DFT Jupyter Notebook

Release 0.1

Apr 26, 2020

Contents

CHAPTER 1

Python

1.1 Classes

1.1.1 1. Classes and Instances

[13]:
```python
class Employee:
    pass

emp_1 = Employee()

emp_1.first = "Corey"
print(emp_1.first)
```

```
Corey
```

[28]:
```python
class Employee:

    raise_amount = 1.4

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay

    def fullname(self):
        return "{} {}".format(self.first, self.last)

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)

    @classmethod
    def self_raise_amt(cls, amount):  # Using cls is a convention
        cls.raise_amount = amount
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def is_workway(day):
        if day.weekday() == 5:
            return False
        else:
            return True

emp_1 = Employee("Corey", "Schafer", 1000)


emp_1.first = "Corey"
print(emp_1.first)
print(emp_1.fullname())
print(Employee.fullname(emp_1))

# class varialbes
print(emp_1.apply_raise())
print(emp_1.pay)

# classmethods
Employee.self_raise_amt(1.5)
print(Employee.raise_amount)
print(emp_1.raise_amount)

# static methods
```

```
Corey
Corey Schafer
Corey Schafer
None
1400
1.5
1.5
```

1.1.2 2. Inheritance

[36]:
```
class Employee:

    raise_amount = 1.4

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay

    def fullname(self):
        return "{} {}".format(self.first, self.last)

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)

class Develop(Employee):
    raise_amount = 1.4
```

(continues on next page)

(continued from previous page)

```

dev_1 = Develop("Corey", "Schafer", 1000)
# print(help(Develop))
print(dev_1.pay)
print(Develop.raise_amount)
```

```
1000
1.4
```

1.1.3 3. Property Decorators - Getters, Setters, and Deleters

[ ]:

```

```

[ ]:

```
## Pro
```

[46]:

```
class Employee:

    raise_amount = 1.4

    def __init__(self, first, last, pay):
        self.first = first
        self.last = last
        self.pay = pay

    @property
    def fullname(self):
        return "{} {}".format(self.first, self.last)

    @fullname.setter
    def fullname(self, name):
        first, last = name.split()
        self.first = first
        self.last = last

emp_1 = Employee("John", "Smith", 100)
print(emp_1.fullname)

emp_1.fullname = "Corey Schafer"
print(emp_1.fullname)
```

```
John Smith
Corey Schafer
```

1.2 Conda

1.2.1 1. Using conda-forge

conda-forge

```
conda config --add channels conda-forge
conda config --set channel_priority strict
conda install <package-name>
```

or edit the ~/.condarc directly.

```
channels:
  - pytorch
  - conda-forge
  - defaults
channel_priority: strict
```

1.2.2 2. Conda cheat sheet

conda cheet sheet

1.3 Snippets

```
[2]: print(list(map(lambda x: x + 2, [1, 2, 3])))
```

```
[3, 4, 5]
```

```
[4]: print(list(filter(lambda x: x <= 2, [1, 2, 3])))
```

```
[1, 2]
```

```
[ ]:
```

CHAPTER 2

Package

2.1 Graphviz

2.1.1 1. The definition of workflow diagram

Definition

Common symbols and shapes

2.1.2 2. The definition of DOT language

https://www.graphviz.org/doc/info/lang.html https://en.wikipedia.org/wiki/DOT_(graph_description_language)

2.1.3 3. Graphviz

Graphviz Docs

example

[1]:
```
from graphviz import Digraph

dot = Digraph(
    name="lucidchart",
    graph_attr={"rankdir": "UD"},
    node_attr={"color": "lightblue", "style": "filled"},
)

dot.attr("node", shape="oval")
dot.node("Purchase Order Received")
dot.node("Submit to controller for Approval")
dot.node("Process Order")
```

(continues on next page)

(continued from previous page)

```
dot.attr("node", shape="box")
dot.node("Process New Customer Record")
dot.node("Input Order into System")
dot.node("Input Order")

dot.attr("node", shape="diamond")
dot.node("Current Customer?")
dot.node("Customer from US")

dot.edge("Purchase Order Received", "Current Customer?")
dot.edge("Current Customer?", "Customer from US", label="No")
dot.edge("Current Customer?", "Input Order into System", label="Yes")
dot.edge("Input Order into System", "Process Order")
dot.edge("Customer from US", "Process New Customer Record", label="Yes")
dot.edge("Customer from US", "Submit to controller for Approval", label="No")
dot.edge("Process New Customer Record", "Input Order")
dot.edge("Input Order", "Process Order")

dot.view(cleanup=True)
```

```
---------------------------------------------------------------------------
ModuleNotFoundError                       Traceback (most recent call last)
<ipython-input-1-d8ad43778d4f> in <module>
----> 1 from graphviz import Digraph
      2 
      3 dot = Digraph(
      4     name="lucidchart",
      5     graph_attr={"rankdir": "UD"},

ModuleNotFoundError: No module named 'graphviz'
```

[ ]:

CHAPTER 3

DFT

3.1 1 Introduction to this book

This book serves two purposes: 1) to provide worked examples of using DFT to model materials prop- erties, and 2) to provide references to more advanced treatments of these topics in the literature. It is not a definitive reference on density functional theory. Along the way to learning how to perform the calculations, you will learn how to analyze the data, make plots, and how to interpret the results. This book is very much “recipe” oriented, with the intention of giving you enough information and knowledge to start your research. In that sense, many of the computations are not publication quality with respect to convergence of calculation parameters.

You will read a lot of python code in this book. I believe that computational work should always be scripted. Scripting provides a written record of everything you have done, making it more probable you (or others) could reproduce your results or report the method of its execution exactly at a later time.

3.2 2 Introduction to DFT

A comprehensive overview of DFT is beyond the scope of this book, as excellent reviews on these subjects are readily found in the literature, and are suggested reading in the following paragraph. Instead, this chapter is intended to provide a useful starting point for a non-expert to begin learning about and using DFT in the manner used in this book. Much of the information presented here is standard knowledge among experts, but a consequence of this is that it is rarely discussed in current papers in the literature. A secondary goal of this chapter is to provide new users with a path through the extensive literature available and to point out potential difficulties and pitfalls in these calculations.

A modern and practical introduction to density functional theory can be found in Sholl and Steckel (*Density Functional Theory: A Practical Introduction*). A fairly standard textbook on DFT is the one written by Parr and Yang (*Density-Functional Theory of Atoms and Molecules*). The Chemist’s Guide to DFT (*A Chemist’s Guide to Density Functional Theory*) is more readable and contains more practical information for running calculations, but both of these books focus on molecular systems. The standard texts in solid state physics are by Kittel (*Introduction to Solid State Physics*) and Ashcroft and Mermin (*Solid State Physics*). Both have their fine points, the former being more mathematically rigorous and the latter more readable. However, neither of these books is particularly easy to relate to chemistry. For this, one should consult the exceptionally clear writings of Roald Hoffman (https:/doi.org/10.1002/anie.198708461), and follow these with the work of Norskov and coworkers (https://doi.org/10.1016/S0360-0564(02)45013-4 & https://doi.org/10.1146/annurev.physchem.53.100301.131630).

In this chapter, only the elements of DFT that are relevant to this work will be discussed. An excellent review on other implementations of DFT can be found in Reference (https://doi.org/10.1146/annurev.ms.25.080195.000255), and details on the various algorithms used in DFT codes can be found in (https://doi.org/10.1103/RevModPhys.64.1045 & https://doi.org/10.1016/0927-0256(96)00008-0).

One of the most useful sources of information has been the dissertations of other students, perhaps because the difficulties they faced in learning the material are still fresh in their minds. Thomas Bligaard, a coauthor of Dacapo, wrote a particularly relevant thesis on exchange/correlation functionals (*Exchange and Correlation Functionals - a study toward improving the precision of electron density functional calculations of atomistic systems* http://www.fysik.dtu.dk/~{}bligaard/masterthesis/masterdirectory/project/project.pdf) and a dissertation illustrating the use of DFT to design new alloys with desirable thermal and mechanical properties (*Understanding Materials Properties on the Basis of Density Functional Theory Calculations* http://www.fysik.dtu.dk/~{}bligaard/phdthesis/phdproject.pdf). The Ph.D. thesis of Ari Seitsonen contains several useful appendices on k-point setups, and convergence tests of calculations, in addition to a thorough description of DFT and analysis of calculation output (*Theoretical Investigations into adsorption and co-adsorption on transition-metal surfaces as models to heterogeneous catalysis* http://edocs.tu-berlin.de/diss/2000/seitsonen_ari.pdf). Finally, another excellent overview of DFT and its applications to bimetallic alloy phase diagrams and surface reactivity is presented in the PhD thesis of Robin Hirschl (*Binary Transition Metal Alloys and Their Surfaces* http://www.hirschl.at/download/diss_part1.pdf and http://www.hirschl.at/download/diss_part2.pdf).

3.2.1 2.1 Background

In 1926, Erwin Schrodinger published the first accounts of his now famous wave equation (17 cite:pauling1963). He later shared the Nobel prize with Paul A. M. Dirac in 1933 for this discovery. Schrodinger's wave function seemed extremely promising, as it contains all of the information available about a system. Unfortunately, most practical systems of interest consist of many interacting electrons, and the effort required to find solutions to Schrodinger's equation increases exponentially with the number of electrons, limiting this approach to systems with a small number of relevant electrons, $N \lesssim O(10)$ (18 cite:RevModPhys.71.1253). Even if this rough estimate is off by an order of magnitude, a system with 100 electrons is still very small, for example, two Ru atoms if all the electrons are counted, or perhaps ten Pt atoms if only the valence electrons are counted. Thus, the wave function method, which has been extremely successful in studying the properties of small molecules, is unsuitable for studies of large, extended solids. Interestingly, this difficulty was recognized by Dirac as early as 1929, when he wrote "The underlying physical laws necessary for the mathematical theory of a large part of physics and the whole of chemistry are thus completely known, and the difficulty is only that the application of these laws leads to equations much too complicated to be soluble." (19 cite:dirac1929:quant-mechan-many-elect-system.)

In 1964, Hohenberg and Kohn showed that the ground state total energy of a system of interacting electrons is a unique functional of the electron density (20 cite:PhysRev.136.B864.) By definition, a function returns a number when given a number. For example, in $f(x) = x^2$, $f(x)$ is the function, and it equals four when $x = 2$. A functional returns a number when given a function. Thus, in $g(f(x)) = \int_0^\pi f(x)dx$, $g(f(x))$ is the functional, and it is equal to two when $f(x) = \sin(x)$. Hohenberg and Kohn further identified a variational principle that appeared to reduce the problem of finding the ground state energy of an electron gas in an external potential (i.e., in the presence of ion cores) to that of the minimization of a functional of the three-dimensional density function. Unfortunately, the definition of the functional involved a set of 3N-dimensional trial wave functions.

In 1965, Kohn and Sham made a significant breakthrough when they showed that the problem of many interacting electrons in an external potential can be mapped exactly to a set of noninteracting electrons in an effective external potential (21 cite:PhysRev.140.A1133.) This led to a set of self-consistent, single particle equations known as the Kohn-Sham (KS) equations:

$$\left(-\frac{1}{2}\nabla^2 + v_{eff}(\mathbf{r}) - \epsilon_j\right)\varphi_j(\mathbf{r}) = 0, (1)$$

with

$$v_{eff}(\mathbf{r}) = v(\mathbf{r}) + \int \frac{n(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} d\mathbf{r}' + v_{xc}(\mathbf{r}), (2)$$

where $v(\mathbf{r})$ is the external potential and $v_{xc}(\mathbf{r})$ is the exchange-correlation potential, which depends on the entire density function. Thus, the density needs to be known in order to define the effective potential so that Eq. (1) can be solved. $\varphi_j(\mathbf{r})$ corresponds to the j^{th} KS orbital of energy ϵ_j.

The ground state density is given by:

$$n(\mathbf{r}) = \sum_{j=1}^{N} |\varphi_j(\mathbf{r})|^2 (3)$$

To solve Eq. (1) then, an initial guess is used for $\varphi_j(r)$ which is used to generate Eq. (3), which is subsequently used in Eq. (2). This equation is then solved for $\varphi_j(\mathbf{r})$ iteratively until the $\varphi_j(\mathbf{r})$ that result from the solution are the same as the $\varphi_j(\mathbf{r})$ that are used to define the equations, that is, the solutions are self-consistent. Finally, the ground state energy is given by:

$$E = \sum_j \epsilon_j + E_{xc}[n(\mathbf{r})] - \int v_{xc}(\mathbf{r}) n(\mathbf{r}) d\mathbf{r} - \frac{1}{2} \int \frac{n(\mathbf{r}) n(\mathbf{r}')}{|\mathbf{r}-\mathbf{r}'|} d\mathbf{r}'\mathbf{r}, (4)$$

where $E_{xc}[n(\mathbf{r})]$ is the exchange-correlation energy functional. Walter Kohn shared the Nobel prize in Chemistry in 1998 for this work (18 cite:RevModPhys.71.1253). The other half of the prize went to John Pople for his efforts in wave function based quantum mechanical methods (22 cite:RevModPhys.71.1267). Provided the exchange-correlation energy functional is known, Eq. (4) is exact. However, the exact form of the exchange-correlation energy functional is not known, thus approximations for this functional must be used.

3.2.2 2.2 Exchange correlation functionals

The two main types of exchange/correlation functionals used in DFT are the local density approximation (LDA) and the generalized gradient approximation (GGA). In the LDA, the exchange-correlation functional is defined for an electron in a uniform electron gas of density n (21 cite:PhysRev.140.A1133). It is exact for a uniform electron gas, and is anticipated to be a reasonable approximation for slowly varying densities. In molecules and solids, however, the density tends to vary substantially in space. Despite this, the LDA has been very successfully used in many systems. It tends to predict overbonding in both molecular and solid systems (23 cite:fuchs1998:pseud), and it tends to make semiconductor systems too metallic (the band gap problem). (24 cite:perdew1982:elect-kohn-sham)

The generalized gradient approximation includes corrections for gradients in the electron density, and is often implemented as a corrective function of the LDA. The form of this corrective function, or "exchange enhancement" function determines which functional it is, e.g. PBE, RPBE, revPBE, etc. (25 cite:hammer1999:improv-pbe.) In this book the PBE GGA functional is used the most. N:nbsphinx-math:'o{}'rskov and coworkers have found that the RPBE functional gives superior chemisorption energies for atomic and molecular bonding to surfaces, but that it gives worse bulk properties, such as lattice constants compared to experimental data cite:hammer1999:improv-pbe.

Finally, there are increasingly new types of functionals in the literature. The so-called hybrid functionals, such as B3LYP, are more popular with gaussian basis sets (e.g. in Gaussian), but they are presently inefficient with planewave basis sets. None of these other types of functionals were used in this work. For more details see Chapter 6 in Ref. (3 citealp:koch2001) and Thomas Bligaard's thesis on exchange and correlation functionals. (13 cite:bligaard2000:exchan-correl-funct.)

3.2.3 2.3 Basis sets

Briefly, VASP utilizes planewaves as the basis set to expand the Kohn-Sham orbitals. In a periodic solid, one can use Bloch's theorem to show that the wave function for an electron can be expressed as the product of a planewave and a function with the periodicity of the lattice (5 cite:ashcroft-mermin):

$$\psi_{n\mathbf{k}}(\mathbf{r}) = \exp(i\mathbf{k}\cdot\mathbf{r})u_{n\mathbf{k}}(\mathbf{r})(5)$$

where $\mathbf{r}$ is a position vector, and $\mathbf{k}$ is a so-called wave vector that will only have certain allowed values defined by the size of the unit cell. Bloch's theorem sets the stage for using planewaves as a basis set, because it suggests a planewave character of the wave function. If the periodic function $u_{n\mathbf{k}}(\mathbf{r})$ is also expanded in terms of planewaves determined by wave vectors of the reciprocal lattice vectors, $\mathbf{G}$, then the wave function can be expressed completely in terms of a sum of planewaves (11 cite:payne1992:iterat):

$$\psi_i(\mathbf{r}) = \sum_{\mathbf{G}} c_{i,\mathbf{k}+\mathbf{G}} \exp(i(\mathbf{k}+\mathbf{G})\cdot\mathbf{r}).(6)$$

where $c_{i,\mathbf{k}+\mathbf{G}}$ are now coefficients that can be varied to determine the lowest energy solution. This also converts Eq.(1) from an integral equation to a set of algebraic equations that can readily be solved using matrix algebra.

In aperiodic systems, such as systems with even one defect, or randomly ordered alloys, there is no periodic unit cell. Instead one must represent the portion of the system of interest in a supercell, which is then subjected to the periodic boundary conditions so that a planewave basis set can be used. It then becomes necessary to ensure the supercell is large enough to avoid interactions between the defects in neighboring supercells. The case of the randomly ordered alloy is virtually hopeless as the energy of different configurations will fluctuate statistically about an average value. These systems were not considered in this work, and for more detailed discussions the reader is referred to (26 Ref. citealp:makov1995:period-bound-condit). Once a supercell is chosen, however, Bloch's theorem can be applied to the new artificially periodic system.

To get a perfect expansion, one needs an infinite number of planewaves. Luckily, the coefficients of the planewaves must go to zero for high energy planewaves, otherwise the energy of the wave function would go to infinity. This provides justification for truncating the planewave basis set above a cutoff energy. Careful testing of the effect of the cutoff energy on the total energy can be done to determine a suitable cutoff energy. The cutoff energy required to obtain a particular convergence precision is also element dependent, shown in Table 1. It can also vary with the "softness" of the pseudopotential. Thus, careful testing should be done to ensure the desired level of convergence of properties in different systems. Table ref:tab:pwcut refers to convergence of total energies. These energies are rarely considered directly, it is usually differences in energy that are important. These tend to converge with the planewave cutoff energy much more quickly than total energies, due to cancellations of convergence errors. In this work, 350 eV was found to be suitable for the H adsorption calculations, but a cutoff energy of 450 eV was required for O adsorption calculations.

Table 1: Planewave cutoff energies (in eV) required for different convergence precisions for two different elements with different pseudopotential setups.

Precision	Low	High
Mo	168	293
O	300	520
O_sv	1066	1847

Bloch's theorem eliminates the need to calculate an infinite number of wave functions, because there are only a finite number of electrons in the unit (super) cell. However, there are still an infinite number of discrete k points that must be considered, and the energy of the unit cell is calculated as an integral over these points. It turns out that wave functions at k points that are close together are similar, thus an interpolation scheme can be used with a finite number of k points. This also converts the integral used to determine the energy into a sum over the k points, which are suitably weighted to account for the finite number of them. There will be errors in the total energy associated with the finite number of k, but these can be reduced and tested for convergence by using higher k-point densities. An excellent discussion of this for aperiodic systems can be found in Ref. (26 citealp:makov1995:period-bound-condit.)

The most common schemes for generating k points are the Chadi-Cohen scheme (27 cite:PhysRevB.8.5747), and the Monkhorst-Pack scheme. (28 cite:PhysRevB.13.5188) The use of these k point setups amounts to an expansion of the periodic function in reciprocal space, which allows a straight-forward interpolation of the function between the points that is more accurate than with other k point generation schemes. (28 cite:PhysRevB.13.5188)

3.2.4 2.4 Pseudopotentials

The core electrons of an atom are computationally expensive with planewave basis sets because they are highly localized. This means that a very large number of planewaves are required to expand their wave functions. Furthermore, the contributions of the core electrons to bonding compared to those of the valence electrons is usually negligible. In fact, the primary role of the core electron wave functions is to ensure proper orthogonality between the valence electrons and core states. Consequently, it is desirable to replace the atomic potential due to the core electrons with a pseudopotential that has the same effect on the valence electrons. (29 cite:PhysRevB.43.1993.) There are essentially two kinds of pseudopotentials, norm-conserving soft pseudopotentials (cite:PhysRevB.43.1993) and Vanderbilt ultrasoft pseudopotentials. (cite:PhysRevB.41.7892) In either case, the pseudopotential function is generated from an all-electron calculation of an atom in some reference state. In norm-conserving pseudopotentials, the charge enclosed in the pseudopotential region is the same as that enclosed by the same space in an all-electron calculation. In ultrasoft pseudopotentials, this requirement is relaxed and charge augmentation functions are used to make up the difference. As its name implies, this allows a "softer" pseudopotential to be generated, which means fewer planewaves are required to expand it.

The pseudopotentials are not unique, and calculated properties depend on them. However, there are standard methods for ensuring the quality and transferability (to different chemical environments) of the pseudopotentials. (cite:PhysRevB.56.15629)

TODO PAW description

VASP provides a database of PAW potentials (32, 33 cite:PhysRevB.50.17953,PhysRevB.59.1758.)

3.2.5 2.5 Fermi Temperature and band occupation numbers

At absolute zero, the occupancies of the bands of a system are well-defined step functions; all bands up to the Fermi level are occupied, and all bands above the Fermi level are unoccupied. There is a particular difficulty in the calculation of the electronic structures of metals compared to semiconductors and molecules. In molecules and semiconductors, there is a clear energy gap between the occupied states and unoccupied states. Thus, the occupancies are insensitive to changes in the energy that occur during the self-consistency cycles. In metals, however, the density of states is continuous at the Fermi level, and there are typically a substantial number of states that are close in energy to the Fermi level. Consequently, small changes in the energy can dramatically change the occupation numbers, resulting in instabilities that make it difficult to converge to the occupation step function. A related problem is that the Brillouin zone integral (which in practice is performed as a sum over a finite number of k points) that defines the band energy converges very slowly with the number of k points due to the discontinuity in occupancies in a continuous distribution of states for metals (12, 34 cite:gillan1989:calcul,Kresse199615.) The difficulty arises because the temperature in most DFT calculations is at absolute zero. At higher temperatures, the DOS is smeared across the Fermi level, resulting in a continuous occupation function over the distribution of states. A finite-temperature version of DFT was developed (35 cite:PhysRev.137.A1441), which is the foundation on which one solution to this problem is based. In this solution, the step function is replaced by a smoothly varying function such as the Fermi-Dirac function at a small, but non-zero temperature. (12 cite:Kresse199615) The total energy is then extrapolated back to absolute zero.

3.2.6 2.6 Spin polarization and magnetism

There are two final points that need to be discussed about these calculations, spin polarization and dipole corrections. Spin polarization is important for systems that contain net spin. For example, iron, cobalt and nickel are magnetic because they have more electrons with spin "up" than spin "down" (or vice versa). Spin polarization must also be considered in atoms and molecules with unpaired electrons, such as hydrogen and oxygen atoms, oxygen molecules and radicals. For example, there are two spin configurations for an oxygen molecule, the singlet state with no unpaired electrons, and the triplet state with two unpaired electrons. The oxygen triplet state is lower in energy than the oxygen singlet state, and thus it corresponds to the ground state for an oxygen atom. A classically known problem involving spin polarization is the dissociation of a hydrogen molecule. In this case, the molecule starts with no net spin, but it dissociates into two atoms, each of which has an unpaired electron. See section 5.3.5 in Reference (3 citealp:koch2001) for more details on this.

In VASP, spin polarization is not considered by default; it must be turned on, and an initial guess for the magnetic moment of each atom in the unit cell must be provided (typically about one Bohr-magneton per unpaired electron). For Fe, Co, and Ni, the experimental values are 2.22, 1.72, and 0.61 Bohr-magnetons, respectively (4 cite:kittel) and are usually good initial guesses. See Reference (31 citealp:PhysRevB.56.15629) for a very thorough discussion of the determination of the magnetic properties of these metals with DFT. For a hydrogen atom, an initial guess of 1.0 Bohr-magnetons (corresponding to one unpaired electron) is usually good. An oxygen atom has two unpaired electrons, thus an initial guess of 2.0 Bohr-magnetons should be used. The spin-polarized solution is sensitive to the initial guess, and typically converges to the closest solution. Thus, a magnetic initial guess usually must be provided to get a magnetic solution. Finally, unless an adsorbate is on a magnetic metal surface, spin polarization typically does not need to be considered, although the gas-phase reference state calculation may need to be done with spin-polarization.

The downside of including spin polarization is that it essentially doubles the calculation time.

3.2.7 2.7 Recommended reading

The original papers on DFT are.(20,21 cite:PhysRev.136.B864,PhysRev.140.A1133)

Kohn's Nobel Lecture (18 cite:RevModPhys.71.1253) and Pople's Nobel Lecture (22 cite:RevModPhys.71.1267) are good reads.

This paper by Hoffman (7 cite:RevModPhys.60.601) is a nice review of solid state physics from a chemist's point of view.

All calculations in this book were performed using VASP (12, 36-38 cite:Kresse199615,PhysRevB.54.11169,PhysRevB.49.14251,PhysRevB.47.558) with the projector augmented wave (PAW) potentials provided in VASP.

3.3 3 Molecules

In this chapter we consider how to construct models of molecules, how to manipulate them, and how to calculate many properties of molecules. For a nice comparison of VASP and Gaussian see (https://doi.org/10.1063/1.1926272).

3.3.1 3.1 Defining and visualizing molecules

We start by learning how to define a molecule and visualize it. We will begin with defining molecules from scratch, then reading molecules from data files, and finally using some built-in databases in ase.

3.1.1 From scratch

When there is no data file for the molecule you want, or no database to get it from, you have to define your atoms geometry by hand. Here is how that is done for a CO molecule (Figure 1). We must define the type and position of each atom, and the unit cell the atoms are in.

[1]:
```python
from ase import Atoms, Atom
from ase.io import write
from ase.visualize import view

# define an Atoms object
atoms = Atoms(
```

(continues on next page)

(continued from previous page)

```
    [Atom("C", [0.0, 0.0, 0.0]), Atom("O", [1.1, 0.0, 0.0])], cell=(10, 10, 10)
)
view(atoms)
print("V = {0:1.0f} Angstrom^3".format(atoms.get_volume()))
# write("images/simple-cubic-cell.png", atoms, show_unit_cell=2)
```

```
V = 1000 Angstrom^3
```

There are two inconvenient features of the simple cubic cell:

1. Since the CO molecule is at the corner, its electron density is spread over the 8 corners of the box, which is not convenient for visualization later (see Visualizing electron density).
2. Due to the geometry of the cube, you need fairly large cubes to make sure the electron density of the molecule does not overlap with that of its images. Electron-electron interactions are repulsive, and the overlap makes the energy increase significantly. Here, the CO molecule has 6 images due to periodic boundary conditions that are 10 Å away. The volume of the unit cell is 1000 Å^3.

The first problem is easily solved by centering the atoms in the unit cell. The second problem can be solved by using a face-centered cubic lattice, which is the lattice with the closest packing. We show the results of the centering in Figure 2, where we have guessed values for b until the CO molecules are on average 10 Å apart. Note the final volume is only about 715 Å^3, which is smaller than the cube. This will result in less computational time to compute properties.

[1]:
```
from ase import Atoms, Atom
from ase.io import write
from ase.visualize import view
b = 7.1
atoms = Atoms(
    [Atom("C", [0.0, 0.0, 0.0]), Atom("O", [1.1, 0.0, 0.0])],
    cell=[[b, b, 0.0], [b, 0.0, b], [0.0, b, b]],
)
print("V = {0:1.0f} Ang^3".format(atoms.get_volume()))
atoms.center()  # translate atoms to center of unit cell
view(atoms, viewer="x3d")
```

```
V = 716 Ang^3
```

[1]:
```
<IPython.core.display.HTML object>
```

At this point you might ask, "How do you know the distance to the neighboring image?" The `ase gui` viewer lets you compute this graphically, but we can use code to determine this too. All we have to do is figure out the length of each lattice vector, because these are what separate the atoms in the images. We use the mod:numpy module to compute the distance of a vector as the square root of the sum of squared elements.

[2]:
```
from ase import Atoms, Atom
import numpy as np

b = 7.1
atoms = Atoms([Atom('C', [0., 0., 0.]),
               Atom('O', [1.1, 0., 0.])],
              cell=[[b, b, 0.],
                    [b, 0., b],
                    [0., b, b]])

# get unit cell vectors and their lengths
(a1, a2, a3) = atoms.get_cell()
print('|a1| = {0:1.2f} Ang'.format(np.sum(a1**2)**0.5))
print('|a2| = {0:1.2f} Ang'.format(np.linalg.norm(a2)))
print('|a3| = {0:1.2f} Ang'.format(np.sum(a3**2)**0.5))
```

```
|a1| = 10.04 Ang
|a2| = 10.04 Ang
|a3| = 10.04 Ang
```

3.1.2 Reading other data formats into a calculation

ase.io.read supports many different file formats: https://wiki.fysik.dtu.dk/ase/ase/io/io.html?highlight=ase.io.write#ase.io.write

You can read XYZ file format to create ase.Atoms objects. Here is what an XYZ file format might look like:

`#+include: molecules/isobutane.xyz`

The first line is the number of atoms in the file. The second line is often a comment. What follows is one line per atom with the symbol and Cartesian coordinates in Å. Note that the XYZ format does not have unit cell information in it, so you will have to figure out a way to provide it. In this example, we center the atoms in a box with vacuum on all sides.

```
[2]: from ase.io import read, write
     from ase.visualize import view

     atoms = read('../molecules/isobutane.xyz')
     atoms.center(vacuum=5)
     view(atoms, viewer="x3d")
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-2-fc251ef99eaa> in <module>
      2 from ase.visualize import view
      3
----> 4 atoms = read('../molecules/isobutane.xyz')
      5 atoms.center(vacuum=5)
      6 view(atoms, viewer="x3d")

~/miniconda3/lib/python3.7/site-packages/ase/io/formats.py in read(filename, index,␣
↪format, parallel, **kwargs)
    603     else:
    604         return next(_iread(filename, slice(index, None), format, io,
--> 605                            parallel=parallel, **kwargs))
    606
    607

~/miniconda3/lib/python3.7/site-packages/ase/parallel.py in new_generator(*args,␣
↪**kwargs)
    262             not kwargs.pop('parallel', True)):
    263             # Disable:
--> 264             for result in generator(*args, **kwargs):
    265                 yield result
    266             return

~/miniconda3/lib/python3.7/site-packages/ase/io/formats.py in _iread(filename, index,␣
↪format, io, parallel, full_output, **kwargs)
    651         if io.acceptsfd:
    652             mode = 'rb' if io.isbinary else 'r'
--> 653             fd = open_with_compression(filename, mode)
    654             must_close_fd = True
    655         else:
```

(continues on next page)

(continued from previous page)

```
~/miniconda3/lib/python3.7/site-packages/ase/io/formats.py in open_with_
↪compression(filename, mode)
    429
    430     if compression is None:
--> 431         return open(filename, mode)
    432     elif compression == 'gz':
    433         import gzip

FileNotFoundError: [Errno 2] No such file or directory: '../molecules/isobutane.xyz'
```

3.1.3 Predefined molecules

ase defines a number of molecular geometries in the `ase.data.molecules` database. For example, the database includes the molecules in the G2/97 database (40 cite:curtiss:1063). This database contains a broad set of atoms and molecules for which good experimental data exists, making them useful for benchmarking studies. See this site for the original files.

The coordinates for the atoms in the database are MP2(full)/6-31G(d) optimized geometries. Here is a list of all the species available in `ase.data.g2`. You may be interested in reading about some of the other databases in `ase.data` too.

[7]:
```
from ase.data import g2
keys = g2.data.keys()
print(keys)
```

```
dict_keys(['Be', 'BeH', 'C', 'C2H2', 'C2H4', 'C2H6', 'CH', 'CH2_s1A1d', 'CH2_s3B1d',
↪'CH3', 'CH3Cl', 'CH3OH', 'CH3SH', 'CH4', 'CN', 'CO', 'CO2', 'CS', 'Cl', 'Cl2', 'ClF
↪', 'ClO', 'F', 'F2', 'H', 'H2CO', 'H2O', 'H2O2', 'HCN', 'HCO', 'HCl', 'HF', 'HOCl',
↪'Li', 'Li2', 'LiF', 'LiH', 'N', 'N2', 'N2H4', 'NH', 'NH2', 'NH3', 'NO', 'Na', 'Na2',
↪ 'NaCl', 'O', 'O2', 'OH', 'P', 'P2', 'PH2', 'PH3', 'S', 'S2', 'SH2', 'SO', 'SO2',
↪'Si', 'Si2', 'Si2H6', 'SiH2_s1A1d', 'SiH2_s3B1d', 'SiH3', 'SiH4', 'SiO', '2-butyne',
↪ 'Al', 'AlCl3', 'AlF3', 'B', 'BCl3', 'BF3', 'C2Cl4', 'C2F4', 'C2H3', 'C2H5',
↪'C2H6CHOH', 'C2H6NH', 'C2H6SO', 'C3H4_C2v', 'C3H4_C3v', 'C3H4_D2d', 'C3H6_Cs',
↪'C3H6_D3h', 'C3H7', 'C3H7Cl', 'C3H8', 'C3H9C', 'C3H9N', 'C4H4NH', 'C4H4O', 'C4H4S',
↪'C5H5N', 'C5H8', 'C6H6', 'CCH', 'CCl4', 'CF3CN', 'CF4', 'CH2NHCH2', 'CH2OCH2',
↪'CH2SCH2', 'CH3CH2Cl', 'CH3CH2NH2', 'CH3CH2O', 'CH3CH2OCH3', 'CH3CH2OH', 'CH3CH2SH',
↪ 'CH3CHO', 'CH3CN', 'CH3CO', 'CH3COCH3', 'CH3COCl', 'CH3COF', 'CH3CONH2', 'CH3COOH',
↪ 'CH3NO2', 'CH3O', 'CH3OCH3', 'CH3ONO', 'CH3S', 'CH3SCH3', 'CH3SiH3', 'COF2', 'CS2',
↪ 'ClF3', 'ClNO', 'F2O', 'H2', 'H2CCHCN', 'H2CCHCl', 'H2CCHF', 'H2CCO', 'H2CCl2',
↪'H2CF2', 'H2COH', 'H3CNH2', 'HCCl3', 'HCF3', 'HCOOCH3', 'HCOOH', 'N2O', 'NCCN', 'NF3
↪', 'NO2', 'O3', 'OCHCHO', 'OCS', 'PF3', 'SH', 'SiCl4', 'SiF4', 'bicyclobutane',
↪'butadiene', 'cyclobutane', 'cyclobutene', 'isobutane', 'isobutene',
↪'methylenecyclopropane', 'trans-butane'])
```

Some other databases include the `ase.data.s22` for weakly interacting dimers and complexes, and `ase.data.extra_molecules` which has a few extras like biphenyl and C60.

Here is an example of getting the geometry of an acetonitrile molecule and writing an image to a file. Note that the default unit cell is a 1 Å × Å × Å cubic cell. That is too small to use if your calculator uses periodic boundary conditions. We center the atoms in the unit cell and add vacuum on each side. We will add 6 Å of vacuum on each side. In the write command we use the option show_unit_cell =2 to draw the unit cell boundaries.

[3]:
```
from ase.build import molecule
from ase.io import write
from ase.visualize import view
```

(continues on next page)

(continued from previous page)

```
atoms = molecule('CH3CN')

atoms.center(vacuum=6)
print('unit cell')
print('---------')
print(atoms.get_cell())

view(atoms, viewer="x3d")
```

```
unit cell
---------
Cell([13.775328, 13.537479, 15.014576])
```

```
[3]: <IPython.core.display.HTML object>
```

It is possible to rotate the atoms with `ase.io.write` if you wanted to see pictures from another angle. In the next example we rotate 45 degrees about the x-axis, then 45 degrees about the y-axis. Note that this only affects the image, not the actual coordinates.

```
[3]: from ase.build import molecule
     from ase.io import write
     atoms = molecule('CH3CN')
     atoms.center(vacuum=6)
     print('unit cell')
     print('---------')
     print(atoms.get_cell())
     write('images/ch3cn-rotated.png', atoms,
           show_unit_cell=2, rotation='45x,45y,0z')
```

```
unit cell
---------
[[13.775328  0.         0.       ]
 [ 0.        13.537479  0.       ]
 [ 0.         0.        15.014576]]
```

If you actually want to rotate the coordinates, there is a nice way to do that too, with the `ase.Atoms. rotate` method. Actually there are some subtleties in rotation. One rotates the molecule an angle (in radians) around a vector, but you have to choose whether the center of mass should be fixed or not. You also must decide whether or not the unit cell should be rotated. In the next example you can see the coordinates have changed due to the rotations. Note that the write function uses the rotation angle in degrees, while the rotate function uses radians.

```
[5]: from ase.build import molecule
     from numpy import pi
     from ase.visualize import view

     atoms = molecule("CH3CN")
     atoms.center(vacuum=6)
     p1 = atoms.get_positions()
     atoms.rotate(pi / 4, v="x", center="COM", rotate_cell=False)
     atoms.rotate(pi / 4, v="y", center="COM", rotate_cell=False)
     view(atoms)
     print("difference in positions after rotating")
     print("atom    difference vector")
     print("--------------------------------------")
     p2 = atoms.get_positions()
     diff = p2 - p1
```

(continues on next page)

(continued from previous page)

```
for i, d in enumerate(diff):
    print("{0} {1}".format(i, d))
```

```
difference in positions after rotating
atom    difference vector
--------------------------------------
0 [-0.01782051  0.01782219  0.0002443 ]
1 [ 2.20136479e-03 -2.20157163e-03 -3.01777230e-05]
2 [ 0.01835166 -0.01835339 -0.00025158]
3 [-0.02277373  0.02287218  0.01436336]
4 [-0.02314601  0.02301662 -0.01887695]
5 [-0.02297922  0.02301662  0.0054581 ]
```

Note in this last case the unit cell is oriented differently than the previous example, since we chose not to rotate the unit cell.

3.1.4 Combining Atoms objects

It is frequently useful to combine two `Atoms` objects, e.g. for computing reaction barriers, or other types of interactions. In ase, we simply add two Atoms objects together. Here is an example of getting an ammonia and oxygen molecule in the same unit cell. We set the `Atoms` about three Å apart using the `ase.Atoms.translate` function.

[1]:
```
from ase.build import molecule
from ase.io import write
from ase.visualize import view

atoms1 = molecule("NH3")
atoms2 = molecule("O2")
atoms2.translate([3, 0, 0])
bothatoms = atoms1 + atoms2
bothatoms.center(5)
view(bothatoms, viewer="x3d")
```

[1]:
```
<IPython.core.display.HTML object>
```

3.3.2 3.2 Simple properties

Simple properties do not require a DFT calculation. They are typically only functions of the atom types and geometries.

3.2.1 Getting cartesian positions

If you want the (x, y, z) coordinates of the atoms, use the `ase.Atoms.get_positions`. If you are interested in the fractional coordinates, use `ase.Atoms.get_scaled_positions`.

[5]:
```
from ase.build import molecule

atoms = molecule("C6H6")

print(atoms)
print(atoms[0])
print(len(atoms))

# access properties on each atom
```

(continues on next page)

(continued from previous page)

```
for i, atom in enumerate(atoms):
    print(i, atom.symbol, atom.x, atom.y, atom.z)

# get all properties in arrays
sym = atoms.get_chemical_symbols()
pos = atoms.get_positions()
num = atoms.get_atomic_numbers()
atom_indices = range(len(atoms))

for i, s, n, p in zip(atom_indices, sym, num, pos):
    px, py, pz = p
    print(i, s, n, px, py, pz)
```

```
Atoms(symbols='C6H6', pbc=False)
Atom('C', [0.0, 1.395248, 0.0], index=0)
12
0 C 0.0 1.395248 0.0
1 C 1.20832 0.697624 0.0
2 C 1.20832 -0.697624 0.0
3 C 0.0 -1.395248 0.0
4 C -1.20832 -0.697624 0.0
5 C -1.20832 0.697624 0.0
6 H 0.0 2.48236 0.0
7 H 2.149787 1.24118 0.0
8 H 2.149787 -1.24118 0.0
9 H 0.0 -2.48236 0.0
10 H -2.149787 -1.24118 0.0
11 H -2.149787 1.24118 0.0
0 C 6 0.0 1.395248 0.0
1 C 6 1.20832 0.697624 0.0
2 C 6 1.20832 -0.697624 0.0
3 C 6 0.0 -1.395248 0.0
4 C 6 -1.20832 -0.697624 0.0
5 C 6 -1.20832 0.697624 0.0
6 H 1 0.0 2.48236 0.0
7 H 1 2.149787 1.24118 0.0
8 H 1 2.149787 -1.24118 0.0
9 H 1 0.0 -2.48236 0.0
10 H 1 -2.149787 -1.24118 0.0
11 H 1 -2.149787 1.24118 0.0
```

3.2.2 Molecular weight and molecular formula

We can quickly compute the molecular weight of a molecule with this recipe. We use `ase.Atoms.get_masses` to get an array of the atomic masses of each atom in the Atoms object, and then just sum them up.

[8]:
```
from ase.build import molecule

atoms = molecule("C6H6")
masses = atoms.get_masses()
print(masses)

molecular_weight = masses.sum()
print(molecular_weight)

```

(continues on next page)

(continued from previous page)

```
molecular_formula = atoms.get_chemical_formula(mode="reduce")
print(molecular_formula)
```

```
[12.011 12.011 12.011 12.011 12.011 12.011  1.008  1.008  1.008  1.008
  1.008  1.008]
78.11399999999998
C6H6
```

Note that the argument reduce=True for `ase.Atoms.get_chemical_formula` collects all the symbols to provide a molecular formula.

The center of mass (COM) is defined as:

$COM = \frac{\sum m_i \cdot r_i}{\sum m_i}$

The center of mass is essentially the average position of the atoms, weighted by the mass of each atom.
Here is an example of getting the center of mass from an `Atoms` object using
`ase.Atoms.get_center_of_mass`.
You can see see that these centers of mass, which are calculated by different methods, are the same.

[13]:
```
from ase.build import molecule
import numpy as np

atoms = molecule("NH3")
# cartesian coordinates
print("COM1 = {0}".format(atoms.get_center_of_mass()))

# compute the center of mass by hand
pos = atoms.positions
masses = atoms.get_masses()
COM = np.array([0.0, 0.0, 0.0])
for m, p in zip(masses, pos):
    COM += m * p
COM /= masses.sum()

print("COM2 = {0}".format(np.dot(masses, pos) / np.sum(masses)))
```

```
COM1 = [1.29821427e-19 5.91861899e-08 4.75435401e-02]
COM2 = [1.29821427e-19 5.91861899e-08 4.75435401e-02]
```

3.2.4 Moments of inertia

The moment of inertia is a measure of resistance to changes in rotation.
It is defined by $I = \sum_{i=1}^{N} m_i r_i^2$ where r_i is the distance to an axis of rotation.
There are typically three moments of inertia, although some may be zero depending on symmetry, and others may be degenerate.
There is a convenient function to get the moments of inertia: `ase.Atoms.get_moments_of_inertia`.
Here are several examples of molecules with different types of symmetry.:

[16]:
```
from ase.build import molecule

print("linear rotors: I = [0 Ia Ia]")
atoms = molecule("CO2")
print("CO2 moments of inertia: {}".format(atoms.get_moments_of_inertia()))

print("symmetric rotors (Ia = Ib) < Ic")
atoms = molecule("NH3")
print("NH3 moments of inertia: {}".format(atoms.get_moments_of_inertia()))

atoms = molecule("C6H6")
print("C6H6 moments of inertia: {}".format(atoms.get_moments_of_inertia()))

print("symmetric rotors Ia < (Ib = Ic)")
atoms = molecule("CH3Cl")
print("CH3Cl moments of inertia: {}".format(atoms.get_moments_of_inertia()))

print("spherical rotors Ia = Ib = Ic")
atoms = molecule("CH4")
print("CH4 moments of inertia: {}".format(atoms.get_moments_of_inertia()))

print("unsymmetric rotors Ia != Ib != Ic")
atoms = molecule("C3H7Cl")
print("C3H7Cl moments of inertia: {}".format(atoms.get_moments_of_inertia()))
```

```
linear rotors: I = [0 Ia Ia]
CO2 moments of inertia: [ 0.          44.45273132 44.45273132]
symmetric rotors (Ia = Ib) < Ic
NH3 moments of inertia: [1.71022353 1.71022474 2.67047664]
C6H6 moments of inertia: [ 88.78025559  88.78027717 177.56053276]
symmetric rotors Ia < (Ib = Ic)
CH3Cl moments of inertia: [ 3.2039126  37.969823   37.96982492]
spherical rotors Ia = Ib = Ic
CH4 moments of inertia: [3.19164619 3.19164619 3.19164619]
unsymmetric rotors Ia != Ib != Ic
C3H7Cl moments of inertia: [ 19.41420447 213.18480664 223.1578698 ]
```

If you want to know the principle axes of rotation, we simply pass `vectors=True` to the function, and it returns the moments of inertia and the principle axes.

This shows the first moment is about the z-axis, the second moment is about the y-axis, and the third moment is about the x-axis.

[18]:
```
from ase.build import molecule

atoms = molecule("CH3Cl")
moments, axes = atoms.get_moments_of_inertia(vectors=True)
print("Moments = {0}".format(moments))
print("axes = {0}".format(axes))
```

```
Moments = [ 3.2039126  37.969823   37.96982492]
axes = [[0. 0. 1.]
 [0. 1. 0.]
 [1. 0. 0.]]
```

3.2.5 Computing bond lengths and angles

A typical question we might ask is, "What is the structure of a molecule?"
In other words, what are the bond lengths, angles between bonds, and similar properties.
The Atoms object contains an `ase.Atoms.get_distance` method to make this easy.
To calculate the distance between two atoms, you have to specify their indices, remembering that the index starts at 0.

[21]:
```
from ase.build import molecule

atoms = molecule("NH3")

for i, atom in enumerate(atoms):
    print(i, atom.symbol)

print("The N-H distance is {} angstroms".format(atoms.get_distance(0, 1)))
```

```
0 N
1 H
2 H
3 H
The N-H distance is 1.0167934463645996 angstroms
```

If we had vectors describing the directions between two atoms, we could use some simple trigonometry to compute the angle between the vectors: $\vec{a} \cdot \vec{b} = |\vec{a}||\vec{b}| \cos(\theta)$.
So we can calculate the angle as $\theta = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{|\vec{a}||\vec{b}|}\right)$, we just have to define our two vectors $\vec{a}$ and $\vec{b}$.
For example, here we compute the angle H-N-H in an ammonia molecule.
This is the angle between N-H_1 and N-H_2.

[29]:
```
from ase.build import molecule

atoms = molecule("NH3")

print("theta = {} degrees".format(atoms.get_angle(1, 0, 2)))
```

```
theta = 106.33462423179174 degrees
```

There is support in ase for computing dihedral angles.
Let us illustrate that for ethane.
We will compute the dihedral angle between atoms 5, 1, 0, and 4.
That is a H-C-C-H dihedral angle, and one can visually see that these atoms have a dihedral angle of 60 degrees.

[34]:
```
# calculate an ethane dihedral angle
from ase.build import molecule
import numpy as np
from ase.visualize import view

atoms = molecule("C2H6")
```

(continues on next page)

(continued from previous page)

```
view(atoms)

print("dihedral angle = {} degrees".format(atoms.get_dihedral(5, 1, 0, 4)))
```

```
dihedral angle = 59.99998556610882 degrees
```

3.3.3 3.3 Simple properties that require single computations

There are many properties that only require a single DFT calculation to obtain the energy, forces, density of states, electron density and electrostatic potential. This section describes some of these calculations and their analysis.

3.3.0 The compilation of VASP on macOS

1. Download and install Intel fortran and C++ compiler and corresponding MKL
2. `open /Library/Developer/CommandLineTools/Packages/macOS_SDK_headers_for_macOS_10.14.pkg` and install it
3. `cd vasp.5.4.4`
4. `cp arch/makefile.include.linux_intel_serial makefile.include`
5. open makefile.include and change `MKL_PATH = $(MKLROOT)/lib/intel64` to `MKL_PATH = $(MKLROOT)/lib` and `-lstdc++` to `-lc++`
6. `make std`

3.3.1 Energy and forces

Two of the most important quantities we are interested in are the total energy and the forces on the atoms.
To get these quantities, we have to define a calculator and attach it to an `ase.Atoms` object so that `ase` knows how to get the data.
After defining the calculator a DFT calculation must be run.

Here is an example of getting the energy and forces from a CO molecule.
The forces in this case are very high, indicating that this geometry is not close to the ground state geometry.
Note that the forces are only along the x-axis, which is along the molecular axis.
We will see how to minimize this force in 3.4.1 Manual determination and 3.4.2 Automatic geometry optimization with VASP.

This is your first DFT calculation in the book! See `ISMEAR`, `SIGMA`, `NBANDS`, and `ENCUT` to learn more about these VASP keywords.

[14]:
```
from ase import Atoms, Atom
from ase.calculators.vasp import Vasp

co = Atoms([Atom("C", [0, 0, 0]), Atom("O", [1.2, 0, 0])], cell=(6.0, 6.0, 6.0))
calc = Vasp(
    xc="pbe",
    nbands=6,  # number of bands
```

(continues on next page)

(continued from previous page)

```
    encut=350,  # planewave cutoff
    ismear=1,  # Methfessel-Paxton smearing
    sigma=0.01,  # very small smearing factor for a molecule
)
co.set_calculator(calc)
print("energy = {0} eV".format(co.get_potential_energy()))
print(co.get_forces())
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
<ipython-input-14-5268d0e4fdb4> in <module>
     11 )
     12 co.set_calculator(calc)
---> 13 print("energy = {0} eV".format(co.get_potential_energy()))
     14 print(co.get_forces())

~/anaconda3/lib/python3.7/site-packages/ase/atoms.py in get_potential_energy(self,␣
↪force_consistent, apply_constraint)
    669                 self, force_consistent=force_consistent)
    670         else:
--> 671             energy = self._calc.get_potential_energy(self)
    672         if apply_constraint:
    673             for constraint in self.constraints:

~/anaconda3/lib/python3.7/site-packages/ase/calculators/vasp/vasp.py in get_potential_
↪energy(self, atoms, force_consistent)
    229
    230     def get_potential_energy(self, atoms, force_consistent=False):
--> 231         self.update(atoms)
    232         if force_consistent:
    233             return self.energy_free

~/anaconda3/lib/python3.7/site-packages/ase/calculators/vasp/vasp.py in update(self,␣
↪atoms)
     81                 # calculator, so delete any old VASP files found.
     82                 self.clean()
---> 83             self.calculate(atoms)
     84
     85     def calculate(self, atoms):

~/anaconda3/lib/python3.7/site-packages/ase/calculators/vasp/vasp.py in␣
↪calculate(self, atoms)
    104
    105         # Execute VASP
--> 106         self.run()
    107         # Read output
    108         atoms_sorted = ase.io.read('CONTCAR', format='vasp')

~/anaconda3/lib/python3.7/site-packages/ase/calculators/vasp/vasp.py in run(self)
    175         sys.stderr = stderr
    176         if exitcode != 0:
--> 177             raise RuntimeError('Vasp exited with exit code: %d.  ' % exitcode)
    178
    179     def restart_load(self):

RuntimeError: Vasp exited with exit code: 6.
```

[ ]:
```
from vasp import Vasp

print(Vasp("molecules/simple-co").vasp)
```

[ ]:
```
from ase import Atoms, Atom
from vasp import Vasp
from vasp.vasprc import VASPRC

VASPRC["queue.ppn"] = 4
co = Atoms([Atom("C", [0, 0, 0]), Atom("O", [1.2, 0, 0])], cell=(6.0, 6.0, 6.0))
calc = Vasp(
    "molecules/simple-co-n4",  # output dir
    xc="PBE",  # the exchange-correlation functional
    nbands=6,  # number of bands
    encut=350,  # planewave cutoff
    ismear=1,  # Methfessel-Paxton smearing
    sigma=0.01,  # very small smearing factor for a molecule
    atoms=co,
)
print("energy = {0} eV".format(co.get_potential_energy()))
print(co.get_forces())
```

[ ]:
```
from vasp import Vasp
from ase import Atoms, Atom
import numpy as np

np.set_printoptions(precision=3, suppress=True)
atoms = Atoms([Atom("C", [0, 0, 0]), Atom("O", [1.2, 0, 0])])
L = [4, 5, 6, 8, 10]
energies = []
ready = True
for a in L:
    atoms.set_cell([a, a, a], scale_atoms=False)
    atoms.center()
    calc = Vasp("molecules/co-L-{0}".format(a), encut=350, xc="PBE", atoms=atoms)
    energies.append(atoms.get_potential_energy())
print(energies)
calc.stop_if(None in energies)
import matplotlib.pyplot as plt

plt.plot(L, energies, "bo-")
plt.xlabel("Unit cell length ($\AA$)")
plt.ylabel("Total energy (eV)")
plt.savefig("images/co-e-v.png")
```

[ ]:
```
from vasp import Vasp

L = [4, 5, 6, 8, 10]
for a in L:
    calc = Vasp("molecules/co-L-{0}".format(a))
    print("{0} {1} seconds".format(a, calc.get_elapsed_time()))
```

[ ]:
```
from ase.units import kB, Pascal
import numpy as np
import matplotlib.pyplot as plt
```

(continues on next page)

(continued from previous page)

```
atm = 101325 * Pascal
L = np.linspace(4, 10)
V = L ** 3
n = 1  # one atom/molecule per unit cell
for T in [298, 600, 1000]:
    P = n / V * kB * T / atm  # convert to atmospheres
    plt.plot(V, P, label="{0}K".format(T))
plt.xlabel("Unit cell volume ($\AA^3$)")
plt.ylabel("Pressure (atm)")
plt.legend(loc="best")
plt.savefig("images/ideal-gas-pressure.png")
```

[ ]:
```
from ase import Atoms, Atom
from vasp import Vasp
import numpy as np

np.set_printoptions(precision=3, suppress=True)
atoms = Atoms([Atom("C", [0, 0, 0]), Atom("O", [1.2, 0, 0])], cell=(6, 6, 6))
atoms.center()
ENCUTS = [250, 300, 350, 400, 450, 500]
calcs = [
    Vasp("molecules/co-en-{0}".format(en), encut=en, xc="PBE", atoms=atoms)
    for en in ENCUTS
]
energies = [calc.potential_energy for calc in calcs]
print(energies)
calcs[0].stop_if(None in energies)
import matplotlib.pyplot as plt

plt.plot(ENCUTS, energies, "bo-")
plt.xlabel("ENCUT (eV)")
plt.ylabel("Total energy (eV)")
plt.savefig("images/co-encut-v.png")
```

3.3.4 3.5 Vibrational frequencies

3.5.1 Manual calculation of vibrational frequency

The principle idea in calculating vibrational frequencies is that we consider a molecular system as masses connected by springs. If the springs are Hookean, e.g. the force is proportional to the displacement, then we can readily solve the equations of motion and find that the vibrational frequencies are related to the force constants and the masses of the atoms.

For example, in a simple molecule like CO where there is only one spring, the frequency is:
$\nu = \frac{1}{2\pi}\sqrt{k/\mu}$
where $\frac{1}{\mu} = \frac{1}{m_C} + \frac{1}{m_O}$ and k is the spring constant. We will compute the value of k from DFT calculations as follows: $k = \frac{\partial^2 E}{\partial x^2}$ at the equilibrium bond length. We actually already have the data to do this from Manual determination. We only need to fit an equation to the energy vs. bond-length data, find the minimum energy bond-length, and then evaluate the second derivative of the fitted function at the minimum. We will use a cubic polynomial for demonstration here. Polynomials are numerically convenient because they are easy to fit, and it is trivial to get the roots and derivatives of the polynomials, as well as to evaluate them at other points using `numpy.polyfit`, `numpy.polyder`, and `numpy.polyval`.

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```

04

05

06

07

08

09