Ionic Energy

September 7, 2017

1 Ionic Energy Calculations

This module collects together the energetic calculations for ionic materials.

The aim of any energy calculation here is to make no assumptions about the atomic configuration being input and instead accept an array of atom coordinates:

$$atoms = \begin{pmatrix} x_1 & y_1 & z_1 & q_1 \\ x_2 & y_2 & z_2 & q_2 \\ . & . & . & . \\ . & . & . & . \\ x_n & y_n & z_n & q_n \end{pmatrix}$$

and calculate the total energy of the configuration. Any normalisation or analysis is not done here.

There are two interactions used to characterise the energy of an ionic solid. There is the electrostatic interation and some shorter range interaction, modelled here with the Lennard-Jones empirical potential.

The interaction between the ith and jth ion are described by:

$$U_{ij}^{electrostatic} = \frac{1}{4\pi\epsilon_0}\frac{q_i q_j}{r_{ij}}$$

$$U_{ij}^{LJ} = \frac{A_{ij}}{r_{ij}^{12}} - \frac{B_{ij}}{r_{ij}^{6}}$$

Hence the total energy is the sum over i and j.

```
from __future__ import division, print_function, absolute_import, with_statement

import numpy as np
import scipy as sp
from scipy import constants
import math
import time
import os
import build
import sys
from lammps import lammps
```

```python
def calculate_ionic_energy(atoms,
                           energies=np.array([0,0,0]),
                           system=build.NaCl_110_110,
                           label='temp'):
    # This combines the coulombic interactions
    # and some short range repulsion potentiale.
    # Currently the classic 6-12 Lennard Jones
    # we are assuming that the values for atomic positions
    # are in angstroms and that the atomic charges are
    # in elemental charges.
    ################################################
    # First we see if a previous run has checkpointed and aborted.
    # If the atoms contained in a temp_atoms.npy file
    # match the tail end of the  array of atoms we have exactly
    # then the simulation must have had the same
    # parameters at creation. So we can skip
    # the leading end of the array.

    test = os.path.isfile(label+'temp_atoms.npy')
    test = test and os.path.isfile(label+'temp_energies.npy')
    if test:
        atoms_from_file = np.load(label+'temp_atoms.npy')
        if np.array_equal(atoms_from_file, atoms[-len(atoms_from_file):]):
        energies = np.load(label+'temp_energies.npy')
        atoms = atoms_from_file

    total_energy = energies[0]
    electrostatic_energy = energies[0]
    repulsive_energy = energies[0]
    timer_start = time.time()

    for n, ref_atom in enumerate(atoms):
        # Here we define a reference atom that we
        # will calculate the interactions for.
        # Once we have defined the reference atom
        # as the first atom in th list we remove the
        # reference atom from the array.
        # This means we don't consider self interactions, and
        # avoid double counting. It also removes the need to deal
        # with dividing by zero in the coulombic calculation.
        ref_charge = ref_atom[3]
        ref_position = ref_atom[:3]
        remaining_atoms = atoms[n+1:]
        step_size = 5000000

        for m in range(0,remaining_atoms.shape[0],step_size):
            # We run through the atoms in
            # slices so we don't fill up memory
            atoms_slice = remaining_atoms[m:m+step_size]
            charges = atoms_slice[:,3]
            positions = atoms_slice[:,:3] # in angstroms
            displacements = positions - ref_position
            seps_squared = np.sum(np.square(displacements),axis=1)
            separations = seps_squared ** 0.5

            mask = separations < 320 * system['r0']

            masked_seps_squared = seps_squared[mask]
            # Using only those bonds less than 320 time
            # slarger than the smallest one will place
            # an upper bound on the computation required
            # without affecting the results
            # at 320 times the shortest bond the LJ term
            # is below the machine precision of a
            # 64 bit float.
            masked_seps_to_minus6 = np.power(masked_seps_squared, -3)
            # as scalars
            charge_products = charges * ref_charge
            masked_charge_products = charge_products[mask]

            electrostatic_energy += coulomb_energy(separations,
            charge_products)
            repulsive_energy += LJ6_energy(masked_seps_to_minus6,
            masked_charge_products,
            ref_charge,
            system=system)
            masked_seps_to_minus12 = masked_seps_to_minus6**2
            repulsive_energy += LJ12_energy(masked_seps_to_minus12,
            masked_charge_products,
            ref_charge,
            system=system)
            total_energy = electrostatic_energy + repulsive_energy
        # If siginificant time has past then a checkpoint will be created
        # We have found the energy for each atom iterated over so far wrt
        # each other and to the atoms in remaining atoms. Hence saving
        # the energies and the remaining atoms provides a suitable restart point
        if time.time() - timer_start > 1000:
            energies = np.array([total_energy,
            electrostatic_energy,
            repulsive_energy])
            np.save(label+'temp_atoms.npy', remaining_atoms)
            np.save(label+'temp_energies.npy', energies)
            timer_start = time.time()

    # Return the energies all in absolute values of joules
    # delete the check point files on a succesful completion
    test = os.path.isfile(label+'temp_atoms.npy')
    test = test and os.path.isfile(label+'temp_energies.npy')
    if test:
        os.remove(label+'temp_atoms.npy')
        os.remove(label+'temp_energies.npy')
    return total_energy, electrostatic_energy, repulsive_energy #in Joules

# The Lennard Jones Parameters are included in the details of the system

def LJ6_energy(seps_to_minus6, charge_products, ref_charge, system=build.NaCl):
    # the classic lennard jones, the 6 term
    energy = 0
    # we always have the cation -anion interactions,
    # and the charge product for these is always -1
    # So mask the like-like interactions where qq > 0

    # we now have a matrix of distances with
    # only the reference cation to anion distances unmasked
    ca_energy = - system['B_ca'] * np.sum(seps_to_minus6[charge_products < 0])
    energy += ca_energy

    if ref_charge > 0: #i.e. a cation
        # cation-cation interaction qq > 0
        # mask where qq < 0
        cc_energy = - system['B_cc'] * np.sum(seps_to_minus6[charge_products>0])
        energy += cc_energy

    if ref_charge < 0: #i.e. an anion
        # anion-anion interaction qq > 0
        # mask where qq < 0
        aa_energy = - system['B_cc'] * np.sum(seps_to_minus6[charge_products>0])
        energy += aa_energy

    return energy #in joules

def LJ12_energy(seps_to_minus12, charge_products, ref_charge, system=build.NaCl):
    # The order 12 term of the Lennard-Jones
    energy = 0
    # we always have the cation-anion interactions,
    # and the charge product for these is always negative

    ca_energy = system['A_ca'] * np.sum(seps_to_minus12[charge_products < 0])
    energy += ca_energy

    if ref_charge > 0: #i.e. a cation
        # cation-cation interaction qq > 0
        # mask where qq < 0
        cc_energy = system['A_cc'] * np.sum(seps_to_minus12[charge_products>0])
```

```
            energy += cc_energy

        if ref_charge < 0: #i.e. an anion
            # anion-anion interaction qq > 0
            # mask where qq < 0
            aa_energy = system['A_cc'] * np.sum(seps_to_minus12[charge_products>0])
            energy += aa_energy

        return energy #in joules

    def coulomb_energy(separations, charge_products):
        # This is a fairly general way of calculating
        # the coulombic energy of the interactions between
        # two sets of point charges.
        # distances is a matrix such that d_ij is the
        # distance between atom i and atom j
        # similarly charge_products_ij is the product of
        # the charge on atom i with the charge of atom j
        # The total energy function defined above is producing a
        # 1xn matrix, but this works for a general 2d matrix
        # watch out for divide by zero errors though

        energy = np.sum(charge_products / separations)
        coulomb_constant = 1/(4*constants.pi*constants.epsilon_0)
        units = coulomb_constant*(constants.e**2)/constants.angstrom
        # Note we're assuming that charges are elemenetal charges
        # and distances are in angstroms
        if energy:
        energy *= units
        else:
        energy = 0
        return energy # In Joules

    def calc_lammps_coloumbic_energy(atoms, slip_system=build.NaCl_110_001):
        x_lo = np.min(atoms[:,0]) - 50
        x_hi = np.max(atoms[:,0]) + 50
        y_lo = np.min(atoms[:,1]) - 50
        y_hi = np.max(atoms[:,1]) + 50
        z_lo = np.min(atoms[:,2]) - slip_system['latt_params'][2]/4
        z_hi = np.max(atoms[:,2]) + slip_system['latt_params'][2]/4

        #build the data file for lamps.
        data_file_header = 'RPT26 on a LAMMPS adventure'
        data_file_header += '\n'
        data_file_header += '\n'
        data_file_header += '\n'
        data_file_header += '{:d} atoms\n\n'.format(len(atoms))
        data_file_header += (' 2 atom types\n')
        data_file_header += '{:} {:} xlo xhi\n'.format(x_lo, x_hi)
        data_file_header += '{:} {:} ylo yhi\n'.format(y_lo, y_hi)
        data_file_header += '{:} {:} zlo zhi\n'.format(z_lo, z_hi)
        data_file_header += '\n'
        data_file_header += '\n'
        data_file_header += 'Atoms\n'
        data_file_header += '\n'
        atoms_text = ''
        for i in range(len(atoms)):
            x = atoms[i,0]
            y = atoms[i,1]
            z = atoms[i,2]
            charge = atoms[i,-1]
            ID = i + 1
            if charge > 0:
                atom_type = 1
            elif charge < 0:
                atom_type = 2
            atoms_text += (' {:6} {: 2,g} '.format(ID, atom_type)
                       + '{: 3,g} {: 9,g} {: 9,g} {: 9,g}\n'.format(charge, x, y, z))

        data_file_text = data_file_header + atoms_text
        with open('data.electro', 'wt') as file:
            file.write(data_file_text)

        lmp = lammps()
        lmp.file('in.electro')
        electro_energy = lmp.get_thermo('pe')
        lmp.close()

        return electro_energy

    def calc_lammps_LJ_energy(atoms, slip_system=build.NaCl_110_001):
        x_lo = 2*np.min(atoms[:,0])
        x_hi = 2*np.max(atoms[:,0])
        y_lo = 2*np.min(atoms[:,1])
        y_hi = 2*np.max(atoms[:,1])
        z_lo = np.min(atoms[:,2]) - slip_system['latt_params'][2]/4
        z_hi = np.max(atoms[:,2]) + slip_system['latt_params'][2]/4

        #build the data file for lamps.
        data_file_header = 'RPT26 on a LAMMPS adventure'
        data_file_header += '\n'
        data_file_header += '\n'
        data_file_header += '\n'
        data_file_header += '{:d} atoms\n\n'.format(len(atoms))
        data_file_header += (' 2 atom types\n')
        data_file_header += '{:} {:} xlo xhi\n'.format(x_lo, x_hi)
```

```python
    data_file_header += '{:} {:} ylo yhi\n'.format(y_lo, y_hi)
    data_file_header += '{:} {:} zlo zhi\n'.format(z_lo, z_hi)
    data_file_header += '\n'
    data_file_header += '\n'
    data_file_header += 'Atoms\n'
    data_file_header += '\n'
    atoms_text = ''
    for i in range(len(atoms)):
        x = atoms[i,0]
        y = atoms[i,1]
        z = atoms[i,2]
        charge = atoms[i,-1]
        ID = i + 1
        if charge > 0:
            atom_type = 1
        elif charge < 0:
            atom_type = 2
        atoms_text += (' {:6} {: 2,g} '.format(ID, atom_type)
                    + '{: 3,g} {: 9,g} {: 9,g} {: 9,g}\n'.format(charge, x, y, z))

    data_file_text = data_file_header + atoms_text
    with open('data.LJ', 'wt') as file:
        file.write(data_file_text)

    lmp = lammps()
    lmp.file('in.LJ')

    LJ_energy = lmp.get_thermo('pe')
    lmp.close()

    return LJ_energy

def calc_lammps_energy(atoms, slip_system=build.NaCl_110_001):
    energy = calc_lammps_coloumbic_energy(atoms, slip_system=slip_system)
    energy += calc_lammps_LJ_energy(atoms, slip_system=slip_system)
    return energy
```