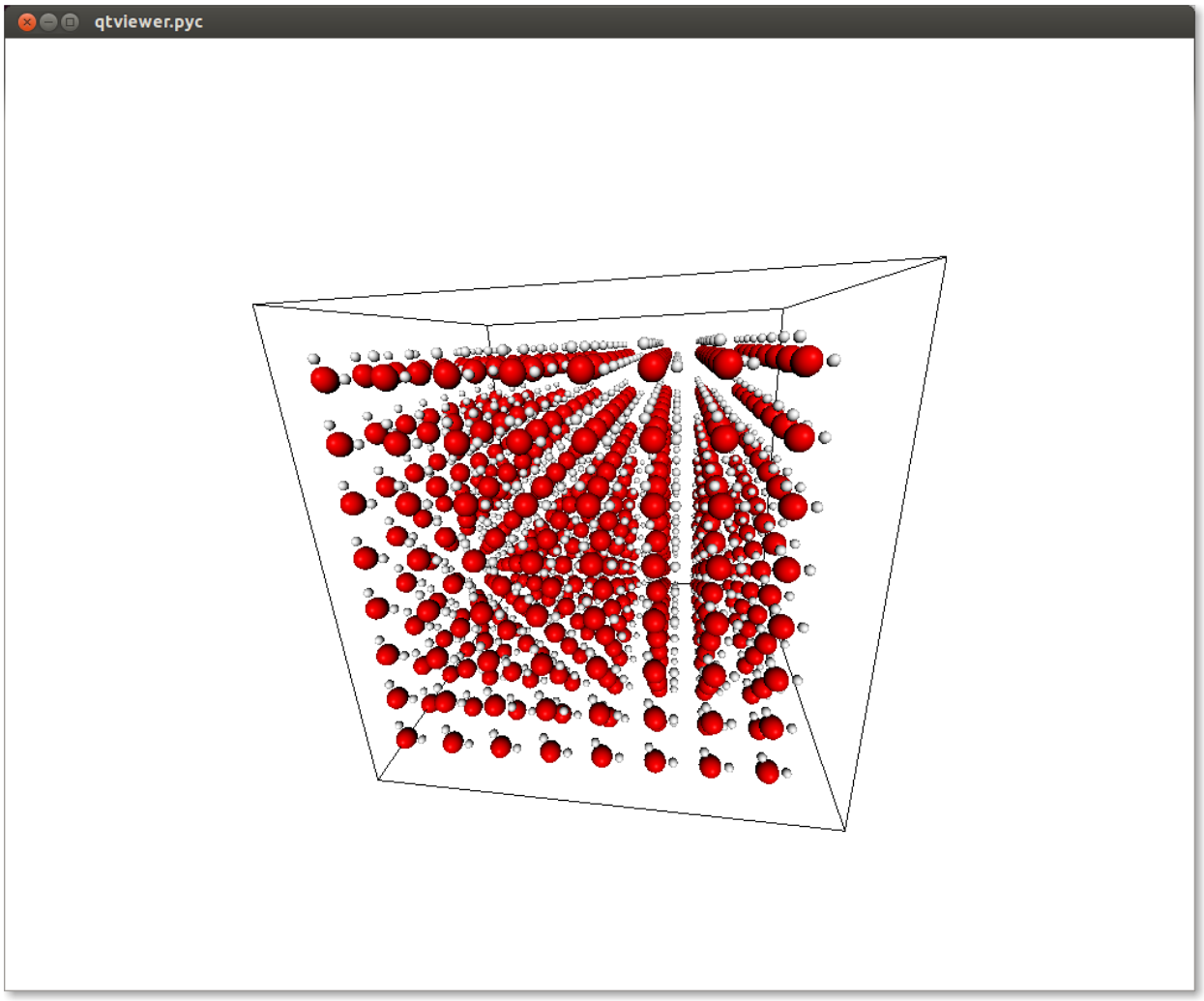
To generate such a box we will:

1. Create a template water `Molecule`;
2. Translate this molecule on the grid points
3. Add the molecule to a preinitialized `System`.

```
import numpy as np
from chemlab.core import Atom, Molecule, System
from chemlab.graphics import display_system

# Spacing between two grid points
spacing = 0.3

# an 8x8x8 grid, for a total of 512 points
grid_size = (8, 8, 8)

# Preallocate the system
# 512 molecules, and 512*3 atoms
s = System.empty(512, 512*3)

# Water template, it contains export informations for gromacs
# more about export later...
water_tmp = Molecule([Atom('O', [0.0, 0.0, 0.0], export={'grotype': 'OW'}),
                      Atom('H', [0.1, 0.0, 0.0], export={'grotype': 'HW1'}),
                      Atom('H', [-0.03333, 0.09428, 0.0], export={'grotype':'HW2'})],
                     export={'groname': 'SOL'})

for a in range(grid_size[0]):
    for b in range(grid_size[1]):
        for c in range(grid_size[2]):
            grid_point = np.array([a,b,c]) * spacing # array operation
            water_tmp.move_to(grid_point)
            s.add(water_tmp)

# Adjust boxsize for periodic boundary conditions
```

```
s.boxsize = 8 * spacing

# Visualize to verify that the system was setup correctly
display_system(s)
```

If you run this, it will display the following window:



Awesome! Now we can write the ".gro" file. Notice that when we defined our water molecule we had to pass an *export* dictionary to the atoms and molecules. The *export* mechanism is the way used by chemlab to handle all the variety of different file formats.

In this specific case, gromacs defines its own atom and molecule names in the ".top" file and then matches those to the ".gro" file to infer the bonds and interactions.

TODO Add picture of the export dictionary

How do we write the .gro file? Since we've already setup our export information, this is an one-liner:

```
from chemlab.io import datafile

datafile("start.gro").write("system", s)
```

2.5.4 .top and .mdp files

I'll give you directly the gromacs input files to do an NPT simulation of water, just create those files in your working directory:

topol.top

```
; We simply import ready-made definitions for the molecule type
; SOL and the atom types OW, HW1 and HW2
#include "ffoplsaa.itp"
#include "spce.itp"

[ system ]
Simple box of water

[ molecules ]
SOL 512
```

run.mdp

```
integrator = md
dt = 0.001
nsteps = 200000
nstxtcout = 100

rlist = 0.9
coulombtype = pme
rcoulomb = 0.9
rvdw = 0.9
dispcorr = enerpres

tcoupl = v-rescale
tc-grps = System
ref_t = 300
tau_t = 0.1

pcoupl = berendsen
compressibility = 4.5e-5
ref_p = 1.0

gen_vel = yes
gen_temp = 300
constraints = all-bonds
```

2.5.5 Running the simulation

To run the simulation with gromacs we have to do two steps:

1. Generate a parameter input, this will check that our input make sense before running the simulation:

   ```
   grompp_d -f run.mdp -c start.gro -p topol.top
   ```

   This will generate a bunch of files in your working directory.

2. Now we run the simulation, in the meantime, go grab coffee:

   ```
   mdrun_d -v
   ```

   This will take a while depending on your machine. If you are not a coffee drinker, don't worry, you can stop the simulation by pressing Ctrl-C. The good news is that chemlab can read files from partial runs!

2.5.6 Viewing the results, the command-line way

To quickly preview trajectories and system energies you can use the script *chemlab* included in the distribution in *scripts/chemlab*.

GROMACS can store the trajectory (in the form of atomic coordinates) in the *.xtc* file. To play the trajectory you can use the command:

```
$ chemlab view start.gro --traj traj.xtc
```

Note: the `nstxtcout = 100` option in the mdp file sets the output frequency in the xtc file

---

You may also be interested to look at some other properties, such as the potential energy, pressure, temperature and density. This information is written by GROMACS in the ".edr" file. You can use the chemlab script to view that:

```
$ chemlab gromacs ener.edr -e Pressure
$ chemlab gromacs ener.edr -e Temperature
$ chemlab gromacs ener.edr -e Potential
$ chemlab gromacs ener.edr -e Density
```

Warning: The chemlab gromacs command is a work in progress, the syntax may change in the future.

It is also possible to view and get the results by directly reading the files and have direct access to the xtc coordinates and the energy stored in the edr files. Take a look at the reference for `chemlab.io.handlers.XtcIO` and `chemlab.io.handlers.EdrIO`.

The tutorial is over, if you have any problem or want to know more, just drop an email on the mailing list python-chemlab@googlegroups.com or file an issue on github https://github.com/chemlab/chemlab/issues

CHAPTER

THREE

REFERENCE DOCUMENTATION

Packages

3.1 chemlab.core

This package contains general functions and the most basic data containers such as Atom, Molecule and System. Plus some utility functions to create and edit common Systems.

3.1.1 The Atom class

class chemlab.core.**Atom**(*type*, *r*, *export=None*)

Create an *Atom* instance. Atom is a generic container for particle data.

See Also:

Atoms, Molecules and Systems

Parameters

type: str Atomic symbol

r: {np.ndarray [3], list [3]} Atomic coordinates in nm

export: dict, optional Additional export information.

Example

```
>>> Atom('H', [0.0, 0.0, 0.0])
```

In this example we're attaching additional data to the *Atom* instance. The *chemlab.io.GroIO* can use this information when exporting in the gro format.

```
>>> Atom('H', [0.0, 0.0, 0.0], {'groname': 'HW1'})
```

type

Type str

The atomic symbol e.g. *Ar*, *H*, *O*.

r

Type np.ndarray(3) of floats

Atomic position in *nm*.

mass

Type float

Mass in atomic mass units.

export

Type dict

Dictionary containing additional information when importing data from various formats.

See Also:

`chemlab.io.gro.GroIO`

fields

Type tuple

This is a *class attribute*. The list of attributes that constitute the Atom. This is used to iterate over the *Atom* attributes at runtime.

copy()
Return a copy of the original Atom.

classmethod from_fields(***kwargs*)
Create an *Atom* instance from a set of fields. This is a slightly faster way to initialize an Atom.

Example

```
>>> Atom.from_fields(type='Ar',
                     r_array=np.array([0.0, 0.0, 0.0]),
                     mass=39.948,
                     export={})
```

3.1.2 The Molecule class

class chemlab.core.**Molecule**(*atoms*, *export=None*)
Molecule is a data container for a set of *N Atoms*.

See Also:

Atoms, Molecules and Systems

Parameters

atoms: list of Atom instances Atoms that constitute the Molecule. Beware that the data **gets copied** and subsequend changes in the *Atom* instances will not reflect in the *Molecule*.

export: dict, optional Export information for the Molecule

r_array

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

Derived from Atom

An array with the coordinates of each *Atom*.

type_array {numpy.array[N] of str}

Type np.ndarray(N, dtype=str)

Derived from Atom

An array containing the chemical symbols of the constituent atoms.

m_array

Type np.ndarray(N, dtype=float)

Derived from Atom

Array of masses.

atom_export_array

Type np.ndarray(N, dtype=object) *array of dicts*

Derived from Atom

Array of *Atom.export* dicts.

n_atoms

Type int

Number of atoms present in the molecule.

export

Type dict

Export information for the whole Molecule.

mass

Type float

Mass of the whole molecule in *amu*.

center_of_mass

Type float

geometric_center

Type float

formula

Type str

The brute formula of the Molecule. i.e. `"H2O"`

copy()
Return a copy of the molecule instance

classmethod from_arrays(***kwargs*)
Create a Molecule from a set of Atom-derived arrays. Please refer to the Molecule *Atom Derived Attributes*. Only *r_array* and *type_array* are absolutely required, the others are optional.

```
>>> Molecule.from_arrays(r_array=np.array([[0.0, 0.0, 0.0],
                                           [1.0, 0.0, 0.0],
                                           [0.0, 1.0, 0.0]]),
                         type_array=np.array(['O', 'H', 'H']))
molecule(H2O)
```

Initializing a molecule in this way can be much faster than the default initialization method.

move_to(*r*)
Translate the molecule to a new position *r*.

3.1.3 The System class

class chemlab.core.**System**(*molecules*, *boxsize=None*, *box_vectors=None*)
A data structure containing information of a set of *N* Molecules and *NA* Atoms.

Parameters

molecules: list of molecules Molecules that constitute the System. The data **gets copied** to the System, subsequent changes to the Molecule are not reflected in the System.

boxsize: float, optional The size of one side of a cubic box containing the system. Periodic boxes are common in molecular dynamics.

box_vectors: np.ndarray((3,3), dtype=float), optional You can specify the periodic box of another shape by giving 3 box vectors instead.

The System class has attributes derived both from the Molecule and the Atom class.

r_array

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

Derived from Atom

Atomic coordinates.

m_array

Type np.ndarray(NA, dtype=float)

Derived from Atom

Atomic masses.

type_array

Type np.ndarray(NA, dtype=object) *array of str*

Derived from Atom

Array of all the atomic symbols. It can be used to select certain atoms in a system.

Example

Suppose you have a box of water defined by the System *s*, to select all oxygen atoms you can use the numpy selection rules:

```
>>> oxygens = s.type_array == 'O'
# oxygens is an array of booleans of length NA where
# each True corresponds to an oxygen atom i.e:
# [True, False, False, True, False, False]
```

You can use the *oxygen* array to access other properties:

```
>>> o_coordinates = s.r_array[oxygens]
>>> o_indices = np.arange(s.n_atoms)[oxygens]
```

atom_export_array

Type np.ndarray(NA, dtype=object) *array of dict*

Derived from Atom

mol_export

Type np.ndarray(N, dtype=object) *array of dict*

Derived from Molecule

Export information relative to the molecule.

box_vectors

Type np.ndarray((3,3), dtype=float) or None

Those are the three vectors that define of the periodic box of the system.

Example

To define an orthorombic box of size 3, 4, 5 nm:

```
>>> np.array([[3.0, 0.0, 0.0],  # Vector a
              [0.0, 4.0, 0.0],  # Vector b
              [0.0, 0.0, 5.0]]) # Vector c
```

boxsize, optional

Type float or None

Defines the size of the periodic box. Boxes defined with boxsize are cubic. Changes in *boxsize* are reflected in box.

n_mol

Type int

Number of molecules.

n_atoms

Type int

Number of atoms.

mol_indices

Type np.ndarray(N, dtype=int)

Gives the starting index for each molecule in the atomic arrays. For example, in a System comprised of 3 water molecules:

```
>>> s.mol_indices
[0, 3, 6]
>>> s.type_array[0:3]
['O', 'H', 'H']
```

This array is used internally to retrieve all the Molecule derived data. Do not modify unless you know what you're doing.

mol_n_atoms

Type np.ndarray(N, dtype=int)

Contains the number of atoms present in each molecule

add(*mol*)

Add the molecule *mol* to a System initialized through `System.empty`.

classmethod empty(*n_mol*, *n_atoms*, *boxsize=None*, *box_vectors=None*)

Initialize an empty System containing *n_mol* Molecules and *n_atoms* Atoms. The molecules can be added by using the method `add()`.

Example

How to initialize a system of 3 water molecules:

```
s = System.empty(3, 9)
for i in range(3):
    s.add(water)
```

classmethod from_arrays(***kwargs*)
Initialize a System from its constituent arrays. It is the fastest way to initialize a System, well suited for reading one or more big System from data files.

Parameters

The following parameters are required:

- r_array
- type_array
- mol_indices

To further speed up the initialization process you optionally pass the other derived arrays:

- m_array
- mol_n_atoms
- atom_export_array
- mol_export

Example

Our classic example of 3 water molecules:

```
r_array = np.random.random((3, 9))
type_array = ['O', 'H', 'H', 'O', 'H', 'H', 'O', 'H', 'H']
mol_indices = [0, 3, 6]
System.from_arrays(r_array=r_array, type_array=type_array,
                   mol_indices=mol_indices)
```

get_molecule(*index*)
Get the Molecule instance corresponding to the molecule at *index*.

This method is useful to use Molecule properties that are generated each time, such as Molecule.formula and Molecule.center_of_mass

mol_to_atom_indices(*indices*)
Given the indices over molecules, return the indices over atoms.

sort()
Sort the molecules in the system according to their brute formula.

3.1.4 Routines to manipulate Systems

chemlab.core.**subsystem_from_molecules**(*orig*, *selection*)
Create a system from the *orig* system by picking the molecules specified in *selection*.

Parameters

orig: System The system from where to extract the subsystem

selection: np.ndarray of int or np.ndarray(N) of bool *selection* can be either a list of molecular indices to select or a boolean array whose elements are True in correspondence of the molecules to select (it is usually the result of a numpy comparison operation).

Example

In this example we can see how to select the molecules whose center of mass that is in the region of space x > 0.1:

```
s = System(...) # It is a set of 10 water molecules

select = []
for i range(s.n_mol):
   if s.get_molecule(i).center_of_mass[0] > 0.1:
       select.append(i)

subs = subsystem_from_molecules(s, np.ndarray(select))
```

Note: The API for operating on molecules is not yet fully developed. In the future there will be smarter ways to *filter* molecule attributes instead of looping and using System.get_molecule.

---

chemlab.core.**subsystem_from_atoms**(*orig*, *selection*)

Generate a subsystem containing the atoms specified by *selection*. If an atom belongs to a molecule, the whole molecule is selected.

Example

This function can be useful when selecting a part of a system based on positions. For example, in this snippet you can see how to select the part of the system (a set of molecules) whose x coordinates is bigger than 1.0 nm:

```
s = System(...)
subs = subsystem_from_atoms(s.r_array[0,:] > 1.0)
```

Parameters

orig: System Original system.

selection: np.ndarray of int or np.ndarray(NA) of bool A boolean array that is True when the ith atom has to be selected or a set of atomic indices to be included.

Returns:

A new System instance.

chemlab.core.**merge_systems**(*sysa*, *sysb*, *bounding=0.0*)

Generate a system by overlapping *sysa* and *sysb*. Overlapping molecules are removed by cutting the molecules of *sysa* that are found inside the space defined by `sysb.box_vectors`.

Parameters

sysa: System First system

sysb: System Second system

bounding: float Extra space used when cutting molecules in *sysa* to make space for *sysb*.

3.1.5 Routines to create Systems

chemlab.core.**crystal**(*positions, molecules, group, cellpar=[1.0, 1.0, 1.0, 90, 90, 90], repetitions=[1, 1, 1]*)

Build a crystal from atomic positions, space group and cell parameters.

Parameters

positions: list of coordinates A list of the atomic positions

molecules: list of Molecule The molecules corresponding to the positions, the molecule will be translated in all the equivalent positions.

group: int | str Space group given either as its number in International Tables or as its Hermann-Mauguin symbol.

repetitions: Repetition of the unit cell in each direction

cellpar: Unit cell parameters

This function was taken and adapted from the *spacegroup* module found in ASE.

The module *spacegroup* module was originally developed by Jesper Frills.

3.2 chemlab.io

This package contains utilities to read, write a variety of chemical file formats.

chemlab.io.**datafile**(*filename*, *format=None*)

Initialize the appropriate `IOHandler` for a given file extension or file format.

The *datafile* function can be conveniently used to quickly read or write data in a certain format:

```
>>> handler = datafile("molecule.pdb")
>>> mol = handler.read("molecule")
# You can also use this shortcut
>>> mol = datafile("molecule.pdb").read("molecule")
```

Parameters

filename: str Path of the file to open.

format: str or None When different from *None*, can be used to specify a format identifier for that file. It should be used when the extension is ambiguous or when there isn't a specified filename. See below for a list of the formats supported by chemlab.

3.2.1 Supported File Formats

edr: GROMACS energy file

Extension .edr

class chemlab.io.handlers.**EdrIO**(*filename*)

EDR files store per-frame information for gromacs trajectories. Examples of properties obtainable from EDR files are:

```
- temperature
- pressure
- density
- potential energy
- total energy
- etc.
```

To know which quantities are available in a certain edr file you can access the feature 'avail quantity':

```
>>> datafile('ener.edr').read('avail quantities')
['Temperature', 'Pressure', 'Potential', ...]
```

To get the frame information for a certain quantity you may use the "quantity" property passing the quantity as additional argument, this will return two arrays, the first is an array of times in ps and the second are the corrisponding quantities:

```
>>> time, temp = datafile('ener.edr').read('quantity', 'Temperature')
```

Features

read(*"quantity"*, *quant*)
Return an array of times in ps and the corresponding quantities at that times.

read(*"avail quantities"*)
Return the available quantities in the file.

read(*"units"*)
Return a dictionary where the keys are the quantities and the value are the units in which that quantity is expressed.

read(*"frames"*)
Return a dictionary where the keys are the quantities and the value are the units in which that quantity is expressed.

gro: GROMACS coordinate files

Extension .gro

class chemlab.io.handlers.**GromacsIO**(*filename*)
Handler for .gro file format. Example at http://manual.gromacs.org/online/gro.html.

Features

read(*"system"*)
Read the gro file and return a `System` instance. It also add the following exporting informations:

groname: The molecule names indicated in the gro file. This is added to each entry of *System.mol_export*.

grotype: The atom names as indicated in the gro file. This is added to each entry of *System.atom_export_array*.

write(*"system"*, *syst*)
Write the *syst* `System` instance to disk. The export arrays should have the *groname* and *grotype* entries as specified in the `read("system")` method.

Example

Export informations for water SPC:

```
Molecule([
          Atom('O', [0.0, 0.0, 0.0], export={'grotype': 'OW'}),
          Atom('H', [0.1, 0.0, 0.0], export={'grotype': 'HW1'}),
          Atom('H', [-0.033, 0.094, 0.0],export={'grotype':'HW2'})],
         export={'groname': 'SOL'})
```

pdb: Protein Data Bank format

Extension .pdb

class chemlab.io.handlers.**PdbIO**(*filename*)
Starting implementation of a PDB file parser.

Note: This handler was developed as an example. If you like to contribute by implementing it you can write an email to the mailing list.

Features

read(*"molecule"*)
Read the pdb file as a huge Molecule.

read(*"system"*)
Read the pdb file as a System, where each residue is a molecule.

xtc: GROMACS compressed trajectory file

Extension .xtc

class chemlab.io.handlers.**XtcIO**(*filename*)
Reader for GROMACS XTC trajectories.

Features

read(*"trajectory"*)
Read the frames from the file and returns the trajectory as an array of times and an array of atomic positions:

```
>>> times, positions = datafile('traj.xtc').read('trajectory')
[t1, t2, t3], [pos1, pos2, ...]
```

positions is a *list* of `np.ndarray(n_atoms, 3)`.

xyz: XYZ coordinate format

Extension .xyz

class chemlab.io.handlers.**XyzIO**(*filename*)
The XYZ format is described in this wikipedia article http://en.wikipedia.org/wiki/XYZ_file_format.

Features

read(*"molecule"*)
Read the coordinates in a `Molecule` instance.

write(*"molecule"*, *mol*)
Writes a `Molecule` instance in the XYZ format.

3.2.2 The class IOHandler

class chemlab.io.handlers.**IOHandler**(*filename*)
Generic base class for file readers and writers. The initialization function takes *filename* as input and sets the instance attribute *filename*.

Subclasses can extend the methods *__init__*, *read* and *write* to implement their reading and writing routines.

Attributes

filename

can_read

Type list of str

A list of *features* that the handler can read.

can_write

Type list of str

A list of *features* that IOHandler can write.

check_feature(*feature*, *readwrite*)
Check if the *feature* is supported in the handler and raise an exception otherwise.

Parameters

feature: str Identifier for a certain feature.

readwrite: "read" or "write" Check if the feature is available for reading or writing.

read(*feature*, **args*, ***kwargs*)
Read and return the feature *feature*. It should raise an ValueError if the feature is not present in the handler *can_read* attribute, use the method `IOHandler.check_feature()` to provide this behaviour.

Certain features may require additional arguments, and it is possible to pass those as well.

Example

Subclasses can reimplement this method to add functionality:

```
class XyzIO(IOHandler):
    can_read = ['molecule']

    def read(self, feature, *args, **kwargs):
        self.check_feature(feature, "read")
        if feature == 'molecule':
            # Do stuff
            return geom
```

write(*feature*, *value*, **args*, ***kwargs*)
Same as `read()`. You have to pass also a *value* to write and you may pass any additional arguments.

Example

```
class XyzIO(IOHandler):
    can_write = ['molecule']

    def write(self, feature, value, *args, **kwargs):
        self.check_feature(feature, "write")
        if feature == 'molecule':
            # Do stuff
            return geom
```

3.3 chemlab.graphics

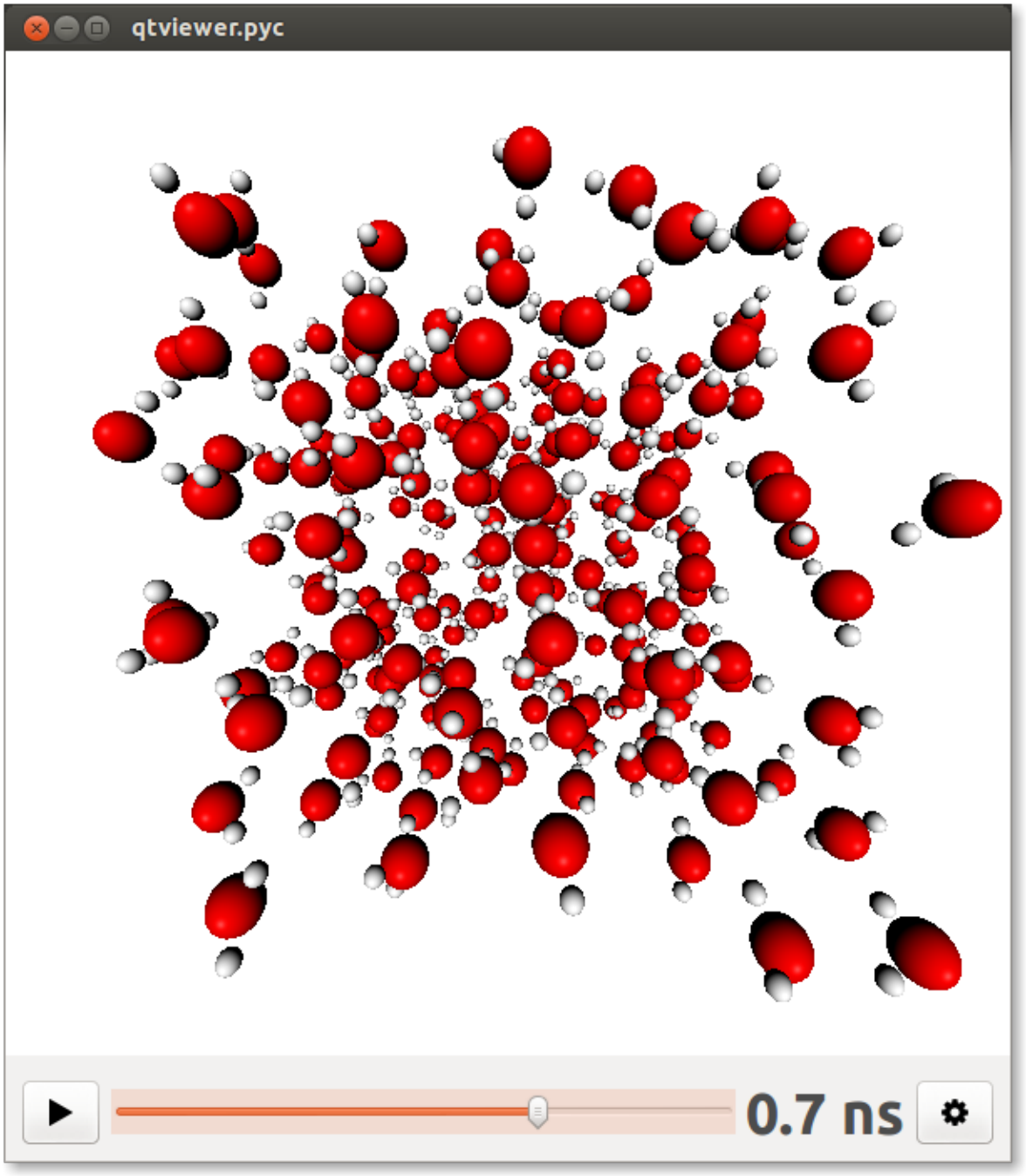
This package contains the features related to the graphic capabilities of chemlab.

3.3.1 Ready to use functions

The two following functions are a convenient way to quickly display and animate a `System` in chemlab.

`chemlab.graphics.`**`display_system`**(*sys*)
Display the system *sys* with the default viewer.

`chemlab.graphics.`**`display_trajectory`**(*sys*, *times*, *coords_list*)
Display the the system *sys* and instrument the trajectory viewer with frames information.



Parameters

sys: `System` The system to be displayed

times: np.ndarray(NFRAMES, dtype=float) The time corresponding to each frame. This is used only for feedback reasons.

coords_list: list of np.ndarray((NFRAMES, 3), dtype=float) Atomic coordinates at each frame.

3.3.2 Builtin 3D viewers

The QtViewer class

class chemlab.graphics.**QtViewer**
Bases: *PySide.QtGui.QMainWindow*

View objects in space.

This class can be used to build your own visualization routines by attaching *renderers* and *uis* to it.

See Also:

Graphics and Visualization

Example

In this example we can draw 3 blue dots and some overlay text:

```
from chemlab.graphics import QtViewer
from chemlab.graphics.renderers import PointRenderer
from chemlab.graphics.uis import TextUI

vertices = [[0.0, 0.0, 0.0], [0.0, 1.0, 0.0], [2.0, 0.0, 0.0]]
blue = (0, 255, 255, 255)

colors = [blue,] * 3

v = QtViewer()

pr = v.add_renderer(PointRenderer, vertices, colors)
tu = v.add_ui(TextUI, 100, 100, 'Hello, world!')

v.run()
```

add_renderer(*klass*, **args*, ***kwargs*)
Add a renderer to the current scene.

Parameter

klass: renderer class The renderer class to be added

args, kwargs: Arguments used by the renderer constructor, except for the *widget* argument.

See Also:

AbstractRenderer

Return

The istantiated renderer. You should keep the return value to be able to update the renderer at run-time.

add_ui(*klass*, **args*, ***kwargs*)
Add an UI element for the current scene. The approach is the same as renderers.

> **Warning:** The UI api is not yet finalized

run()
Display the QtViewer

schedule(*callback*, *timeout=100*)
Schedule a function to be called repeated time.

This method can be used to perform animations.

Example

This is a typical way to perform an animation, just:

```
from chemlab.graphics import QtViewer
from chemlab.graphics.renderers import SphereRenderer

v = QtViewer()
sr = v.add_renderer(SphereRenderer, centers, radii, colors)

def update():
    # calculate new_positions
    sr.update_positions(new_positions)
    v.widget.repaint()

v.schedule(update)
v.run()
```

Note: remember to call QtViewer.widget.repaint() each once you want to update the display.

Parameters

callback: function() A function that takes no arguments that will be called at intervals.

timeout: int Time in milliseconds between calls of the *callback* function.

Returns a *QTimer*, to stop the animation you can use *Qtimer.stop*

The QtTrajectoryViewer class

class chemlab.graphics.**QtTrajectoryViewer**
Bases: *PySide.QtGui.QMainWindow*

Interface for viewing trajectory.

It provides interface elements to play/pause and set the speed of the animation.

Example

To set up a QtTrajectoryViewer you have to add renderers to the scene, set the number of frames present in the animation by calling ;py:meth:*~chemlab.graphics.QtTrajectoryViewer.set_ticks* and define an update function.

Below is an example taken from the function chemlab.graphics.display_trajectory():

```
from chemlab.graphics import QtTrajectoryViewer

# sys = some System
# coords_list = some list of atomic coordinates

v = QtTrajectoryViewer()
sr = v.add_renderer(AtomRenderer, sys.r_array, sys.type_array,
                    backend='impostors')
br = v.add_renderer(BoxRenderer, sys.box_vectors)
```

```
v.set_ticks(len(coords_list))

@v.update_function
def on_update(index):
    sr.update_positions(coords_list[index])
    br.update(sys.box_vectors)
    v.set_text(format_time(times[index]))
    v.widget.repaint()

v.run()
```

Warning: Use with caution, the API for this element is not fully stabilized and may be subject to change.

add_renderer (*klass*, **args*, ***kwargs*)
The behaviour of this function is the same as `chemlab.graphics.QtViewer.add_renderer()`.

add_ui (*klass*, **args*, ***kwargs*)
Add an UI element for the current scene. The approach is the same as renderers.

Warning: The UI api is not yet finalized

set_text (*text*)
Update the time indicator in the interface.

set_ticks (*number*)
Set the number of frames to animate.

update_function (*func*)
Set the function to be called when it's time to display a frame.

func should be a function that takes one integer argument that represents the frame that has to be played:

```
def func(index):
    # Update the renderers to match the
    # current animation index
```

3.3.3 Renderers and UIs

List of available renderers

Interfaces

class chemlab.graphics.renderers.**AbstractRenderer** (*widget*, **args*, ***kwargs*)
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 and Visualization 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 chemlab.graphics.renderers.**ShaderBaseRenderer**(*widget*, *vertex*, *fragment*)
Bases: chemlab.graphics.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

draw_vertices()
Method to be reimplemented by the subclasses.

class chemlab.graphics.renderers.**DefaultRenderer**(*widget*)
Bases: chemlab.graphics.renderers.base.ShaderBaseRenderer

Same as ShaderBaseRenderer with the default shaders.

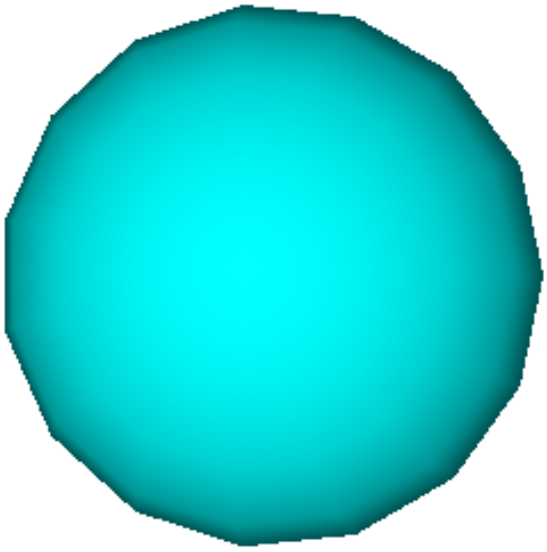
You can find the shaders in `chemlab/graphics/renderers/shaders/` under the names of `default_persp.vert` and `default_persp.frag`.

draw_vertices()
Subclasses should reimplement this method.

SphereRenderer

class chemlab.graphics.renderers.**SphereRenderer**(*widget*, *poslist*, *radiuslist*, *colorlist*)
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]

update_positions(*positions*)
Update the sphere positions.

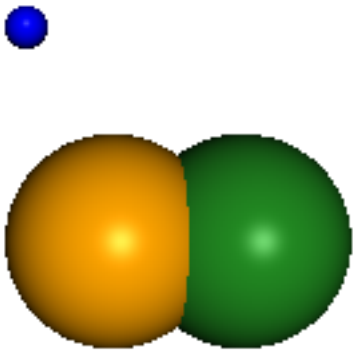
SphereImpostorRenderer

class `chemlab.graphics.renderers.`**SphereImpostorRenderer**(*viewer*, *poslist*, *radiuslist*, *colorlist*)

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.



AtomRenderer

class chemlab.graphics.renderers.**AtomRenderer**(*widget*, *r_array*, *type_array*, *backend="impostors"*, *color_scheme=colors.default_atom_map*, *radii_map=vdw_dict*)

Render atoms by using different rendering methods.

Parameters

widget: The parent QChemlabWidget

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

type_array: np.ndarray((NATOMS, 3), dtype=object) An array containing all the atomic symbols like *Ar*, *H*, *O*. If the atomic type is unknown, use the *Xx* symbol.

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

color_scheme: dict, should contain the 'Xx' key,value pair A dictionary mapping atom types to colors. By default it is the color scheme provided by *chemlab.graphics.colors.default_atom_map*. The 'Xx' symbol value is taken as the default color.

radii_map: dict, should contain the 'Xx' key,value pair. A dictionary mapping atom types to radii. The default is the mapping contained in *chemlab.data.vdw.vdw_dict*

update_positions(*r_array*)
Update the atomic positions

PointRenderer

class chemlab.graphics.renderers.**PointRenderer**(*widget*, *positions*, *colors*)
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]

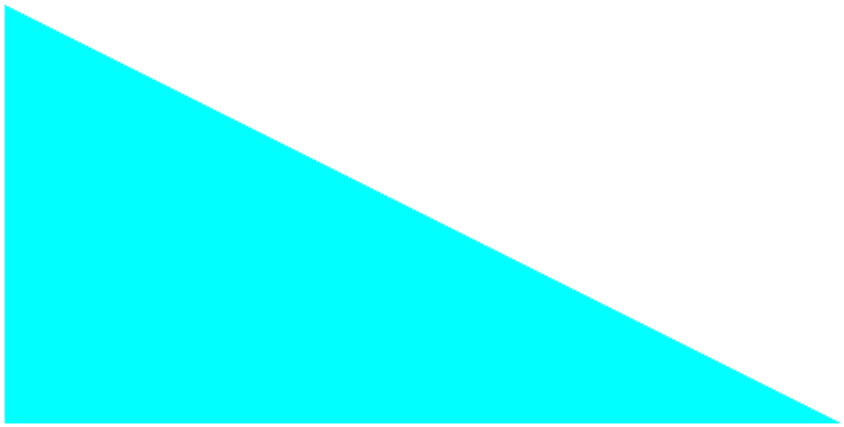
update_colors(*colors*)
Update the colors

update_positions(*vertices*)
Update the point positions

TriangleRenderer

class chemlab.graphics.renderers.**TriangleRenderer**(*widget*, *vertices*, *normals*, *colors*)
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]

update_colors(*colors*)
Update the triangle colors.

update_normals(*normals*)
Update the triangle normals.

update_vertices(*vertices*)
Update the triangle vertices.

BoxRenderer

class chemlab.graphics.renderers.**BoxRenderer**(*widget*, *vectors*, *color=(0, 0, 0, 255)*)
Used to render a black wireframed box starting from the origin.

Parameters

widget: The parent QChemlabWidget

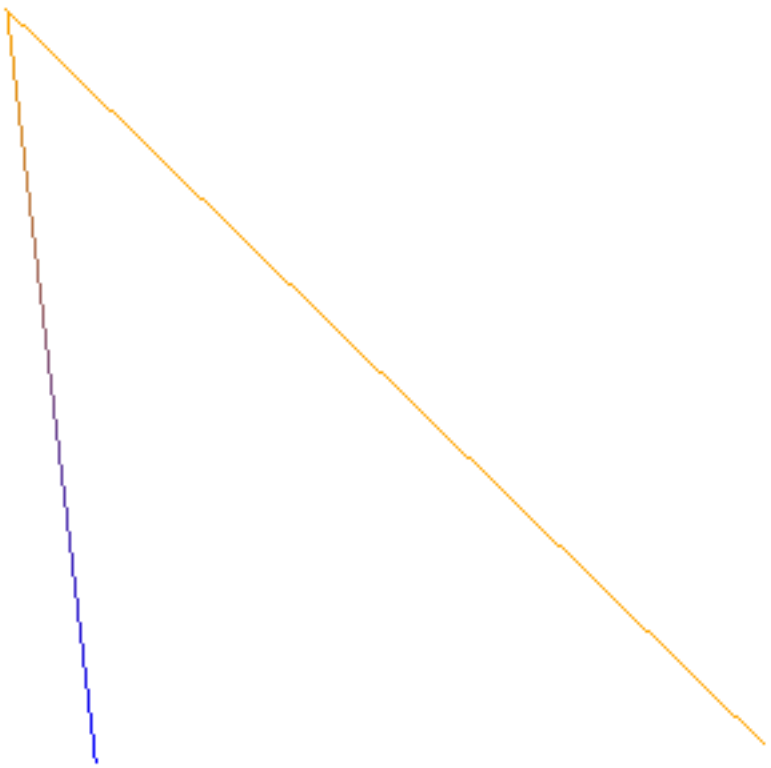
vectors: np.ndarray((3,3), dtype=float) The three vectors representing the sides of the box.

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

update(*vectors*)
Update the box vectors.

LineRenderer

class chemlab.graphics.renderers.**LineRenderer**(*widget*, *startends*, *colors*)
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.

update_colors(*colors*)
Update the colors

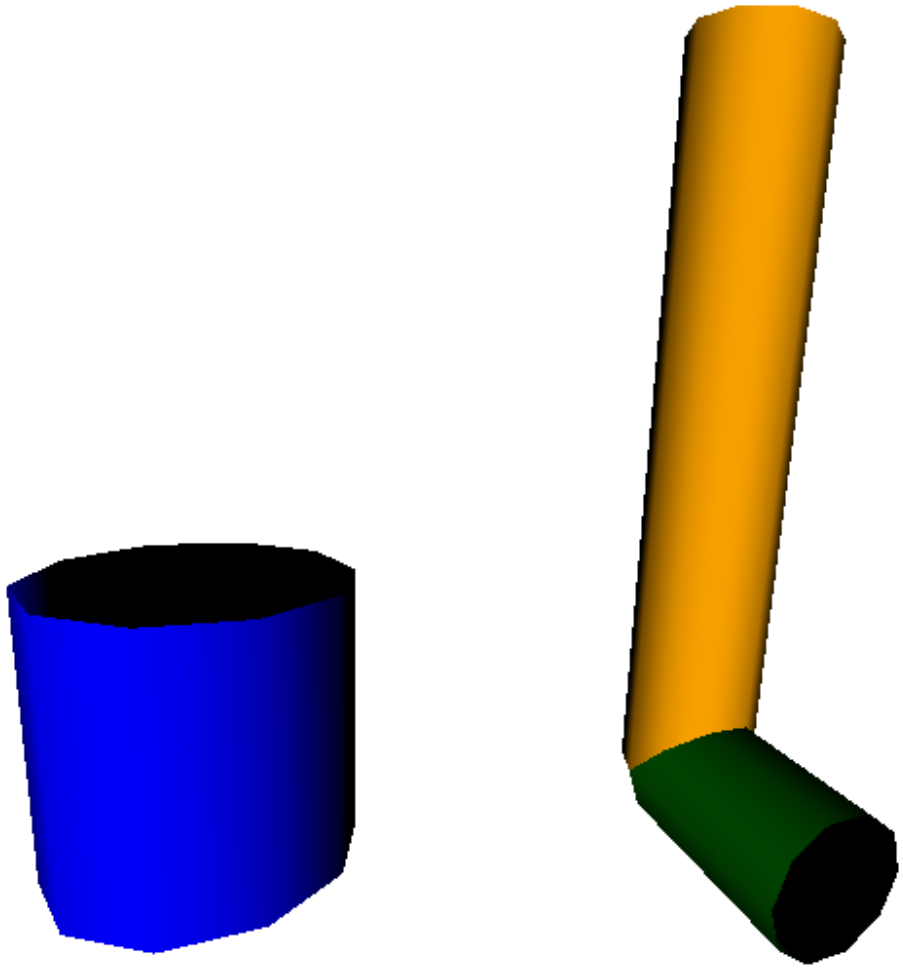
update_positions(*vertices*)
Update the line positions

CylinderRenderer

class chemlab.graphics.renderers.**CylinderRenderer**(*widget*, *bounds*, *radii*, *colors*)
Renders a set of cylinders.

The API is quite similar to LineRenderer

Note: The current implementation is a bit slow and can't render a lot of cylinders (~1000) fast enough, we expect to optimize it in the near future.



Parameters

widget: The parent QChemlabWidget

bounds: np.ndarray((NCYL, 2, 3), dtype=float) Start and end points of the cylinder.

colors: np.ndarray((NYCL, 4), dtype=np.uint8) The color for each cylinder.

update_bounds (*bounds*)
Update cylinders start and end positions

List of available UIs

TextUI

class chemlab.graphics.uis.**TextUI** (*widget*, *x*, *y*, *text*)
Display an overlay text at the point *x*, *y* in screen space.

Warning: The API for this element and uis in general is not yet finalized.

Parameters

widget: The parent *QChemlabWidget*

x, y: int Points in screen coordinates. *x* pixels from left, *y* pixels from top.

text: str String of text to display

3.3.4 Low level widgets

The QChemlabWidget class

This is the molecular viewer widget used by chemlab.

class `chemlab.graphics.`**`QChemlabWidget`**(*parent=None*)

Extensible and modular OpenGL widget developed using the Qt (PySide) Framework. This widget can be used in other PySide programs.

The widget by itself doesn't draw anything, it delegates the writing task to external components called 'renderers' that expose the interface found in `AbstractRenderer`. Renderers are responsible for drawing objects in space and have access to their parent widget.

To attach a renderer to QChemlabWidget you can simply append it to the `renderers` attribute:

```
from chemlab.graphics import QChemlabWidget
from chemlab.graphics.renderers import SphereRenderer

widget = QChemlabWidget()
widget.renderers.append(SphereRenderer(widget, ...))
```

You can also add other elements for the scene such as user interface elements, for example some text. This is done in a way similar to renderers:

```
from chemlab.graphics import QChemlabWidget
from chemlab.graphics.uis import TextUI

widget = QChemlabWidget()
widget.uis.append(TextUI(widget, 200, 200, 'Hello, world!'))
```

> **Warning:** At this point there is only one ui element available. PySide provides a lot of UI elements so there's the possibility that UI elements will be converted into renderers.

QChemlabWidget has its own mouse gestures:

- Left Mouse Drag: Orbit the scene;
- Right Mouse Drag: Pan the scene;
- Wheel: Zoom the scene.

`renderers`

Type list of `AbstractRenderer` subclasses

It is a list containing the active renderers. QChemlabWidget will call their `draw` method when appropriate.

`camera`

Type `Camera`

The camera encapsulates our viewpoint on the world. That is where is our position and our orientation. You should use on the camera to rotate, move, or zoom the scene.

light_dir

Type np.ndarray(3, dtype=float)

Default np.arrray([0.0, 0.0, 1.0])

The light direction in camera space. Assume you are in the space looking at a certain point, your position is the origin. now imagine you have a lamp in your hand. *light_dir* is the direction this lamp is pointing. And if you move, jump, or rotate, the lamp will move with you.

Note: With the current lighting mode there isn't a "light position". The light is assumed to be infinitely distant and light rays are all parallel to the light direction.

---

background_color

Type tuple

Default (255, 255, 255, 255) white

A 4-element (r, g, b, a) tuple that specity the background color. Values for r,g,b,a are in the range [0, 255]. You can use the colors contained in chemlab.graphics.colors.

paintGL()
GL function called each time a frame is drawn

The Camera class

class chemlab.graphics.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/

scale

Type float

Scale factor 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.

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.

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).

3.3.5 Transformations

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 chemlab.graphics.transformations.**Arcball**(*initial=None*)

Virtual Trackball Control.

```
>>> ball = Arcball()
>>> ball = Arcball(initial=numpy.identity(4))
>>> ball.place([320, 320], 320)
>>> ball.down([500, 250])
>>> ball.drag([475, 275])
>>> R = ball.matrix()
>>> numpy.allclose(numpy.sum(R), 3.90583455)
True
>>> ball = Arcball(initial=[1, 0, 0, 0])
>>> ball.place([320, 320], 320)
>>> ball.setaxes([1, 1, 0], [-1, 1, 0])
>>> ball.setconstrain(True)
>>> ball.down([400, 200])
>>> ball.drag([200, 400])
>>> R = ball.matrix()
>>> numpy.allclose(numpy.sum(R), 0.2055924)
True
>>> ball.next()
```

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.

chemlab.graphics.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()

chemlab.graphics.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
```

chemlab.graphics.transformations.**arcball_constrain_to_axis**(*point*, *axis*)

Return sphere point perpendicular to axis.

chemlab.graphics.transformations.**arcball_map_to_sphere**(*point*, *center*, *radius*)

Return unit sphere coordinates from window coordinates.

chemlab.graphics.transformations.**arcball_nearest_axis**(*point*, *axes*)

Return axis, which arc is nearest to point.

chemlab.graphics.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.])
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.transformations.**distance**(*x1*, *x2*)
Distance between two points in space

chemlab.graphics.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")
```

chemlab.graphics.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
```

chemlab.graphics.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)
```

chemlab.graphics.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
```

chemlab.graphics.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)
```

chemlab.graphics.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
```

chemlab.graphics.transformations.**normalized**(*x*)
Return the x vector normalized

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.transformations.**quaternion_imag**(*quaternion*)

Return imaginary part of quaternion.

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

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.transformations.**quaternion_real**(*quaternion*)

Return real part of quaternion.

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

chemlab.graphics.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
```

chemlab.graphics.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)
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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

chemlab.graphics.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
```

chemlab.graphics.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)
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.transformations.**simple_clip_matrix**(*scale*, *znear*, *zfar*, *aspectratio=1.0*)
Given the parameters for a frustum returns a 4x4 perspective projection matrix

Parameters: float scale: float znear,zfar: near/far plane z, float

Return: a 4x4 perspective matrix

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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
```

chemlab.graphics.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]
```

chemlab.graphics.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
```

chemlab.graphics.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
```

CHAPTER

FOUR

LICENSE

Chemlab is released under the GNU GPLv3 and its main developer is Gabriele Lanaro.

PYTHON MODULE INDEX

C

chemlab.graphics.transformations, ??