NB: The graded, first version of the report must be returned if you hand in a second time!

HA2: Shape of aluminium nanoparticles

Albin Annér and David Kastö

February 7, 2018

Task №	Points	Avail. points
$\sum$		

Introduction

The properties of nanoparticles differ immensely from the equivalents in bulk materials. However, the properties are not only dependent on the size of the particles; their shape turns out to play an important role as well. Usually the shape can experimentally be controlled by the adjustment of various parameters during the synthesis. In that way non-equilibrated (from a thermodynamic point of view) particles can be made by e.g. applying stabilizing organic molecules onto the surface. However, to predict the thermodynamic shape (corresponding to the one with the lowest energy for a given number of particles) the so-called Wulff construction may be used. In this construction the surface energies of the nanoparticle's various facets are essential[1]. This study computationally investigates such Wulff constructions for large aluminium nanoparticles by applying density functional theory (DFT), with a restriction to the (111) and the (100) facets. The goal is to calculate the most stable shape of these nanoparticles and examine how external conditions (such as molecules in gas phase) affect the shape.

Computational method

DFT is in in principle exact. However, the exact form of the exchange-correlation potential is not known and hence various approximations are made to solve specific problems. The approximations can become increasingly intricate and elaborate, often with accompanying increase in the computational complexity and cost. A commonly used approximation is to treat the exchange-correlation potential as a second degree Taylor potential in the electron density. !! This violates properties of the XC potential which are known to be true. How and why can we use this then!!

There are many different GGA functionals an no single one is optimal for all applications. For metals, the PBE, or simply the LSDA functional will capture the relvant physical properties of the system as the density does not vary rapidly [].

Atop of xc-functionals, an appropriate basis set needs to be specified for the problem at hand. Commonly used basis sets are linnear combination of atomic oribitals (LCAO), a set of plane waves, and real space grids. A difficulty with LCAO is that one need sufficient information about the system at hand to make an educated guess about the atomci orbitals. However, the computational process is easy to parallelize and is generally easily computed since the solution for the orbitals can be extrapolated in space. The real space method is generally computationally expensive (however often exact). Plane waves have neat physical properties, such as that the energy is only dependent on the wave-vector and that solutions become more exact when one increases the number of plane waves. A problem with the plane waves basis set is that it utalizes fourier transforms which are generally tedious to parallellize. We did not investigate the atomic atorbitals of aluminum enough to make an educated guess and we tried to avoid unreasonably heavy computational burden. On top of this, we really enjoyed that the plane waves converges to an accurate solution as the basis set expanded. Thus, we decided to use the plane wave basis set in the simulation of the aluminum nano particles.

Another thing to consider is the discrete k spacing for the energy bands. The k-space domain need too be sampled at fine enough distances to capture most parts of the band structure. If k values are sampled too far apart, the energy can differ from the actual energy, both an increase and a decrease. To get a sufficient amount of k-points is often not the most computationally heavy step and therefore can be exaggerated a bit. To find a reasonable value for the spacing, simulations were made with plane wave sets with increasing energy, and the spacing for which the total energy had stabilized was used in further simulations.

To get a good estimate of the surface energy one has to include atoms of the bulk since they effect the interatomic distance near the surface, and hence the surface energy. The surface approaches an stable energy at about XXXX atoms, see figure XXXXX.

The simulation was made both in vacuum and with CO molecules attached to the surface. The CO molecules was attached on top of aluminium atoms in both the 111 and 100 surface. The occupacy of atoms was set to 1/4 by adsorbing one CO molecule to a 2x2 surface. Is it ok to do it this way?

When updating the density in the self consistensy loop, it is useful to keep some information about the previous density when navigating in the energy landscape. (The information about the direction towards the minimal energy should not be lost in each iteration step). This is implmented with a so called mixer. The new density is usually updated according to the following rule:

$$n^{new} = \beta n^{old} + (1 - \beta) n^{new} \tag{1}$$

For metals, the recommended value for β is about 0.1 **??**. GPAW has a build in mixer, which takes in three arguments, β, number of previous generations used 'nmaxold' and a weight. For the two latter parameters, the default value was used, 5 generations weight 50.

Results

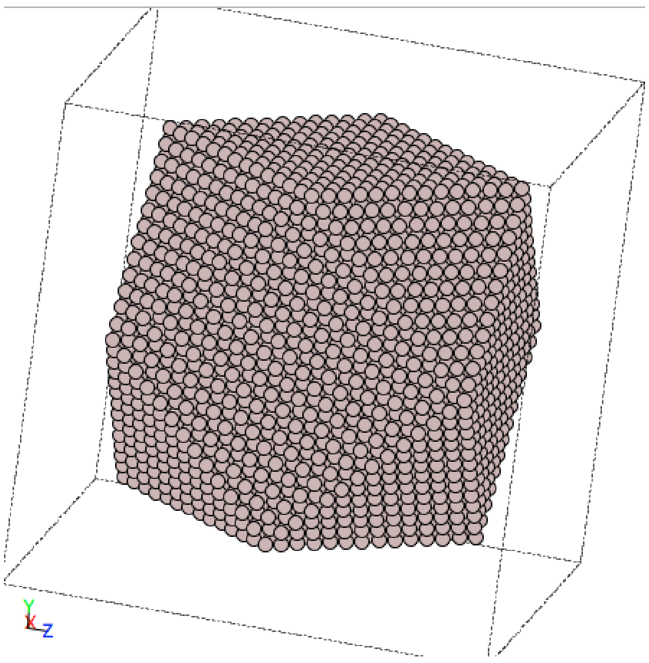

Figure 1: Wulff construction of aluminum in vacuum

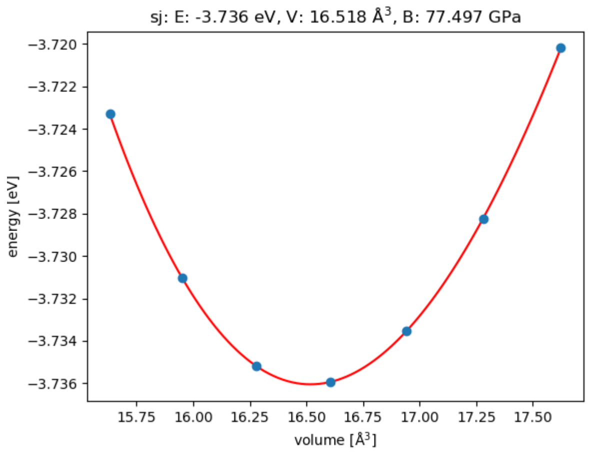

Figure 2: Total lattice energy for varying lattice spacings

Different shapes of wulf construction

Discussion

(Try to bring order in your results, rationalize the results to the best of your ability)

Conclution

References

[1] Magnus Rahm, *Task and background*. Pingpong - TIF035 Computational Materials Physics VT18, 2018.
`https://pingpong.chalmers.se/courseId/9094/content.do?id=4260146`. Retrieved 2018-02-02.

A Source Code

A.1 Main simulation: /HA2_simulation.py

```python
#!/usr/bin/env python
# coding=utf-8

# # # imports # # #
import numpy as np
from gpaw import GPAW, Mixer, PW
from ase.build import *
from ase.io import read
from ase.units import kJ, J, m
from ase.eos import EquationOfState
from ase.build import fcc111, fcc100, add_adsorbate
from ase.cluster.wulff import wulff_construction
from ase.visualize import view
from ase.parallel import rank
from ase import Atoms

# # # Constants and parameters # # #
a_al = 4.05  # Lattice constant for Al, experimentally determined
N_lattice_spacings = 7  # Number of lattice constants to loop over to find equilibrium
E_al = 84.67567  # Ionization energy for hardest bound core electron
theta = 1 / 4.0  # Monolayer coverage

# # # Create Al bulk and initialize calculator parameters # # #
al = bulk('Al', 'fcc', a=a_al, cubic=False)  # Create Al bulk
mixer = Mixer(beta=0.1, nmaxold=5,  weight=50.0)  # Recommended values for small systems
E_cut = [50, E_al, 100, 200, 300, 400, 500]  # cut-off energy

for energy in [500]:  # Change to E_cut to loop and check convergence
    calc = GPAW(mode=PW(energy),  # use the LCAO basis mode
                h=0.18,  # grid spacing, recommended value in this course
                xc='PBE',  # Exchange-correlation functional
                mixer=mixer,
                kpts=(12, 12, 12),  # k-point grid - LOOP OVER LATER TO CHECK "CONVERGENCE"
                txt='out.txt')  # name of GPAW output text file
    al.set_calculator(calc)

    # # # Find lattice constant with lowest energy # # #
    cell_0 = al.cell  # Unit cell object of the Al bulk
    for eps in np.linspace(-0.02, 0.02, N_lattice_spacings):
        al.cell = (1 + eps) * cell_0  # Adjust lattice constant of unit cell
        al.get_potential_energy()  # Calculate the potential energy for the Al bulk

    confs = read('out.txt@0:' + str(N_lattice_spacings))  # Read the c o n gurations

    # Extract volumes and energies:
    volumes = [atoms.get_volume() for atoms in confs]
    energies = [atoms.get_potential_energy() for atoms in confs]
    # if rank == 0:
    #     print energies, shape(energies)

    # Plot energies as a function of unit cell volume (directly related to latt. const.)
    eos = EquationOfState(volumes, energies)
    v0, E_bulk, B = eos.fit()
    eos.plot('Al_eos.png')
    a_calc = (4 * v0)**(1 / 3.0)  # Latt. const. acc. to ASE doc., but why is this correct?

    N_x = 1
    N_y = 1
    N_z = 6

    surface111 = fcc111('Al', size=(N_x, N_y, N_z), a=a_calc, vacuum=7.5)
    surface100 = fcc100('Al', size=(N_x, N_y, N_z), a=a_calc, vacuum=7.5)
    surface111.center(axis=2)
    surface100.center(axis=2)

    # Initialize new calculator that only considers k-space in xy-plane,
    # since we're only looking at the surface
    calc2 = GPAW(mode=PW(energy),  # use the LCAO basis mode
                 h=0.18,  # grid spacing
                 xc='PBE',  # XC-functional
                 mixer=mixer,
                 kpts=(12, 12, 1),  # k-point grid
                 txt='out2.txt')  # name of GPAW output text file

    surface111.set_calculator(calc2)
    surface100.set_calculator(calc2)

    cell111 = surface111.get_cell()  # Unit cell object of the Al FCC 111
    area111 = np.linalg.norm(np.cross(cell111[0], cell111[1]))  # Calc. surface area
    surfEn111 = surface111.get_potential_energy()  # Calc pot. energy of FCC 111
    cell100 = surface100.get_cell()  # Unit cell object of the Al FCC 100
    area100 = np.linalg.norm(np.cross(cell100[0], cell100[1]))  # Calc. surface area
    surfEn100 = surface100.get_potential_energy()  # Calc pot. energy of FCC 100

    # Calc. surf. energy per area (sigma) for FCC 111 and 100
    sigma111 = (1 / (2.0 * area111)) * (surfEn111 - N_x * N_y * E_bulk)
    sigma100 = (1 / (2.0 * area100)) * (surfEn100 - N_x * N_y * E_bulk)

    # # # Add CO adsorbate to Al surface # # #
    d_CO = 1.128  # CO bondlength in [  ]

    CO = Atoms('CO')  # Create CO molecule object
    add_adsorbate(slab=surface111, adsorbate=CO, height=4.5, position='ontop')
    add_adsorbate(slab=surface100, adsorbate=CO, height=4.5, position='ontop')
    # height above based on values for CO in ASE doc. Future: We could also
    # perform equilibrium scan by looping over various heights

    CO.set_cell([10, 10, 10])
    CO.center()
    CO.set_calculator(calc)

    energy_CO = CO.get_potential_energy()

    surfEn111_ads = surface111.get_potential_energy()
    surfEn100_ads = surface100.get_potential_energy()

    sigma100_ads = sigma100 + theta * (surfEn100_ads - surfEn100 - energy_CO) / area100
    sigma111_ads = sigma111 + theta * (surfEn111_ads - surfEn111 - energy_CO) / area111

    file = open('sigma_Al.txt', 'w')
    file.write(str(sigma111) + '\t' + str(sigma100))
    file.close()

    file = open('sigma_ads.txt', 'w')
    file.write(str(sigma111_ads) + '\t' + str(sigma100_ads))
    file.close()

    if rank == 0:
        a = 0
        print area111, surfEn111, E_bulk, sigma111
```

A.2 Data processing and wulff construction: `HA2_data_processing.py`

```python

#from ase.io import read
from ase.build import fcc111, fcc100
from ase.cluster.wulff import wulff_construction
from ase.visualize import view
import matplotlib.pyplot as plt
import numpy as np

# input files
filename1 = 'sigmas.txt'
filename2 = 'sigmas_ads.txt'

# import and manage data
sigmas = np.array(np.loadtxt(filename1))
sigma111 = sigmas[0]
sigma100 = sigmas[1]
sigmas_ads = np.array(np.loadtxt(filename2))
sigma111_ads = sigmas_ads[0]
sigma100_ads = sigmas_ads[1]
theta = 1 / 4  # Monolayer coverage
print(sigma111, sigma100)
print(sigma111_ads, sigma100_ads)

Al = wulff_construction('Al',
                        surfaces=[(1, 0, 0),
                                  (1, 1, 1)],
                        energies=[sigma100, sigma111],
                        size=10000,
                        structure='fcc',
                        rounding='below')  # What does this one do?
Al.center(vacuum=10)
view(Al)

sigma100_ads = sigma100 + theta * sigma100_ads
sigma111_ads = sigma111 + theta * sigma111_ads
```

```python

Al_ads = wulff_construction('Al',
                            surfaces=[(1, 0, 0),
                                      (1, 1, 1)],
                            energies=[sigma100_ads, sigma111_ads],
                            size=10000,
                            structure='fcc',
                            rounding='below')  # What does this one do?
Al_ads.center(vacuum=10)
view(Al_ads)
```