postqe Documentation

Release 0.1

M. Palumbo, D. Brunato, P. D. Delugas

August 30, 2017

Contents:

CHAPTER

ONE

INTRODUCTION

`postqe` is a Python package for postprocessing of results obtained with the Quantum Espresso (QE) code [1]. The package provides Python API functions for example for plotting the charge density (or the bare/Hartree/total potentials) on 1D or 2D sections, fitting the total energy with an equation of state (EOS) and other tasks. The package makes available in Python some QE functionalities using the F2PY code [2] and wrappers to generate Python modules from QE dynamically linked libraries. Finally, it also includes an interface with the popular Atomic Simulation Environment (*ASE*) [3], which in fact leverages for some functionalities.

It is meant to be imported in your own Python code or used from the command line (see the Tutorial part of this documentation). It is also meant for people who want to tinker with the code and adapt it to their own needs. The package is based on *numpy*, *scipy*, *matplotlib* and *ASE* libraries.

It is also meant as a software framework where Quantum Espresso developers or advanced users may implement new functionalities which are needed by the community. In this respect, it offers the possibility to develop code in different languages (Python, C/C++, Fortran) and then use Python to "glue" everything together.

Current features of the package include:

- Fit the total energy $E_{tot}(V)$ with an equation of state (Murnaghan, Vinet, Birch, etc.)
- Calculate and plot the electronic band structure
- Calculate and plot the electronic density of states (DOS)
- Plot 1D or 2D sections of the charge density
- Plot 1D or 2D sections of different potentials (Hartree, exchange-correlation, etc.)

Installation

You can download all package files from GitHub and then install it with the command:

```
sudo python setup.py install
```

The most useful functions for the user are directly accessible. You can import all of them as:

```
from postqe import *
```

or you can import only the ones you need. The above command also makes available a number of useful constants that you can use for unit conversions.

More functions are available as submodules. See the related documentation for more details. Note, however, that most of these functions are less well documented and are meant for advanced users or if you want to tinker with the code.

[1] http://www.quantum-espresso.org/
[2] https://docs.scipy.org/doc/numpy-dev/f2py/
[3] https://wiki.fysik.dtu.dk/ase/

General notes

Version

The package is still under development. Hence, it may contain bugs and some features may not be fully implemented. Use at your own risk.

Plotting

`postqe` uses the *matplotlib* library for Plotting. Some functions in the package are simply useful wrappers for *matplotlib* functionalities of common uses. They return a *matplotlib* object which can be further adapted to specific needs and personal taste. Alternatively, you can of course manipulate and plot the post-procecessed data with any other Python tool of your choice.

CHAPTER

TWO

TUTORIAL

This is a simple tutorial demonstrating the main functionalities of `postqe`. The examples below show how to use the package to perform the most common tasks. The code examples can be found in the directory *examples* of the package and can be run either as interactive sessions in your Python intepreter or as scripts. The tutorial is based on the following examples:

Several simplified plotting functions are available in `postqe` and are used in the following tutorial to show what you can plot. Note however that all plotting functions need the matplotlib library, which must be available on your system and can be used to further tailor your plot.

Fitting the total energy using Murnaghan EOS (examples 1)

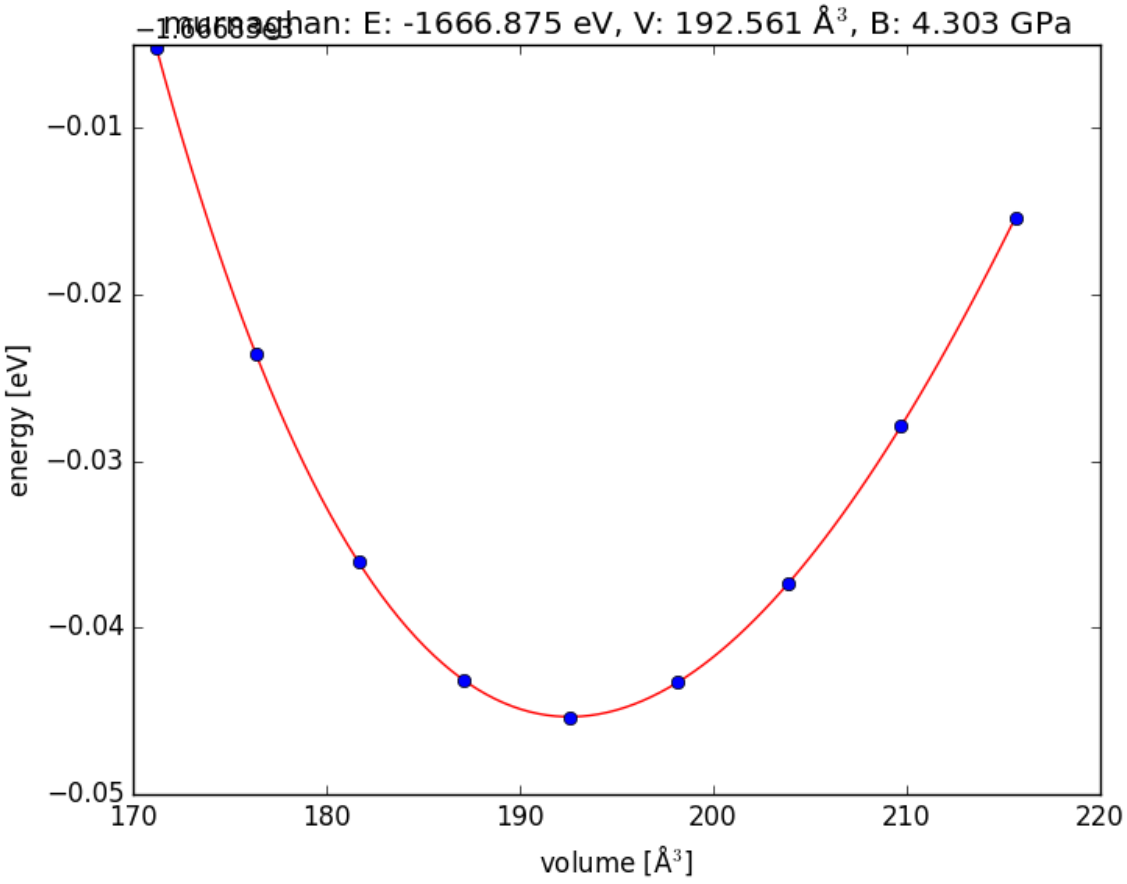
The simplest task you can do with `postqe` is to fit the total energy as a function of volume $E_{tot}(V)$. You can use an equation of state (EOS) such as Murnaghan's or similar. Currently you can use Murnaghan, Vinet, Birch, Birch-Murnaghan, Pourier-Tarantola and Anton-schmidt EOS and 2nd and 3rd order polynomials in `postqe`.

Let's see how to fit $E_{tot}(V)$. This is the case of isotropic cubic systems (simple cubic, body centered cubic, face centered cubic) or systems which can be approximated as isotropic (for example an hexagonal system with nearly constant c/a ratio).

```
from postqe import units, get_eos

eos = get_eos(label="./Nienergies.dat", eos='murnaghan')
v0, e0, B = eos.fit()
# Print some data and plot
print('Equilibrium volume = '+str(v0)+' Ang^3')
print('Equilibrium energy = '+str(e0)+' eV')
print('Equilibrium Bulk modulus = '+str(B / units.kJ * 1.0e24)+' GPa')
fig = eos.plot('Ni-eos.png', show=True)

# Save the plot in a different format (pdf) with Matplotlib if you like
fig.figure.savefig("figure.pdf", format='pdf')
```

The `get_eos()` needs in input a file with two columns: the first with the volumes (in $a.u.^3$), the second with energies (in $Ryd/cell$). You also define here what EOS to use, in this case Murnaghan's. This function returns an *eos* object. The method `fit()` performs the fitting and returns the equilibrium volume *v0*, energy *e0* and bulk modulus *B*. The fitting results can then be printed or further processed.

Optionally, you can plot the results with the `plot_EV()`. The original data are represented as points.

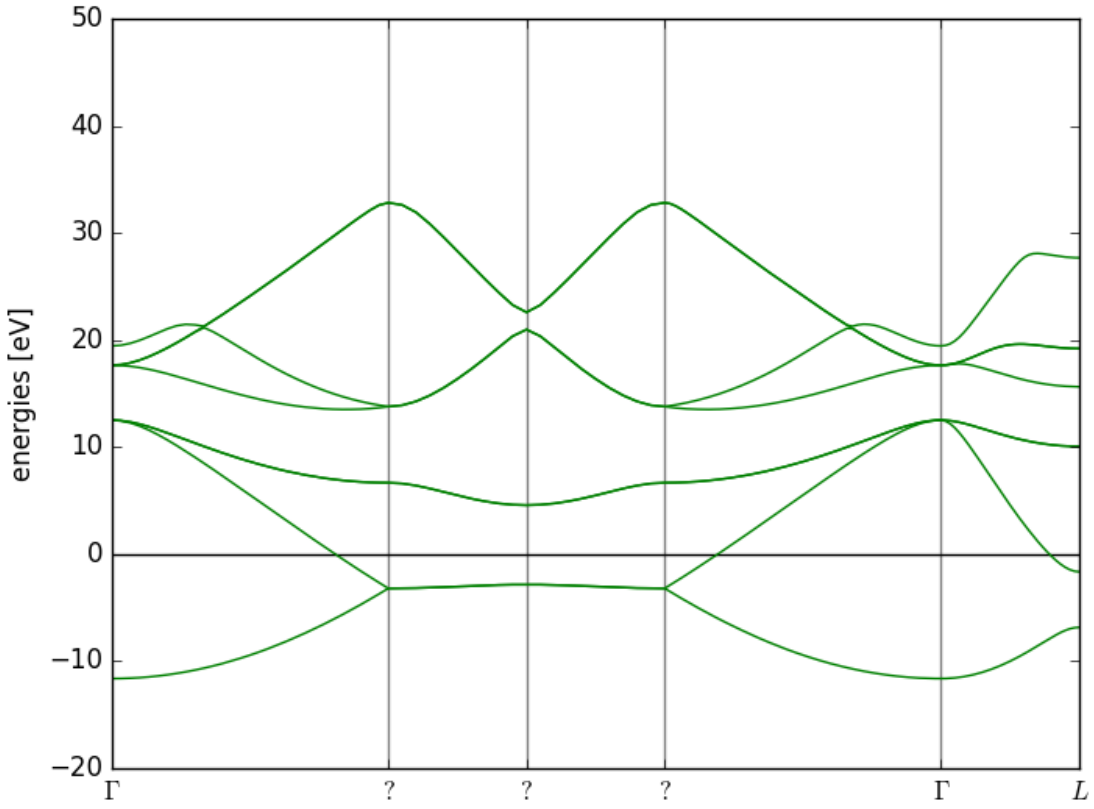


Calculate and plot the band structure of silicon (examples 2)

This example shows how to calculate the electronic band structure of silicon with `postqe`. All necessary information is extracted from the standard xml output file of Quantum Espresso, produced by a proper calculation along the wanted path in the Brillouin zone.

```
bs = get_band_structure(label="./Si", schema='../../schemas/qes.xsd', reference_energy=0)
fig = bs.plot(emin=-20, emax=50, show=True, filename='Sibands.png')
```

The `get_band_structure()` needs a parameter *label* which identies the system and the corresponding xml file (label.xml). *label* may contain the full path to the file. The *schema* (optional) parameter allows the code to properly parse and validate the xml file. The parameter *reference_energy* (usually the Fermi level) allows you to shift the plot accordingly. `get_band_structure()` returns a band structure object which can be further processed. For example, the method `plot()` creates a figure and save it in a png file.



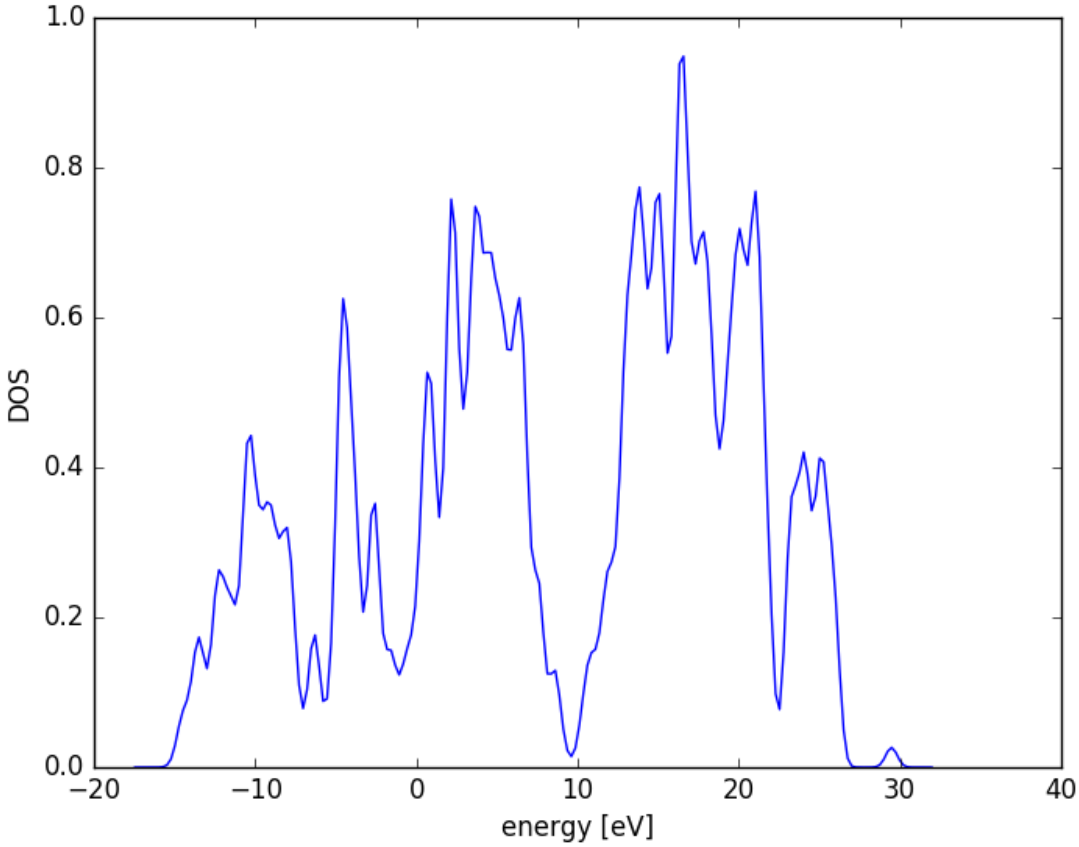
Calculate and plot the density of states (DOS) of silicon (examples 3)

This example shows how to calculate the electronic density of states (DOS) with `postqe`. All necessary information is extracted from the standard xml output file. The following code shows how to do it for silicon (xml output file: Si.xml)

```
from postqe import get_dos

# get a DOS object
dos = get_dos(label="./Si", schema='../../schemas/qes.xsd', width=0.5, npts=200)

# get the dos and energies for further processing
d = dos.get_dos()
e = dos.get_energies()

# Plot the DOS with Matplotlib...
import matplotlib.pyplot as plt
plt.plot(e, d)
plt.xlabel('energy [eV]')
plt.ylabel('DOS')
plt.savefig("figure.png")
plt.show()
```

The `get_dos()` needs in input the xml file produced by pw.x, after a proper DOS calculation. This is identified using *label* which may contain the full path to the file (.xml is automatically added). The *schema* (optional) parameter allows the code to properly parse and validate the xml file. You must also specify the number of energy steps in the DOS (*npts*), plus the Gaussian broadening (*width*). `get_dos()` then returns a DOS object.

The DOS values and the corresponding energies can be obtained from the DOS object using the methods `get_dos()` and `get_energies()`. If you want you can further manipulate these values. For example you can make a plot with the Python library *Matplotlib*. The output plot looks like the following:



You can of course continue to calculate other quantities in your script.

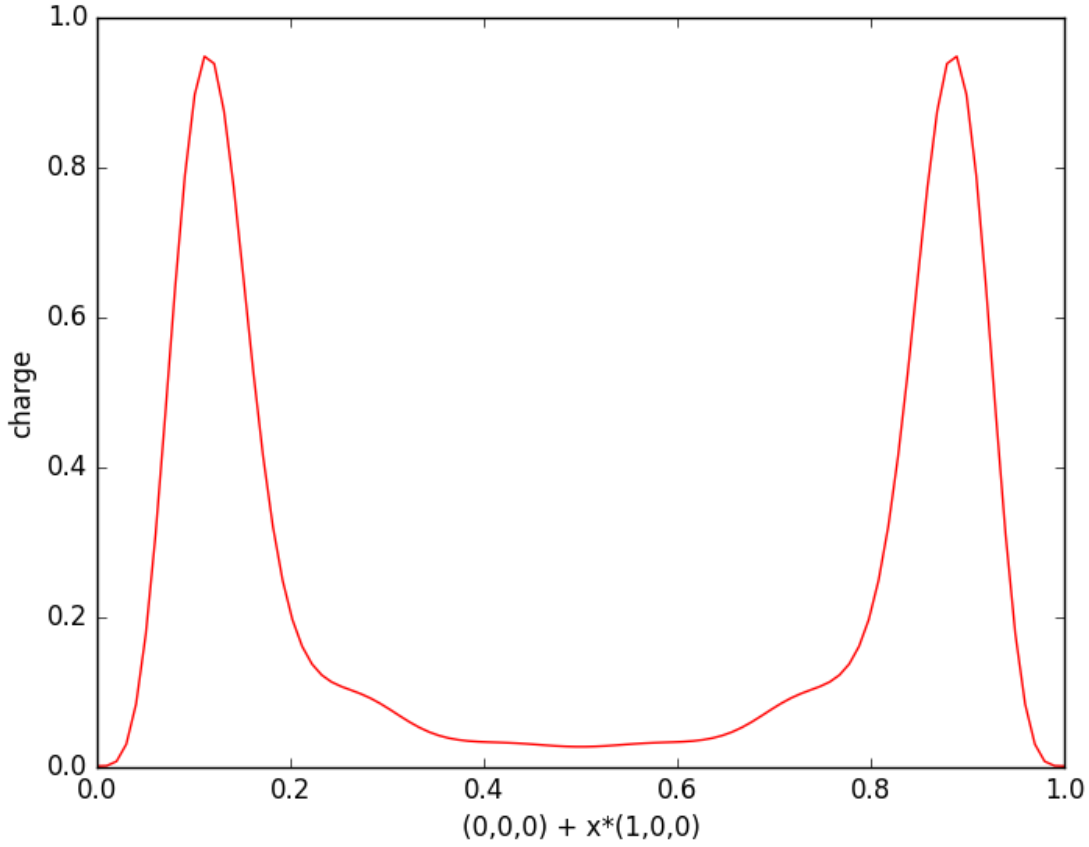
Plotting a 1D section of the charge density (examples 4)

A common task you can do with `postqe` is to plot the electronic charge density along one direction. The charge is read from the HDF5 output file create by the Quantum Espresso calculation in *outdir*. Additional information are extracted from the standard xml output file, identified by the 'label' parameter in the `get_charge()` function. The *schema* (optional) parameter allows the code to properly parse and validate the xml file. The full code to do this is shown below:

```
from postqe import get_charge

charge = get_charge(label="./Ni", schema='../../schemas/qes.xsd')
```

```
charge.write('outputcharge.dat')

figure = charge.plot(x0=(0, 0, 0), e1=(1, 0, 0), nx=100)
figure.savefig("figure_1.png")
```

The call to `get_charge()` creates a *charge* object. The charge can be written in a text file using the method `write()`. The call to the method `plot()` returns a *Matplotlib* figure object containing a 1D section plot of the charge from the point *x0* along the direction *e1*. By default, the charge is plotted from the point (0,0,0) along the direction (1,0,0). The *Matplotlib* figure object can be further modified with the standard methods of this library. For example, the plot can be save in a png file using the method `savefig()`. The result is shown below.



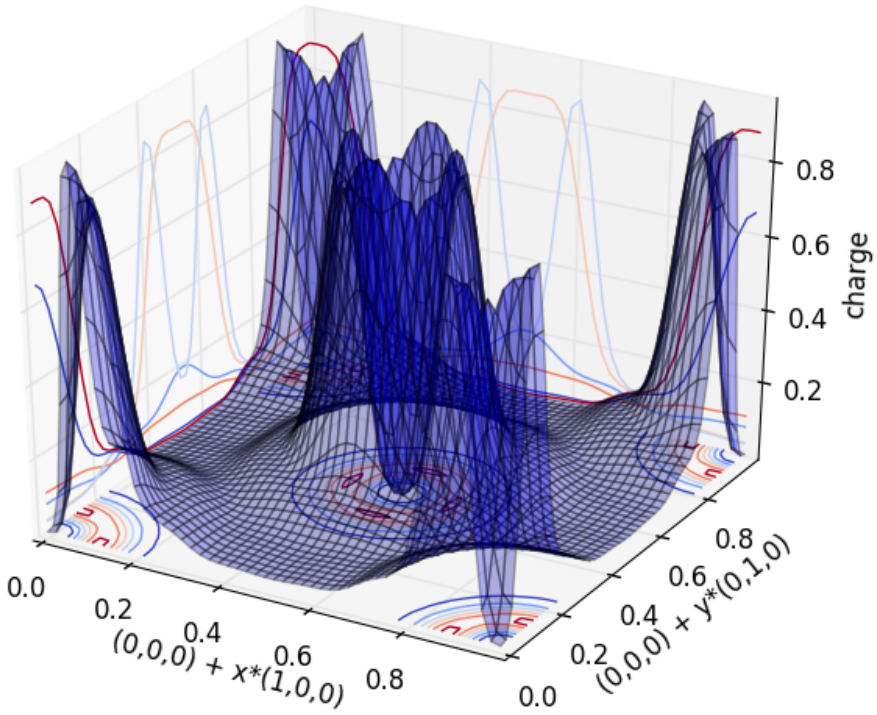
Plotting a 2D section of the charge density (examples 5)

This example is similar to the previous one except for producing a 2D plot of a planar section of the electronic charge density. The plane is defined by an initial point *x0* and two vectors, *e1* and *e2* which define the plane, which are given as parameters to the method `plot()` together with *dim=2* to define a 2D plot.

```
from postqe import get_charge

charge = get_charge(label="./Ni", schema='../../schemas/qes.xsd')

figure = charge.plot(x0=(0, 0, 0), e1=(1, 0, 0), e2=(0, 1, 0), nx=50, ny=50, dim=2)
figure.savefig("figure_1.png")
```

```
figure.savefig("figure_1.pdf", format='pdf')
```

The resulting *Matplotlib* plot is



Plotting 1D sections of different potentials (examples 6)

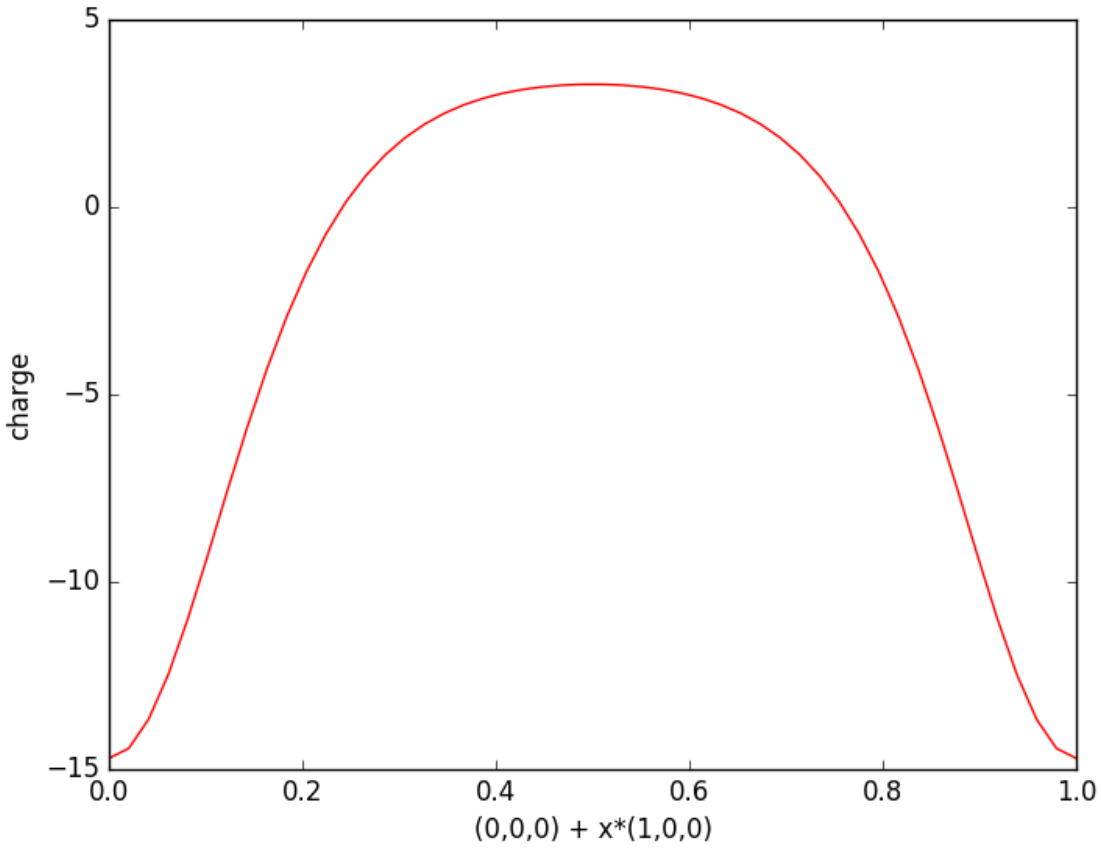
This example computes all the different potentials available, i.e. the bare potential V_{bare}, the Hartree potential V_H, the exchange-correlation potential V_{xc} and the total potential $V_{tot} = V_{bare} + V_H + V_{xc}$. All necessary information is taken from the xml output file of QE and the HDF5 charge file. The pseudopotential file is also necessary to compute V_{bare}.

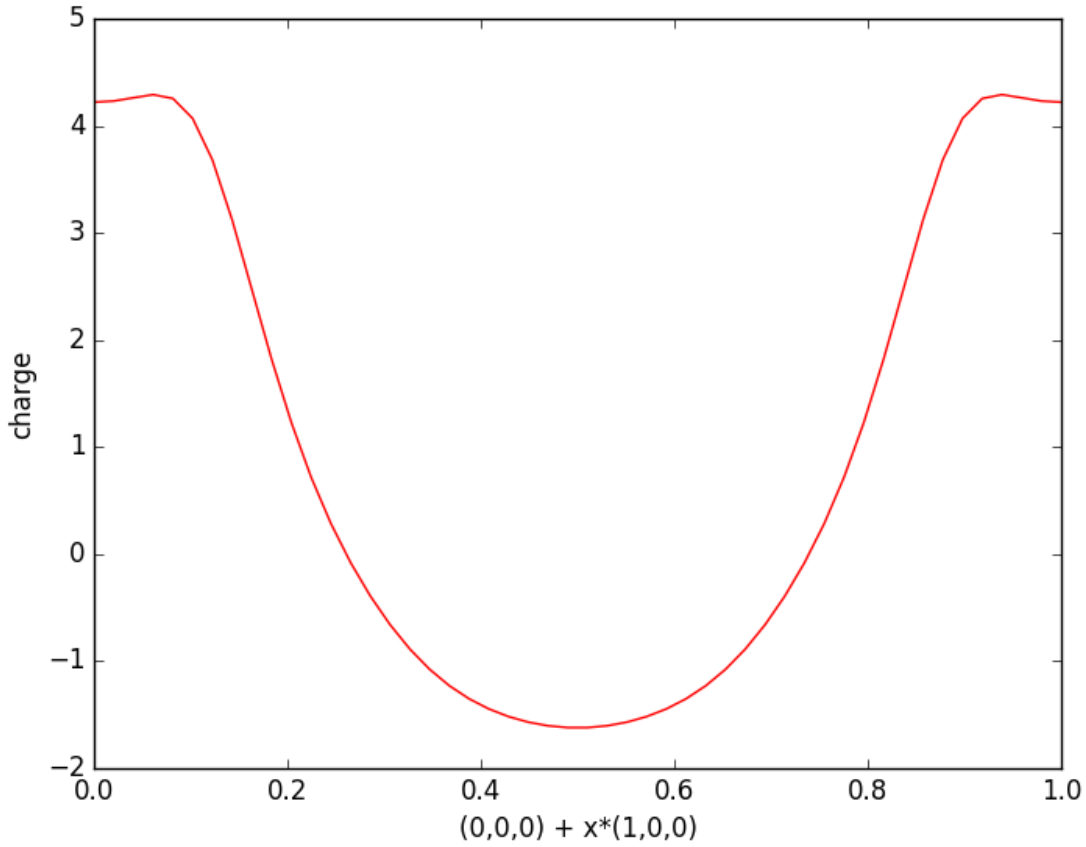
```
from postqe import get_potential

v_bare = get_potential(label="./Ni", schema='../../schemas/qes.xsd', pot_type='v_bare')
v_bare.write('v_bare.dat')
fig1 = v_bare.plot(x0=(0, 0, 0), e1=(1, 0, 0), nx=50)
fig1.savefig("figure_v_bare.png")

v_h = get_potential(label="./Ni", schema='../../schemas/qes.xsd', pot_type='v_h')
v_h.write('v_h.dat')
fig2 = v_h.plot(x0=(0, 0, 0), e1=(1, 0, 0), nx=50)
fig2.savefig("figure_v_h.png")

v_xc = get_potential(label="./Ni", schema='../../schemas/qes.xsd', pot_type='v_xc')
```

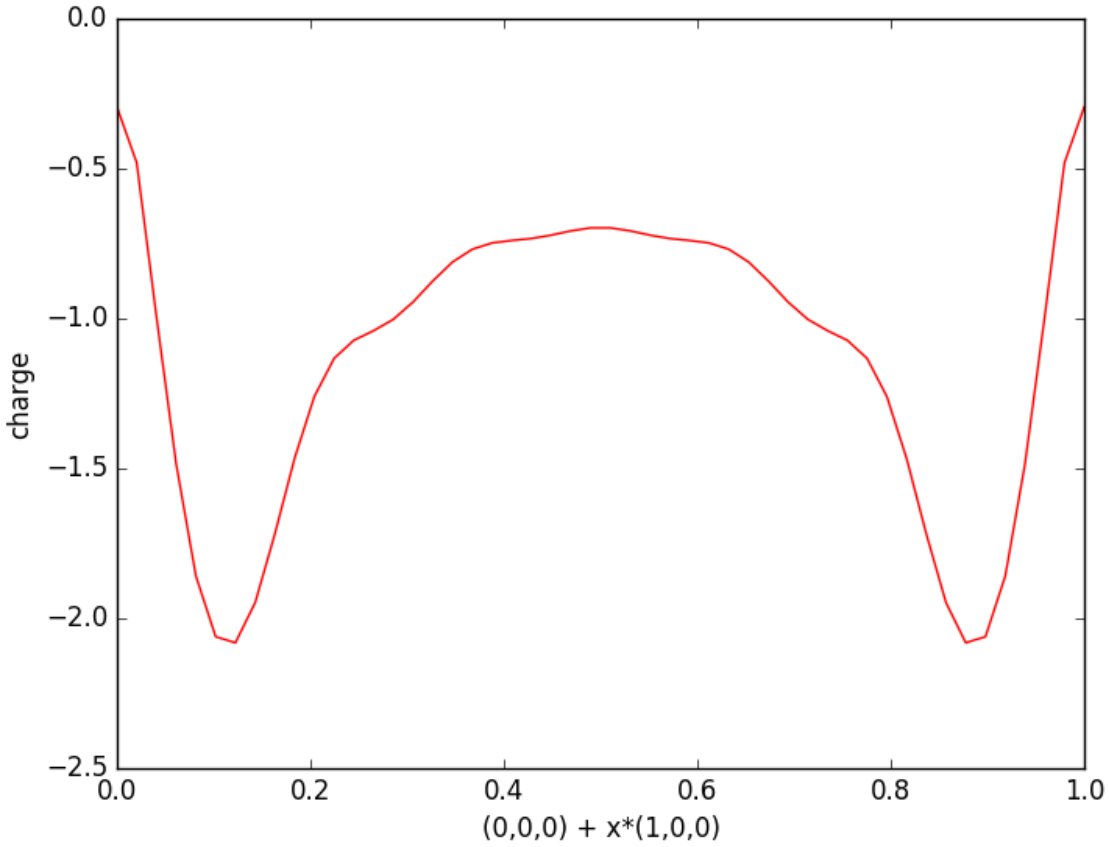
```
v_xc.write('v_xc.dat')
fig3 = v_xc.plot(x0=(0, 0, 0), e1=(1, 0, 0), nx=50)
fig3.savefig("figure_v_xc.png")

v_tot = get_potential(label="./Ni", schema='../../schemas/qes.xsd', pot_type='v_tot')
v_tot.write('v_tot.dat')
fig4 = v_tot.plot(x0=(0, 0, 0), e1=(1, 0, 0), nx=50)
fig4.savefig("figure_v_tot.png")
```

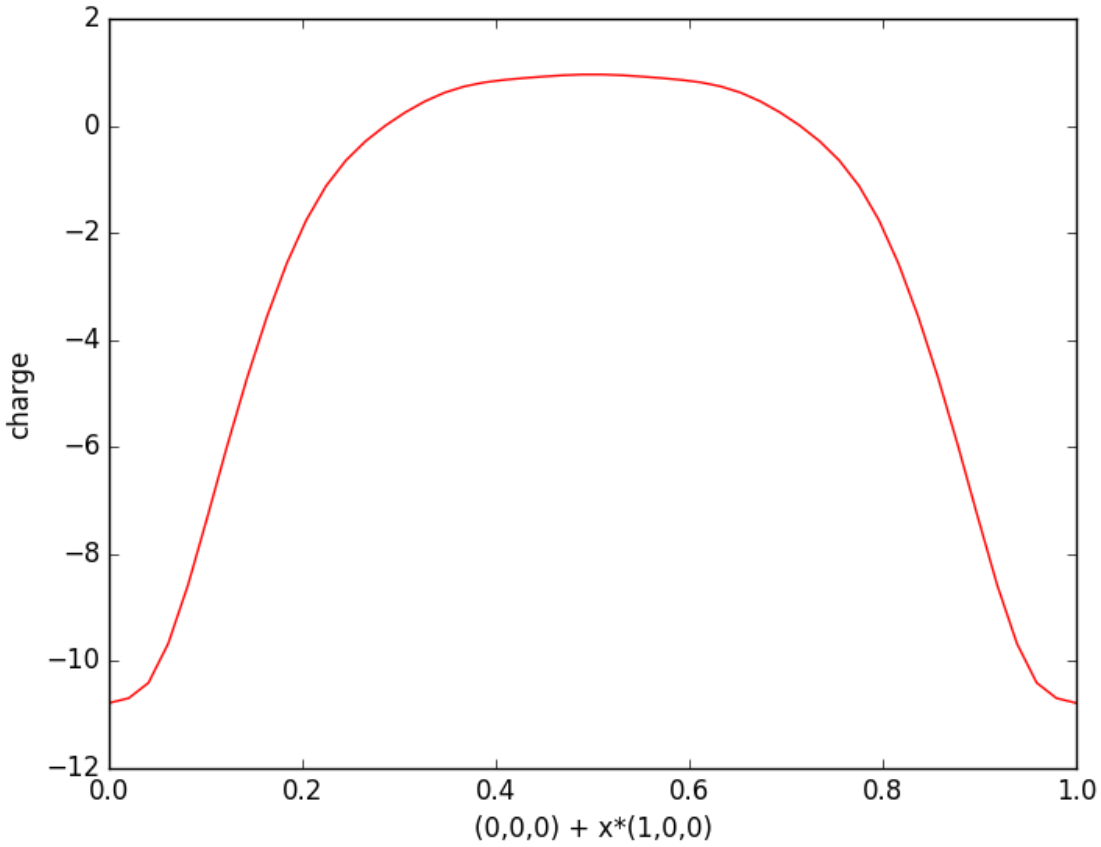
The code essentialy call the function `get_potential()`, which returns a potential object of the type defined in *pot_typ*. The `write()` and `plot()` methods are then used to write the output in a text file and produce the plots as in the previous example (in fact they accept the same parameters).

The output figures are as follows:




5
4
3
2
1
0
−1
−2
charge
0.0
0.2
0.4
0.6
0.8
1.0
(0,0,0) + x*(1,0,0)


0.0
−0.5
−1.0
−1.5
−2.0
−2.5
charge
0.0
0.2
0.4
0.6
0.8
1.0
(0,0,0) + x*(1,0,0)


2
0
−2
−4
−6
−8
−10
−12
charge
0.0
0.2
0.4
0.6
0.8
1.0
(0,0,0) + x*(1,0,0)

CHAPTER

THREE

POSTQE PACKAGE

Submodules

Additional functions are available as submodules. Please note the documentation of these functions is still ongoing and can be incomplete or wrong.

postqe.constants module

postqe.eos module

postqe.plot module

postqe.readutils module

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search