Hands-On Exercises

PARADIM Summer School
An Introduction to Density Functional Theory for Experimentalists Including the Latest Advances

Betül Pamuk, Feliciano Giustino, Tomás Arias
Cornell University
July 23-29, 2023

These hands-on exercises are heavily based on, and in parts directly reproduced from the past PARADIM Summer Schools "An Introduction to Density Functional Theory for Experimentalists" from 2018 and 2021, prepared and co-lectured by Feliciano Giustino of UT Austin as well as the tutorials at the JDFTx website http://jdftx.org/Tutorials.html.

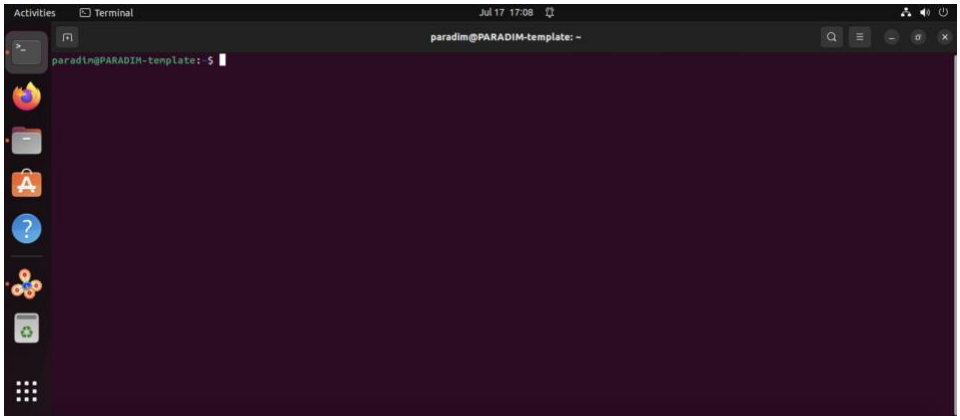
The original lecture notes and videos from past PARADIM summer schools by Feliciano Giustino can be accessed on the website: https://www.paradim.org/summer_schools

Hands-On 0
Connecting to the supercomputer, ARCH at JHU

We will be performing our calculations at the educational cluster of Advanced Research Computing at Hopkins (ARCH) supercomputer at Johns Hopkins University. Different operating systems can use different tools to connect to supercomputers:

Linux or macOS

For computers that run on Linux (e.g. Ubuntu, RedHat, etc.) or macOS, search for and open the Terminal app. We will use this option for the remainder of the Hands-On sessions. For the virtual machines that we use in this summer school, you will see the something like the following screen when you open the Terminal:



We can now monitor the network availability of the cluster:

```
$ ping paradim.arch.jhu.edu
PING paradim.arch.jhu.edu (162.129.223.75): 56 data bytes
64 bytes from 162.129.223.75: icmp_seq=0 ttl=30 time=82.495 ms
64 bytes from 162.129.223.75: icmp_seq=1 ttl=30 time=79.949 ms
```

To stop pinging, press `Ctrl+C`. Once you see the ping response, you are ready to continue to Hands-On 1.

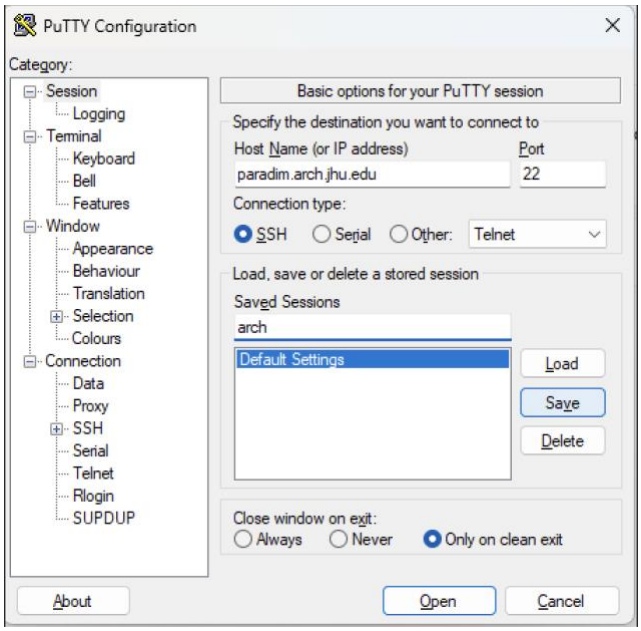
Windows

For computers that run on Windows, you will need a software that can emulate a terminal and support a secure shell (SSH) connection.

A popular choice our of many is PuTTY, which can be downloaded from:

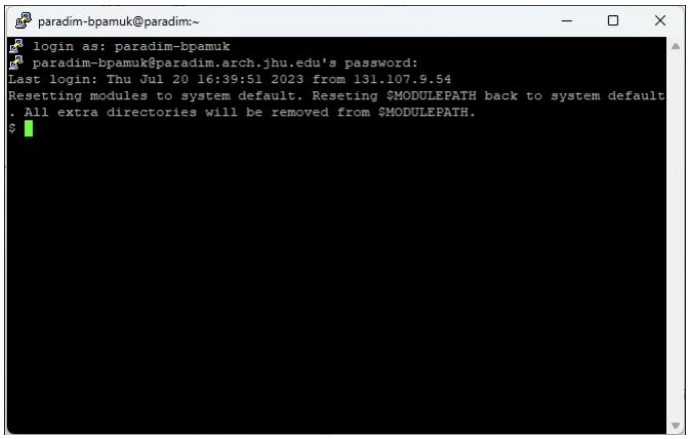
https://www.putty.org

Once you install and open PuTTY, you will see the following screen:



Fist time you use PuTTY to connect to ARCH, in the field for “Host Name”, you will enter:
`paradim.arch.jhu.edu`
and in “Saved Sessions” enter:
`arch`

Then it will open a terminal window and ask you to enter your username and password:



Next time you open PuTTY, you will see “arch” under the “Saved Sessions”, so you can “Load” it directly.

From here on, everything is the same for all operating systems.

Hands-On 1
Setting up ARCH

Login shell and setting up the modules

We will perform calculations on the at the educational cluster of ARCH supercomputer at JHU.

To connect to arch, we first open a terminal as we have learned in previous section, and then type:

```
$ ssh paradim.arch.jhu.edu -l paradim-bpamuk
```

where the username `paradim-bpamuk` must be replaced by the username that you have been assigned. Once you hit enter, you will be prompted to enter your password:

```
paradim-bpamuk@paradim.arch.jhu.edu's password:
```

The very first time, your password is:
`temppass`

Then you should run the command:
`$ passwd`
and follow instructions on the terminal to set up your own unique password, which you will use going forward.

Note that nothing appears while you are entering your password for security reasons, but it still registers what you are typing. When you enter your password, you will see a message with your last login info:

```
Last login: Sun Jul 16 16:28:48 2023 from 50.46.233.216
[paradim-bpamuk@paradim ~]$
```

We can now customize the Unix shell environment by editing the `.bashrc` file by shortening the prompt, creating alias'es for some shortcuts, loading modules and some commands that we will need to run the JDFTx code. Note that when you are copying and pasting from this document, you need copy the formatting correctly, because the `bash` shell is very strict with spaces.

```
$ cat >> .bashrc << EOF
PS1="$ "
alias c="clear"
alias l="ls -lrth --color="none" -column"
export SCRATCH="/scratch/paradim/paradim-bpamuk"
ml restore
module load jdftx/1.7.0-gcc
PATH="/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/scripts/:$PATH"
EOF
$ source ~/.bashrc
```

Make sure to change `paradim-bpamuk` with your own username. This will make the prompt to be simply `$` and the shortcuts command `c` for clearing the screen and `l` for listing the contents of a directory. The JDFTx code is already installed in the cluster, and we can access its executables by loading it as a module. Putting it in this file enables loading that module automatically when we log in to the cluster. We also add the post-processing scripts into our PATH to access them directly.

Next, we want to create a scratch directory (`mkdir`) from where we will run our calculations. The supercomputers are set up such that the users should never perform their calculations directly from their home directory, as this will be inefficient and slow down the collective access to the cluster. All the calculations must run from the scratch directory.

```
$ mkdir $SCRATCH
$ chmod go-rwx $SCRATCH
```

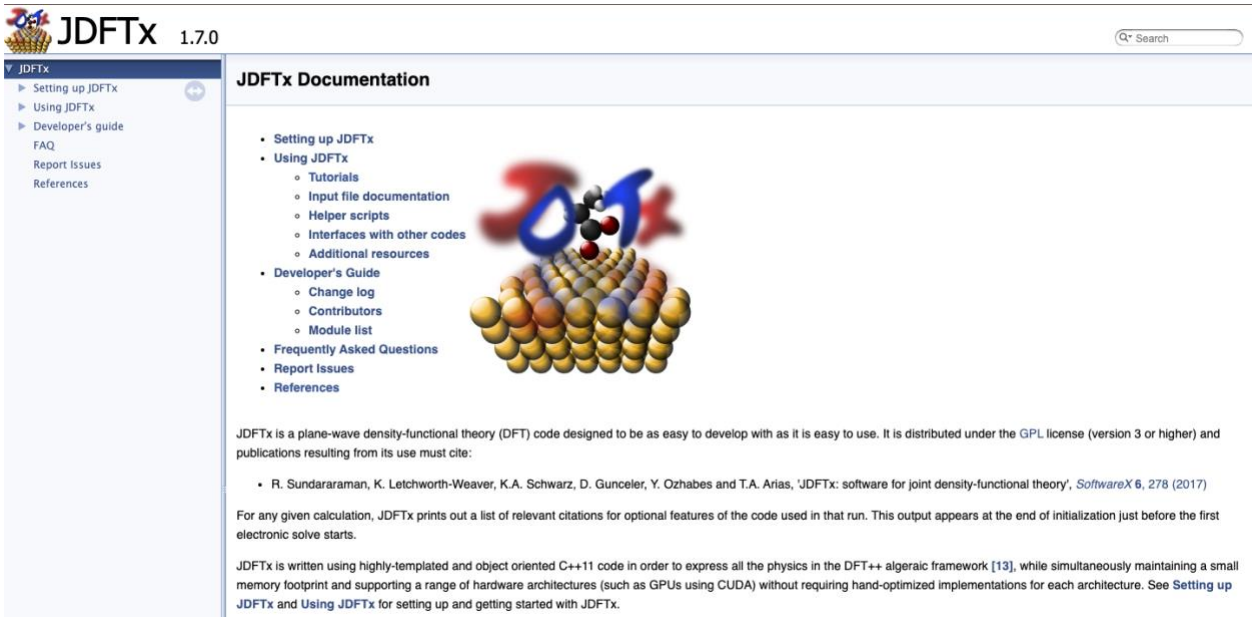
The second line changes the permissions to that folder so that only you can edit the contents of it.

We can also create a folder (`mkdir`) in the home directory (`~`) and go inside by changing directory (`cd`):

```
$ mkdir ~/PARADIM ; cd ~/PARADIM
```

Resources for the JDFTx code

In this summer school, we will use the JDFTx software for joint density functional theory. JDFTx is an open-source software for *ab initio* electronic structure calculations based on plane wave basis sets and pseudopotentials.
The website for JDFTx is http://www.jdftx.org/ This website contains details of its features that go beyond what is covered at this summer school as well as tutorials, some of which are incorporated here as hands-on exercises.



You can also access the list and descriptions of all the parameters that are used in the input files here: http://jdftx.org/Commands.html

Test run

Now we are ready to perform a simple test calculation on the cluster to ensure that everything runs smoothly.

As always, we will perform all our calculations in the `$SCRATCH` folder:
```
$ cd $SCRATCH
```

Let us create a folder specific for the calculations performed today:
```
$ mkdir HandsOn1; cd HandsOn1
```

We will calculate the total energy of silicon and write out its electronic charge density. Silicon has a diamond lattice structure with a cubic lattice constant of 5.43 Angstroms. We can capture the periodicity of the crystal structure using the face-centered cubic Bravais lattice with two silicon atoms as basis. Based on this information, we can now create the input file:

```
$ cat <<EOF > Si.in
lattice face-centered Cubic 5.43
latt-scale 1.88973 1.88973 1.88973  #Convert lattice vectors from Å to Bohr

coords-type lattice        #Specify coordinates based on lattice vectors (fractional)
ion Si 0.00 0.00 0.00  0   #This covers the vertex and face centers of the cube
ion Si 0.25 0.25 0.25  0   #This covers the half-cell body centers

kpoint-folding 8 8 8       #Brillouin zone sampling

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/\$ID_lda.uspp
elec-cutoff 20    #wavefcn (and density) cutoff in Ha
elec-ex-corr lda
electronic-SCF          #Perform a Self-Consistent Field optimization

#Output files
dump-name Si.\$VAR       #Prefix for the output files
dump End ElecDensity     #Write out the final electronic charge density
EOF
```

The JDFTx code is based on pseudopotentials. For these tutorials, we will use the GBRV pseudopotentials https://www.physics.rutgers.edu/gbrv/ that are in the cluster under the directory:

```
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/
```

and we need to write the full path to this directory for the code to find and access these pseudopotential files. Note that all of this is in one line.

To run our jobs, we will use interactive sessions in this tutorial. An interactive session is launched by the following command:

```
$ interact -p defq -n 12 -t 120
```

where the flag `-p defq` specifies which type of nodes will be used for the calculation and we will use the default queue for our calculations; the flag `-n 12` asks for 12 cores from the `defq` nodes. We will increase this number as needed; the flag `-t 120` asks for this interactive session to run for 120 minutes.

Important: We will run this `interact` command ***only once*** at the beginning of each session.

You might need to wait for a few minutes for the interactive session to start. Once it starts, we can run the JDFTx code by the following command:

```
$ mpirun -n 4 jdftx < Si.in > Si.out
```

This command runs `jdftx` executable in parallel using `mpirun` on `-n 4` cores, with the < flag specifying the input file `Si.in` and the output is piped into > flag to write the output to the file `Si.out`. Note that we can ask up to 12 cores here because we have started our interactive session above requesting 12 cores.

This calculation takes about a minute to complete and the output file `Si.out` should look like the following:

```
$ more Si.out

*************** JDFTx 1.7.0  ***************
```

```
Start date and time: Mon Jul 17 17:28:32 2023
Executable jdftx with command-line: -i Si.in
Running on hosts (process indices):  c01 (0-3)
Divided in process groups (process indices):  0 (0)  1 (1)  2 (2)  3 (3)
Resource initialization completed at t[s]:      0.00
Run totals: 4 processes, 24 threads, 0 GPUs


Input parsed successfully to the following command list (including defaults):
…

Dumping 'Si.n' ... done
End date and time: Mon Jul 17 17:28:47 2023  (Duration: 0-0:00:14.17)
Done!
```

Additionally, as we requested in the input file, the electronic charge density is written into the `Si.n` file, which is not human readable.

Note that interactive sessions are typically used for code development to immediately access the outputs and debugging. For most production runs, we instead use a batch queueing system. To place a job in the queue, we prepare a job submission script file that specifies the resources needed for that job and submit it to a scheduler (SLURM in ARCH) that monitors the available resources and allocates them in order of priority. A job script file to run this calculation looks like this:

```
$ cat << EOF > job.sh
#!/bin/bash
#SBATCH --job-name=paradim      #name of the job
#SBATCH --time=02:00:00         #reservation time (hh:mm:ss)
#SBATCH --partition=defq        #name of the queue, defines type of nodes to be used
#SBATCH --nodes=1               #number of nodes
#SBATCH --ntasks-per-node=4     #number of cores/tasks per node
#SBATCH -x c02,c05              #exclude these two nodes which are problematic

#load modules
ml restore
module load jdftx/1.7.0-gcc
ml

#run the calculation
mpirun -n 4 jdftx < Si.in > Si.out
EOF
```

Once we create this file in the same folder as our input file Si.in, we can submit it to the queue:

```
$ sbatch job.sh
```

We can check the status of this job by using the `squeue` command with your username. If you forget your username, you can always check it by:

```
$whoami
paradim-bpamuk

$ squeue -u paradim-bpamuk
             JOBID PARTITION     NAME     USER ST       TIME  NODES NODELIST(REASON)
              3857      defq  paradim paradim-  R       0:02      1 c01
```

This shows that I have the the job with the name `paradim` running (`ST`: status `R`: running, `PD`: pending).

To stop a submitted job from running, you can use `scancel` command with its associated `JOBID`; e.g:

```
$ scancel 3857
```

Here is the resource from the arch website regarding the queueing system for further details: https://www.arch.jhu.edu/support/slurm-queueing-system/

Generating new input files

In this section we will generate new input files by modifying the already existing input file using `vi` instead of the `cat` command we used in the previous section.
First, we will copy the previous file to a new file:

```
$ cp Si.in Si-2.in
```

Then, we will edit this new file using `vi`:

```
$ vi Si-2.in
```

vi opens the file within the same terminal window. There are two modes in vi: 1) command mode 2) insert mode.
We navigate the document using the arrows instead of clicking on it.
We need to hit `i` to go to the insert mode and then we can modify the document.
We need to hit `ESC` to go to the command mode when we are done editing the document.
We need to hit `:wq` to write and quit the document.

Now as an example let us change the kinetic energy cutoff of the plane waves by editing the `elec-cutoff` parameter from `20` Ha to `40` Ha. Details of this will be covered in the theory lectures. We can also change the Brillouin zone sampling by editing the parameter `kpoint-folding` from `8 8 8` to `16 16 16`. This choice of parameters will make the calculations more accurate, and take slightly longer to compute:

```
$ mpirun -n 4 jdftx < Si-2.in > Si-2.out 2>&1 &
```

The last part of this command `2>&1` will write the error messages to the output file instead of the terminal console. You can safely ignore the OpenFabrics related warning and error messages for our purposes. The ampersand `&` symbol runs the command in the background so we can continue using the terminal while that continues to run. When it is done, you will receive the following message on the screen as you hit `ENTER`:

```
$
[1]+  Done                    mpirun -n 4 jdftx < Si-2.in > Si-2.out 2>&1
```

Unix cheat sheet

Below is a summary of useful Unix commands that we will use throughout this class:

`ls`	List content of current folder
`clear`	Clear the screen
`pwd`	Print working directory
`cd thisfolder`	Enter the folder called `thisfolder`
`cd ..`	Go up one folder
`cd`	Go to home directory
`more thisfile`	Show the content of the file called `thisfile`
`rm thisfile`	Remove the file called `thisfile`
`cp file1 file2`	Copy `file1` into `file2`
`mv thisfile thisfolder/`	Move the file `thisfile` into the folder `thisfolder`
`vi thisfile`	Open the file `thisfile` using the Vi text editor

`grep thisword thisfile`	Search for the word `thisword` inside the file `thisfile`
`man thiscommand`	Show the manual page for the command `thiscommand`
`scp myfile username@remoteip:~/`	Copy the file `myfile` to the home directory of user username in the remote computer identified by the IP address `remoteip`.

Copying files from scratch directory to home directory

As best practice, we now move the files from the scratch directory, which typically is treated as a temporary space that is not backed up in the clusters to our home directory, which typically is backed up and is used to keep or backup important calculation results.

```
$ mkdir ~/PARADIM/HandsOn1
$ cp *.in ~/PARADIM/HandsOn1
$ cp *.out ~/PARADIM/HandsOn1
```

Here we first create the HandsOn1 folder under the PARADIM folder that is located in our home directory `~` . Next we copy all the input and output files to this new folder. The asterisk `*` is a wildcard that selects all the files, e.g., `*.in` selects all the files that ends with .in and they all are copied to the HandsOn1 folder.

We encourage you to follow this practice at the end of each hands-on session.

Hands-On 2
Convergence and Scaling

We start the hands-on session by starting an interactive session – only if your previous session has expired –, going to the scratch directory and creating and moving into a folder for this session:

```
$ interact -p defq -n 12 -t 120
$ cd $SCRATCH
$ mkdir HandsOn2; cd HandsOn2
```

Convergence of plane-wave kinetic energy cutoff

It is important to understand the parameters that we are using in our DFT calculations and how those parameters might affect the results that we obtain.

Always make sure that the physical observable that you are interested in calculating is converged with respect to the calculation parameters.

First of these parameters is the plane-wave kinetic energy cutoff, `elec-cutoff`.

Let us now copy the input file from the first hands-on session to edit it accordingly:

```
$ cp ~/PARADIM/HandsOn1/Si.in ./Si_4.in
```

Using vi, we change the `elec-cutoff` parameter from 20 to 4 Ha. (Note that 1 Ha = 2 Ry = 27.2114 eV). Now we can run this calculation:

```
$ mpirun -n 4 jdftx < Si_4.in > Si_4.out 2>&1 &
```

Let us familiarize ourselves with parts of this output file. The first section of the output file prints out all the input variables that we have specified as well as the default values that it uses for the variables that are not specified in the input file. We can confirm that it parses the plane-wave cutoff as `elec-cutoff 4`.

It also writes out `elec-ex-corr lda-PZ`, which means that the exchange and correlation functional that is used here is within the local density approximation (LDA) – one of the standard DFT functionals. This is also reprinted later on in the output file in detail as well:

```
---------- Exchange Correlation functional ----------
Initalized Slater LDA exchange.
Initalized Perdew-Zunger LDA correlation.
```

This choice can be changed by adding this flag into the input file, for example for the Generalized Gradient Approximation (GGA) as implemented in the PBE functional [Perdew, Burke, Ernzerhof, PRL 77, 3865 (1996)]:
`elec-ex-corr pbe`
which is the default choice in the JDFTx code.

You can also read that we have 8 electrons in 4 states in our system. This is consistent with the fact that in silicon we have no spin-polarization, so each spatial wavefunction describes two electrons (one spin-up and one spin-down). From the Periodic Table we know that the silicon atom has 14 electrons. Here we only have the 3s and 3p electrons (4 in total) because the 1*s*, 2*s*, and 2*p* electrons (10 in total) are "fused" together with the nucleus inside the pseudopotential. We say that we have 3*s* and 3*p* **valence states**, and 1*s*, 2*s*, and 2*p* **frozen core states**.

```
nElectrons:   8.000000   nBands: 4
```

After the electronic minimization block, we can read the most important result of this output, that is the total energy of the system:

```
Etot =       -7.9987538962313174
```

This is printed out in units of Ha. This value should be taken with caution: it contains an offset that depends on the chosen pseudopotentials and on some conventions in the code. This energy is not referred to vacuum (as for example when we solve the Schrödinger equation for the hydrogen atom), because there is no vacuum reference when we perform a calculation for an infinitely extended crystal. Consequently, the absolute value of DFT total energy in extended solids is not meaningful; what is meaningful is any total energy difference, where the artificial offset cancels out.

Finally, you can read the duration of the calculation at the end of the file:

```
Dumping 'Si.n' ... done
End date and time: Mon Jul 17 18:26:27 2023  (Duration: 0-0:00:06.85)
Done!
```

Repeat the above steps by setting `elec-cutoff` to 4, 8, 10, 12, 14, 16, 18, 20, 22, 24, 28, 30, 35, 40, 45, 50, 100, 150 in the input file. It is convenient to generate separate input/output files and then search for the total energy, duration time of the calculation in each output file. For this run we will use 4 CPUs, that is `mpirun -n 4 jdftx`.

For example, you can collect the results by creating a text file using `vi ecut.dat` and entering your results one by one. You should be able to construct a file looking like this (omitted rows are for you to fill in):

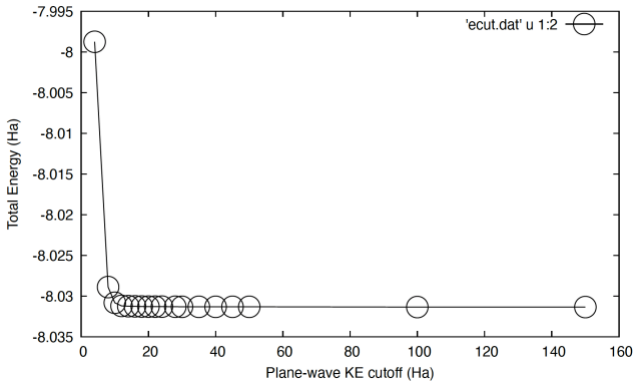
```
$ more ecut.dat
4        -7.9987538962313174
8        -8.0289007040333740
10       -8.0308303527127407
12       -8.0312136141785224
…
100      -8.0313514221773730
150      -8.0313562607444382
```

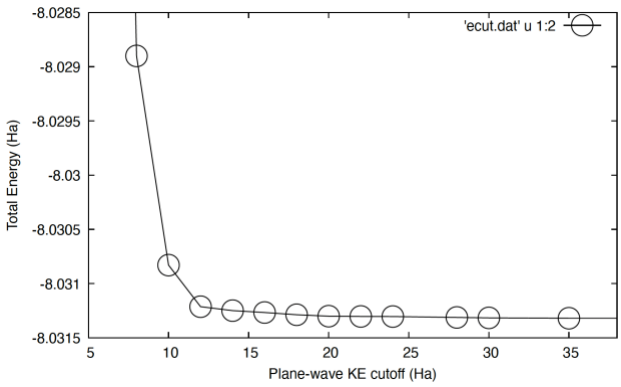
At this point we can analyze our results. To this end we can proceed in one of two ways:

1. Select the content of the file using the left click of the mouse; open an empty text file in your desktop/laptop; paste the selected content using the right click of the mouse. Obviously, this is convenient only when the file consists of a few lines.
2. Transfer the file exercise1.txt using the command `scp` or the program `filezilla`. The syntax for `scp` on Linux/Mac is as follows:
   `scp paradim-bpamuk@paradim.arch.jhu.edu:/scratch/paradim/paradim-bpamuk/myfile ./`
   This command transfers the file "`myfile`" from `$SCRATCH` to the current directory on the local computer. Mind to replace the name "`paradim-bpamuk`" by your own username.

Once the file is in your desktop/laptop, you can plot the data using your favourite software (e.g. gnuplot, Origin, Excel, etc).

For the total energy you should find something like this (left: full range; right: zoom)





The total energy difference between the calculation with plane-wave cutoff of 12 Ha and 150 Ha is < 2 meV/atom, which is quite accurate for DFT calculations. Most quantities that can be computed using DFT **depend critically on this cutoff**, therefore we must always perform this test when running DFT calculations. In general, different properties (total energy, equilibrium structure, band structures, vibrations, etc.) will exhibit a slightly different convergence behavior, therefore it is very important to **always check** that a given property is converged with respect to the planewaves cutoff.

Since the planewaves cutoff is so important, can we not use a very large value to be on the safe side?
The answer is that the higher the cutoff, the more time-consuming the calculation.

To confirm the above point, plot the duration of the calculation time vs. the cutoff using the values.
In this example the runtime is of the order of a few seconds, therefore we can use a very high cutoff without problems. However, in most DFT calculations a careful choice of cutoff can save us weeks of computer time. The longer times required for higher cutoffs relate to the fact that we are performing linear algebra operations using larger arrays to describe the electron wavefunctions.

Note: If you want to plot your data using `gnuplot`, here is the standard command line:

```
$ gnuplot
> plot 'ecut.dat' u 1:2 w lp pt 6 ps 2 lc -1
```

Here `u 1:2` indicates that we want to plot column 1 as x-coordinate and column 2 as y-coordinate. `w lp` means that we want lines and points, `pt` and `ps` are the type and size of the points, respectively, and `lc -1` sets the line color to black.
More information about gnuplot can be found at https://gnuplot.sourceforge.net/demo_5.4/simple.html.

Note: If you do not want to change each input file manually, you can use the following bash script to generate the files:

```
$ cp ~/PARADIM/HandsOn1/Si.in .
$ more ecut.sh
#!/bin/bash
sed "s/20/NEW/g" Si.in > tmp

for DIST in 4 8 10 12 14 16 18 20 22 24 28 30 35 40 45 50 100 150 ; do
      sed "s/NEW/$DIST/g" tmp > Si_$DIST.in
      echo "mpirun -n 4 jdftx < Si_$DIST.in > Si_$DIST.out" >> job.sh
done
$ sh ecut.sh
$ sbatch job.sh
```

Furthermore, you can use the command `grep` to extract the information that you are looking for automatically. For example:

```
$ grep "Etot =" Si_*out
```

Convergence of Brillouin zone sampling

We now want to explore one other convergence parameter of DFT calculations for crystals, the Brillouin zone sampling `kpoint-folding`. This set of parameters specifies the grid used to discretize the Brillouin zone in reciprocal space. These concepts are discussed in theory lectures.
In the input file Si.in we had requested a uniform sampling of Bloch wavevectors k by setting
`kpoint-folding 8 8 8.`
This means that we want to slice the Brillouin zone in an $8 \times 8 \times 8$ grid, and the default is to center the grid at the Γ point.

So now we expect the code to work with exactly $8 \times 8 \times 8 = 512$ k-vectors. You can see that it uses only 29 k-points instead by searching the following block:

```
Folded 1 k-points by 8x8x8 to 512 k-points.

---------- Setting up k-points, bands, fillings ----------
Reduced to 29 k-points under symmetry.
```

The reason for this difference is that many points in our grid are equivalent by symmetry. The code recognizes these symmetries and only performs explicit calculations for the inequivalent points.

Determine how the total energy of silicon varies with the number of k-points, using the same procedure as in previous exercise. Consider the following situations for the input parameters:

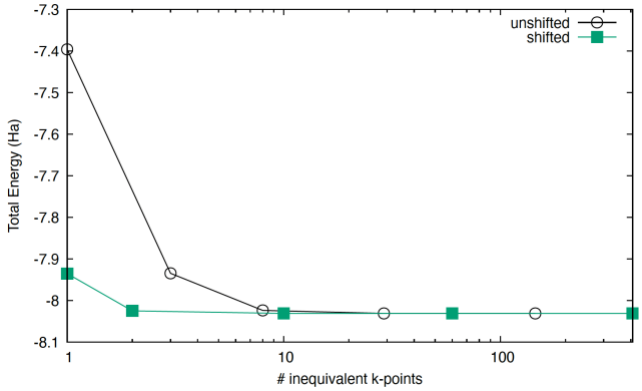
```
kpoint-folding 1 1 1/ 2 2 2/ 4 4 4/ 8 8 8/ 16 16 16
```

For these calculations you can use our 'converged' cutoff `elec-cutoff` = 12 Ha.

In addition, we can shift this grid by half a grid spacing in each direction by adding the following line in the input file:

```
kpoint 0.5 0.5 0.5 #To offset
```

This shift is used because it usually provides a better sampling of the Brillouin zone.



Here we can see that by using the $4 \times 4 \times 4$ grid that is shifted by half a grid spacing, we obtain a total energy which is already very good, only < 1 meV/atom higher than our best value at the shifted $16 \times 16 \times 16$ grid. We also see that the shifted grid converges faster in terms of k-point count than the unshifted grid.

Scaling with the number of processors

The motivation for using parallel computers is that, by splitting the computational tasks among multiple cores, we reduce the total execution time. In this exercise we want to check how the execution time depends on the number of cores used during parallel execution.

Copy a file that has 16 Si atoms corresponding to a 2x2x2 supercell, i.e., computational cells containing multiple primitive cells to perform this exercise:

```
$ cp ~/data/paradim-bpamuk/Si_16.in .
```

Using the input file `Si_16.in`, corresponding to a supercell with 16 silicon atoms, determine the CPU time required to perform run a calculation as a function of the number of cores. To this aim you will need

(i) to execute

```
$ mpirun -n  1 jdftx < Si_16.in > Si_16.out 2>&1 &
$ mpirun -n  2 jdftx < Si_16.in > Si_16.out 2>&1 &
$ …
$ mpirun -n 24 jdftx < Si_16.in > Si_16.out 2>&1 &
```

(ii) to find the CPU time at the end of the output file and collect the data in a table.

Plot the parallel speedup factor for the previous calculation. The parallel speedup is defined as follows:

$$\text{speedup}(n) = \frac{\text{CPU time for 1 core}}{\text{CPU time for } n \text{ cores}}$$

A hypothetical software, perfectly parallelized, should exhibit a speedup equal to the number of cores:

$$\text{ideal speedup}(n) = n$$

Plot the speedup of the previous set of calculations and compare your data with the ideal speedup. What do you think is happening here?

We finish the session by copying the important files that we would like to keep from scratch to home directory:

```
$ mkdir ~/PARADIM/HandsOn2
$ cp Si*.in ~/PARADIM/HandsOn2
$ cp Si*.out ~/PARADIM/HandsOn2
```

Hands-On 3
Equilibrium Structures

We start the hands-on session by going to the scratch directory and creating and moving into a folder for this session:

```
$ cd $SCRATCH
$ mkdir HandsOn3; cd HandsOn3
```

Equilibrium structure of a diatomic molecule

In this exercise we want to obtain the equilibrium structure which gives us the optimal bond length of a diatomic distance, i.e. distance between the two atoms. The main rule that we will follow in this hands-on session is:

Among all possible structures, the equilibrium structure at zero temperature and zero pressure is found by minimizing the DFT total energy.

The DFT total energy can be obtained by `grep "Etot =" *.out` as we practiced in the previous hands-on session. This quantity includes all the terms appearing in the electron-ion Hamiltonian, except the kinetic energy of the ions, therefore it is also referred to as the **potential energy surface**.

Today we will calculate the equilibrium structure of the Cl_2 molecule. It has only 2 atoms, where the structure is specified by the distance between the two atoms, so our goal is to find the Cl-Cl distance with the lowest total energy.

First, we copy an input file from the first hands-on session that we saved under the related folder in our home directory to `here` donated by a dot `.`

```
$ cp ~/PARADIM/HandsOn1/Si.in ./Cl2.in
$ cp $SCRATCH/HandsOn1/job.sh .
```

Then using vi, we can edit the input file such that:

```
$ more Cl2.in
lattice Cubic 20

coords-type cartesian   #Coordinates in terms of absolute atom position in Bohr
ion Cl 0.00 0.00 0.00  0
ion Cl 2.00 0.00 0.00  0

kpoint-folding 1 1 1 #Gamma-centered by default

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/$ID_lda.uspp
elec-ex-corr lda
elec-cutoff 50 200   #Plane-wave KE cutoff for wfcn and charge density in Ha
electronic-SCF       #Perform a Self-Consistent Field optimization

dump-name Cl2.$VAR   #Prefix for the output files
dump End ElecDensity #Write out the final electronic charge density
```

We want to put two atoms in a cubic box with side length of 20 Bohr. As the periodic boundary conditions apply in all our DFT simulations, we want to make sure that the box size is large enough that our molecule does not see the effects of its periodic image. For this purpose, I also encourage you to investigate the effects of Coulomb truncation commands to your results in the future, which we will ignore for simplicity today.

We also switch the coordinate type to Cartesian so that the two Cl atoms are 2 Bohr apart. Because we do not want to study an extended crystal, we do not need Brillouin zone sampling, therefore we also set the k-point folding to include the Γ point only.

Now we can run our calculation by editing number of processors in the `job.sh` file:

```
#SBATCH --ntasks-per-node=6     #number of cores/tasks per node
…
mpirun -n 6 jdftx < Cl2.in > Cl2.out
```

Please verify the convergence of the plane-wave cutoff of 50 Ha is reasonable as compared to the most converged value that you can computationally afford. Make a plot of the total energy vs. plane-wave cutoff.

By observing the output file, we can see the convergence steps of the DFT self-consistent field cycle (SCF). If we look for the total energy in the output file:

```
$ grep Etot Cl2.out
LCAOMinimize: Iter:   0  Etot: -26.7999507131258454  |grad|_K:  1.758e-02  alpha:  1.000e+00
LCAOMinimize: Iter:   1  Etot: -26.8115931446286595  |grad|_K:  1.050e-03  alpha:  9.170e-01  linmin:  4.143e-01  cgtest: -1.010e+00  t[s]:
22.06
LCAOMinimize: Iter:   2  Etot: -26.8115931446286808  |grad|_K:  1.050e-03  alpha:  0.000e+00
LCAOMinimize: Iter:   3  Etot: -26.8116317816717746  |grad|_K:  5.561e-06  alpha:  8.645e-01  linmin:  2.231e-01  cgtest: -6.188e-01  t[s]:
27.93
SCF: Cycle:  0   Etot: -27.618276626028706   dEtot: -8.066e-01   |Residual|: 6.927e-01   |deigs|: 1.554e-01  t[s]:     34.87
SCF: Cycle:  1   Etot: -27.817177035870156   dEtot: -1.989e-01   |Residual|: 4.111e-01   |deigs|: 1.436e-01  t[s]:     39.67
SCF: Cycle:  2   Etot: -27.906651259454044   dEtot: -8.947e-02   |Residual|: 2.385e-01   |deigs|: 1.615e-01  t[s]:     44.61
SCF: Cycle:  3   Etot: -27.941935603798274   dEtot: -3.528e-02   |Residual|: 8.845e-02   |deigs|: 8.294e-02  t[s]:     49.54
SCF: Cycle:  4   Etot: -27.942100204403772   dEtot: -1.646e-04   |Residual|: 6.611e-02   |deigs|: 5.763e-03  t[s]:     54.61
SCF: Cycle:  5   Etot: -27.945675282462226   dEtot: -3.575e-03   |Residual|: 8.895e-03   |deigs|: 2.093e-02  t[s]:     59.67
SCF: Cycle:  6   Etot: -27.945830819223662   dEtot: -1.555e-04   |Residual|: 3.733e-03   |deigs|: 2.618e-03  t[s]:     64.77
SCF: Cycle:  7   Etot: -27.945888859541178   dEtot: -5.804e-05   |Residual|: 3.493e-03   |deigs|: 1.860e-03  t[s]:     69.94
SCF: Cycle:  8   Etot: -27.945931904173406   dEtot: -4.304e-05   |Residual|: 4.054e-03   |deigs|: 8.621e-05  t[s]:     75.12
SCF: Cycle:  9   Etot: -27.945962861113699   dEtot: -3.096e-05   |Residual|: 1.658e-03   |deigs|: 1.207e-03  t[s]:     80.39
SCF: Cycle: 10   Etot: -27.945965306855040   dEtot: -2.446e-06   |Residual|: 2.765e-03   |deigs|: 7.970e-04  t[s]:     85.65
SCF: Cycle: 11   Etot: -27.945971180472732   dEtot: -5.874e-06   |Residual|: 1.066e-03   |deigs|: 3.148e-04  t[s]:     90.80
SCF: Cycle: 12   Etot: -27.945972026058925   dEtot: -8.456e-07   |Residual|: 1.231e-03   |deigs|: 1.076e-04  t[s]:     94.83
SCF: Cycle: 13   Etot: -27.945973820902012   dEtot: -1.795e-06   |Residual|: 4.770e-04   |deigs|: 9.368e-05  t[s]:    100.07
SCF: Cycle: 14   Etot: -27.945974611120008   dEtot: -7.902e-07   |Residual|: 2.921e-04   |deigs|: 3.258e-05  t[s]:    105.28
SCF: Cycle: 15   Etot: -27.945974885247029   dEtot: -2.741e-07   |Residual|: 1.123e-03   |deigs|: 1.379e-04  t[s]:    110.54
SCF: Cycle: 16   Etot: -27.945975674865501   dEtot: -7.896e-07   |Residual|: 3.014e-04   |deigs|: 1.183e-04  t[s]:    115.73
SCF: Cycle: 17   Etot: -27.945975801762874   dEtot: -1.269e-07   |Residual|: 3.535e-04   |deigs|: 6.276e-05  t[s]:    119.69
SCF: Cycle: 18   Etot: -27.945976053908772   dEtot: -2.521e-07   |Residual|: 1.619e-04   |deigs|: 3.425e-05  t[s]:    125.01
SCF: Cycle: 19   Etot: -27.945976214383766   dEtot: -1.605e-07   |Residual|: 7.829e-05   |deigs|: 2.943e-05  t[s]:    130.11
SCF: Cycle: 20   Etot: -27.945976312934050   dEtot: -9.855e-08   |Residual|: 9.360e-05   |deigs|: 1.934e-05  t[s]:    135.36
SCF: Cycle: 21   Etot: -27.945976380262934   dEtot: -6.733e-08   |Residual|: 1.996e-05   |deigs|: 1.147e-05  t[s]:    140.62
SCF: Cycle: 22   Etot: -27.945976423736674   dEtot: -4.347e-08   |Residual|: 1.119e-05   |deigs|: 1.861e-06  t[s]:    145.81
SCF: Cycle: 23   Etot: -27.945976454876423   dEtot: -3.114e-08   |Residual|: 1.718e-05   |deigs|: 1.219e-06  t[s]:    150.98
SCF: Cycle: 24   Etot: -27.945976475886951   dEtot: -2.101e-08   |Residual|: 1.190e-05   |deigs|: 1.132e-06  t[s]:    156.29
SCF: Cycle: 25   Etot: -27.945976488881126   dEtot: -1.299e-08   |Residual|: 1.099e-05   |deigs|: 8.725e-07  t[s]:    161.59
SCF: Cycle: 26   Etot: -27.945976497116902   dEtot: -8.236e-09   |Residual|: 1.958e-05   |deigs|: 9.065e-07  t[s]:    166.86
SCF: Cycle: 27   Etot: -27.945976502488286   dEtot: -5.371e-09   |Residual|: 1.248e-05   |deigs|: 3.219e-06  t[s]:    172.13
     Etot =      -27.9459765024882856
IonicMinimize: Iter:   0  Etot: -27.945976502488286  |grad|_K:  0.000e+00  t[s]:    174.27
```

Note the command `electronic-scf`, which uses a different algorithm from the "variational minimize" to solve the Kohn-Sham equations. Essentially, Kohn-Sham eigenvalues and orbitals are calculated for an input potential, and a new density and potential are constructed from those orbitals. This output potential would in general be different from the input one, so the above step is repeated with successive guesses for the input potential until the input and output potentials become identical (within some threshold). In practice, the guess for the input potential at one step is based on input and output potentials from several previous steps (Pulay algorithm). Note that whether we used SCF or Minimize is not important for the remainder of this tutorial; we can interchangeably use both algorithms for total energy calculations.

We see that the energy reaches its minimum in 27 iterations. It also shows that it stops when the energy difference between the last two iterations is less than 1e-09 Ha. If more accurate calculations are needed, you can lower this convergence tolerance using the keywords in the input file:

```
electronic-scf energyDiffThreshold 1e-09
```

Now we can change the bond length between the atoms and check how the total energy changes. We can use a bash script to automate the procedure:

```
$ cat > bl.sh << EOF
#!/bin/bash
sed "s/2.00/NEW/g" Cl2.in > tmp
for DIST in 2.2 2.4 2.6 2.8 3.0 3.2 3.4 3.6 3.8 4.0 4.2 4.4 4.6; do
    sed "s/NEW/\$DIST/g" tmp > Cl2_\$DIST.in
    echo "mpirun -n 6 jdftx < Cl2_\$DIST.in > Cl2_\$DIST.out " >> job.sh
done
EOF
$ sh bl.sh
```

This script creates multiple input files where only the coordinate of the second atom is changed:

```
$ ls Cl2_*
Cl2_2.2.in  Cl2_2.4.in  Cl2_2.6.in  Cl2_2.8.in  Cl2_3.0.in  Cl2_3.2.in  Cl2_3.4.in  Cl2_3.6.in
Cl2_3.8.in  Cl2_4.0.in  Cl2_4.2.in  Cl2_4.4.in  Cl2_4.6.in
```

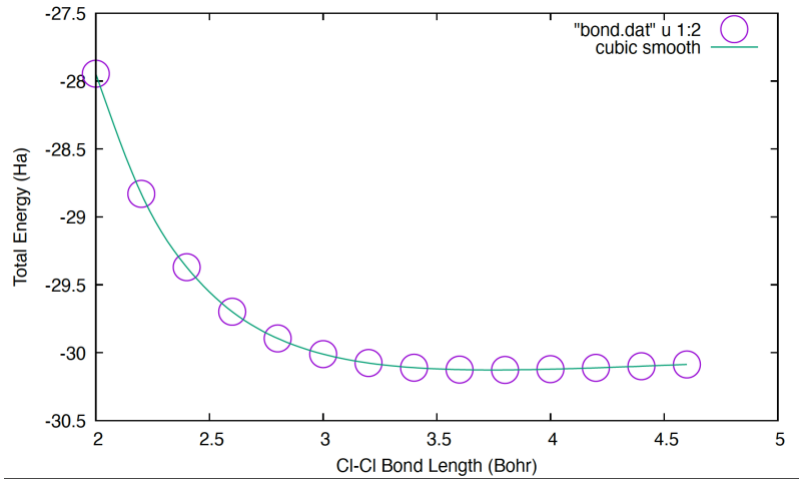
Now we are ready to run the jobs for each of these input files, by adding the following at the end of our `job.sh` file:

```
mpirun -n 6 jdftx < Cl2_2.2.in > Cl2_2.2.out
…
mpirun -n 6 jdftx < Cl2_4.6.in > Cl2_4.6.out
```

Once the calculations are done, we can look at the total energy from the output files:

```
$ grep "Etot =" Cl2*.out
Cl2_2.2.out:     Etot =      -28.8304694768322562
Cl2_2.4.out:     Etot =      -29.3710699724625286
…
Cl2_4.6.out:     Etot =      - 30.0869920854358561
```

Plot the total energy as a function of Cl-Cl bond length as shown below:



In this plot the dots are the calculated datapoints, and the line is a spline interpolation. In `gnuplot` this interpolation is obtained using the flag `smooth csplines` at the end of the `plot` command.

By zooming into this figure, we find that the minimum is located at 3.743 Bohr = 1.98 Å for the bond length. The calculated bond length is slightly shorter than the measured value of 1.99 Å.

- Repeat this exercise using a plane-wave cutoff of 5 Ha instead of 50 Ha. Note that you need to converge the observable that you are interested in calculating in terms of these parameters.

Binding energy of a diatomic molecule

The total energy of Cl_2 at the equilibrium structure can be used to calculate the binding (or dissociation) energy of this molecule:

$$E_{\text{bind}} = E_{\text{Cl}_2} - 2E_{\text{Cl}}$$

where E_{Cl} is the total energy of an isolated Cl atom and E_{Cl_2} is the total energy of the diatomic molecule, each of which can be calculated with DFT.

First, we need to repeat the calculation from the previous section with the correct Cl-Cl bond length in the input file to obtain the total energy of the diatomic molecule, E_{Cl_2}. For this, we need to edit the input file such that the atomic position of the second atom is the optimal bond length we have calculated in the previous section:

```
coords-type cartesian   #Coordinates in terms of absolute atom position in Bohr
ion Cl 0.000 0.00 0.00  0
ion Cl 3.743 0.00 0.00  0
```

This calculation gives the following total energy: $E_{\text{Cl}_2} = -30.127$ Ha.

Next, we calculate the total energy of the isolated atom, E_{Cl}. An important detail here is to notice that the outermost electronic shell of Cl has one unpaired electron, which is also referred to as an open-shell system:

To take this into account, we need to do a **spin-polarized** calculation by including the following in the input file:

```
$ cat > Cl.in << EOF
lattice Cubic 20

coords-type cartesian    #Coordinates in terms of absolute atom position in Bohr
ion Cl 0.00 0.00 0.00  0

kpoint-folding 1 1 1 #Gamma-centered by default

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/\$ID_lda.uspp
elec-ex-corr lda
elec-cutoff 50 200  #Plane-wave KE cutoff for wfcn and charge density in Ha
electronic-SCF      #Perform a Self-Consistent Field optimization
spintype z-spin                    #z-spin = up/dn polarization (non-relativistic spin)
elec-initial-magnetization +1 yes  #yes = hold magnetization fixed, no = optimize it

dump-name Cl.\$VAR    #output file name
dump End ElecDensity #what you would like it to output
EOF
```

The keyword `spintype z-spin` tells the code to differentiate between the number electrons with up and down spin. The keyword `elec-initial-magnetization +1 yes` tells the code that we want 1 unpaired electron, and all other orbitals are doubly occupied.

This calculation gives the following total energy: $E_{\text{Cl}} = -14.998$ Ha.

Using these two results, we get: $E_{\text{bind}} = 0.131$ Ha = 3.57 eV.

This result should be compared to the experimental value of 2.51 eV from

https://en.wikipedia.org/wiki/Bond-dissociation_energy.
We can see that DFT/LDA overestimates the dissociation energy of Cl_2 by about 1.1 eV (the calculated value is about 42% higher than in experiments): interatomic bonding is too strong in LDA.

Equilibrium structure of a bulk crystal

In this section, we want to repeat what we have learned for a bulk crystal. We again consider the silicon crystal from the first hands-on session:

```
cp ~/PARADIM/HandsOn1/Si.in .
```

and we will use the converged parameters for planewaves cutoff and Brillouin-zone sampling that we have calculated in the second hands-on session:

```
$ cat > Si.in << EOF
lattice face-centered Cubic 5.43
latt-scale 1.88973 1.88973 1.88973  #Convert lattice vectors from Angstroms to bohrs

coords-type lattice       #Specify atom coordinates in terms of the lattice vectors (fractional
coordinates)
ion Si 0.00 0.00 0.00  0   #This covers the vertex and face centers of the cube
ion Si 0.25 0.25 0.25  0   #This covers the half-cell body centers

kpoint-folding 4 4 4  #Gamma-centered by default
kpoint 0.5 0.5 0.5    #To offset

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/\$ID_lda.uspp
elec-cutoff 12  #wavefcn (density) cutoff Ha
elec-ex-corr lda
electronic-SCF       #Perform a Self-Consistent Field optimization

dump-name Si.\$VAR   #output file name
dump End ElecDensity #what you would like it to output
EOF
```

In the case of bulk crystals, we often have information about the structure from XRD measurements. This information simplifies considerably the determination of the equilibrium structure. For example, in the case of silicon, the diamond structure is uniquely determined by the lattice parameter a, therefore the minimization of the total energy is really a one-dimensional optimization problem. This situation is therefore similar to the case of the Cl_2 molecule, where we optimized only one parameter, the bond length.

https://en.wikipedia.org/wiki/Monocrystalline_silicon

Later on, we will explore the slightly more complicated situation where we need to decide which one among several possible crystal structures is the most stable.

To find the equilibrium lattice parameter of silicon we perform total energy calculations for a series of plausible parameters. We can generate multiple input files at once by using the following script:

```
$ cp job.sh job_Si.sh
```
Edit the end to remove the Cl-related lines of the job script first.

```
$ cat > alat.sh << EOF
#!/bin/bash
sed "s/5.43/ALAT/g" Si.in > tmp
for ALAT in 5.20 5.25 5.30 5.35 5.40 5.45 5.50 5.55 5.60 5.65; do
```

```
    sed "s/ALAT/\$ALAT/g" tmp > Si_\$ALAT.in
    echo "mpirun -n 6 jdftx < Si_\$ALAT.in > Si_\$ALAT.out " >> job_Si.sh
done
EOF
$ sh alat.sh
```

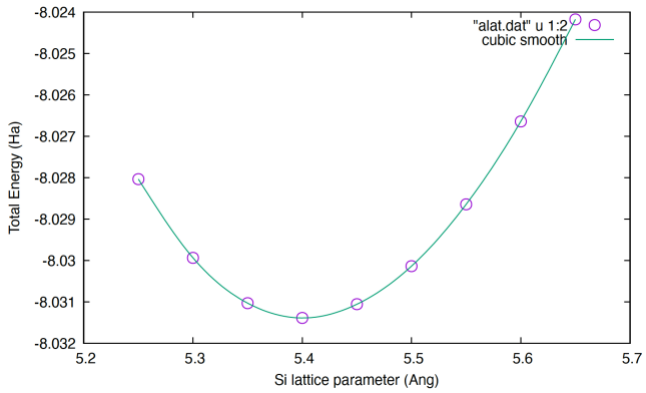
Now we can run each of these calculations by adding the following at the end of the job.sh script:

```
mpirun -n 6 jdftx < Si_5.30.in > Si_5.30.out
…
mpirun -n 6 jdftx < Si_5.65.in > Si_5.65.out
```

After executing these calculations, we can obtain the total energy of each calculation:

```
$ grep "Etot =" Si_*.out
Si_5.25.out:     Etot =       -8.0280374478534409
…
Si_5.65.out:     Etot =       -8.0241794300053506
```

When you plot the total energy as a function of lattice parameter,



you can zoom into the minimum of the plot to find that the equilibrium lattice parameter is $a = 5.4$ Å. Our calculated value is close to the experimental value of 5.43 Å, but DFT/LDA underestimates the measured lattice constant.

Cohesive energy of a bulk crystal

The cohesive energy is defined as the heat of sublimation of a solid into its elements. This quantity is the solid-state counterpart of the molecular dissociation energy that we have seen for Cl2. It quantifies how much energy is needed to break a solid into a set of isolated atoms.
The calculation is almost identical to the case of the dissociation energy of the Cl2 molecule: we need to take the difference between the total energy at the equilibrium lattice parameter and the total energy of each atom in isolation.

For the energy at equilibrium, we just repeat a calculation from the previous exercise, but this time using the equilibrium lattice parameter just determined:

```
lattice face-centered Cubic 5.4
```

This calculation yields:

$E_{\text{bulk}} = 8.031$ Ha

(This is the total energy per unit cell, and each unit cell contains 2 Si atoms)

To determine the total energy of one Si atom in isolation we consider a cubic box with a large lattice parameter and one Si atom, as we did for the Cl atom.
In this case we need to consider that Si has 4 valence electrons in the configuration $3s^2p^2$. According to Hund's rules the spins in the p shell must be arranged as follows:


```
$ cat > silicon.in << EOF
lattice face-centered Cubic 10
latt-scale 1.88973 1.88973 1.88973  #Convert lattice vectors from Angstroms to bohrs

coords-type lattice       #Specify atom coordinates in terms of the lattice vectors (fractional
coordinates)
ion Si 0.00 0.00 0.00  0

kpoint-folding 1 1 1  #Gamma-centered by default

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/\$ID_lda.uspp
elec-cutoff 12  #wavefcn (density) cutoff Ha
elec-ex-corr lda
electronic-SCF       #Perform a Self-Consistent Field optimization
spintype z-spin                   #z-spin = up/dn polarization (non-relativistic spin)
elec-initial-magnetization 2 yes  #yes = hold magnetization fixed, no = optimize it

dump-name Si.\$VAR   #output file name
dump End ElecDensity #what you would like it to output
EOF
```

Here the input variable `elec-initial-magnetization 2 yes` is used to request that the code finds the lowest energy electronic configuration with two electron spins pointing in the same direction, and all other electrons occupy the remaining orbitals in pairs for up and down spin.
The calculation for the isolated atom gives:

$E_{\mathrm{Si}} = 3.815$ Ha

By combining the last two results we obtain:

$E_{\mathrm{cohes}} = \frac{E_{\mathrm{bulk}} - 2E_{\mathrm{Si}}}{2} = 0.200$ Ha $= 5.454$ eV

Note that we divide by 2 since there are 2 Si atoms in each crystal unit cell, so that this quantity gives the cohesive energy per atom.

The measured heat of sublimation of silicon is 4.62 eV, therefore our DFT/LDA calculation overestimates the experimental value by 14%.

The underestimation of lattice parameters and the overestimation of cohesive energies are typical of DFT/LDA. We can summarize these observations by stating that DFT/LDA tends to overbind molecules and solids.

Note: While the DFT/LDA tends to overbind, another extremely popular approximation to the exchange and correlation functional, the PBE functional tends to underbind. For example, DFT/PBE usually yields lattice parameter slightly larger than in experiments (~1%).

We finish the session by copying the important files that we would like to keep from scratch to home directory:

```
$ mkdir ~/PARADIM/HandsOn3
```

```
$ cp *.in ~/PARADIM/HandsOn3
$ cp *.out ~/PARADIM/HandsOn3
$ cp *.sh ~/PARADIM/HandsOn3
```

Hands-On 4
Layered Systems and Visualization

We start the hands-on session by going to the scratch directory, and creating and moving into a folder for this session:

```
$ cd $SCRATCH
$ mkdir HandsOn4; cd HandsOn4
```

Equilibrium structure of diamond

The crystal structure of diamond is almost identical to the one that we used for silicon in HandsOn03. The two important differences are (i) this time we need a pseudopotential for diamond, and (ii) we expect the equilibrium lattice parameter to be considerably smaller than in silicon.

- Create an input file for diamond, diamond.in, by modifying the input file Si.in from HandsOn03. For the time being we can set the lattice parameter to the experimental value, 3.56 Å.

- Calculate the total energy as a function of plane-wave cutoff and plot your data.

You can generate the input files for various cutoff energies manually, or by using the following script (adapted from HandsOn3):

```
$ cat > loop-diamond.sh << EOF
#!/bin/bash
for ECUTWFC in 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 90 100 150 200; do
    sed "s/12/\$ECUTWFC/g" diamond.in > diamond_\$ECUTWFC.in
done
EOF
$ sh loop-diamond.sh
```

- Determine the planewaves cutoff that is required to have the total energy per atom converged to within 25 meV. We can take as 'exact' result the value for the highest cutoff considered, 200 Ha.

- Using the planewaves cutoff determined in the previous step, calculate the total energy of diamond as a function of lattice parameter and plot your data.

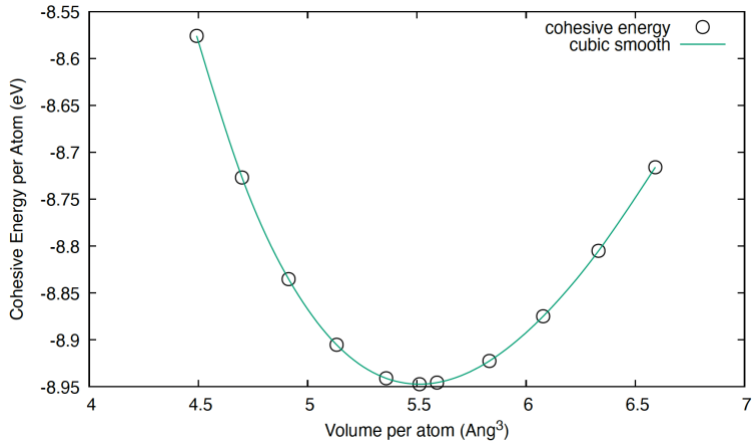
- Determine the equilibrium lattice parameter of diamond and compare your result with the experimental value.

- Using the equilibrium lattice parameter determined in the previous exercise, calculate the cohesive energy of diamond, and compare your value with experiments. For this calculation we use the same strategy employed in HandsOn03 for silicon. The C atom in its ground state has a valence electronic configuration: $2s^{\uparrow\downarrow}\ 2p^{\uparrow}\ ^{\uparrow}\ ^{\ }$, therefore, we need to run a spin-polarized calculation.

- Compare your calculated cohesive energy of diamond with the experimental value, 7.37 eV.

- Plot the cohesive energy vs. volume/atom for all the lattice parameters that you considered. Use units of eV for the energy, and $Å^3$ for the volume. The volume of the unit cell in each calculation can be found in the output file, next to the keyword: `"unit cell volume ="` . You can try using `grep` as in previous exercises.

As a reference, the plot should look like:



Equilibrium structure of graphite

In this exercise we study the equilibrium structure of graphite. We consider the structure of graphite in the Bernal stacking (AB), as obtained from solid state physics textbooks or online databases (we will discuss databases in the following classes):

https://en.wikipedia.org/wiki/Graphite

The unit cell of graphite is hexagonal, with lattice vectors:

$$a_1 = (\ a \quad 0 \quad 0)$$
$$a_2 = (-a/2 \quad a\sqrt{3}/2 \quad 0)$$
$$a_3 = (\ 0 \quad 0 \quad c)$$

(a =2.464 Å and c = 6.712 Å), and with 4 C atoms per primitive unit cell, with fractional coordinates:

$$\mathrm{C}_1 = (\ 0 \quad 0 \quad 1/4)$$
$$\mathrm{C}_2 = (\ 0 \quad 0 \quad 3/4)$$
$$\mathrm{C}_3 = (1/3 \quad 2/3 \quad 1/4)$$
$$\mathrm{C}_4 = (2/3 \quad 1/3 \quad 3/4)$$

- Starting from the input file that you used for diamond in the previous exercise, build an input file for calculating the total energy of graphite, using the experimental crystal structure given above.

Here you will need to pay attention to the keyword `lattice` in the input file. Search for these entries in the documentation page:

http://jdftx.org/CommandLattice.html

Based on this information we must use:

```
lattice Hexagonal 4.6562852 12.683842
```

and the above atomic coordinates with the fractional coordinates with the input keyword:

```
coords-type lattice
```

As a sanity check, if you run a calculation with `elec-cutoff 50` Ha and `kpoint-folding 1 1 1` you should obtain a total energy of -22.429 Ha.

- After performing convergence test with respect to the number of k-points, I found that the total energy is converged to 4 meV/atom when using a shifted 6 x 6 x 2 grid, that is:

```
kpoint-folding 6 6 2  #Gamma-centered by default
kpoint 0.5 0.5 0.5    #To offset
```

Using this setup for the Brillouin-zone sampling, calculate the lattice parameters of graphite a and c at equilibrium. Note that this requires a minimization of the total energy in a two-dimensional parameter space.

For this calculation it is convenient to automatically generate input files as follows, assuming that your input file is called `graphite-tmp.in` and a job submission file `job-tmp.sh` :

- Replace the two values for `lattice Hexagonal 4.6562852 12.683842` by the placeholders `ALAT` and `CLAT`, respectively.

- Rename the input and output file names in the job-tmp.sh file:
  `mpirun -n 4 jdftx < graphite-ALAT-CLAT.in > graphite-ALAT-CLAT.out`

- Create a script graphite.sh with the following content:

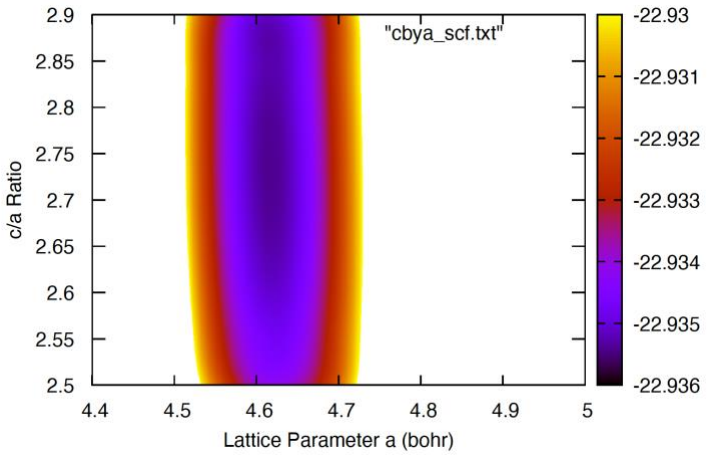
```
#!/bin/bash
for A in 4.4 4.5 4.6 4.7 4.8 4.9 5.0; do
   for CBYA in 2.50 2.55 2.60 2.65 2.70 2.75 2.80 2.85 2.90; do
       C=`echo $A $CBYA | awk '{printf("%5.3f"), $1*$2}'`
       sed "s/ALAT/$A/g" graphite-tmp.in > tmp
       sed "s/CLAT/$C/g" tmp > graphite-$A-$CBYA.in
       sed "s/ALAT/$A/g" job-tmp.sh > jobtmp
       sed "s/CLAT/$CBYA/g" jobtmp > job-$A-$C.sh
       sbatch job-$A-$C.sh
    done
done
```

- By running `sh graphite.sh` you will be able to generate input files for all these combinations of a and c, execute jdftx for each file, and store the output in the corresponding .out files. Note that this will produce 7 x 9 = 63 input files, but the total execution time on 4 cores should be no more than a few minutes.

- At the end you will be able to extract the total energies by using

```
grep "Etot =" graphite-*-*.out > graphite.txt
```

If you plot the total energies that you obtained as a function of a and c/a, (we suggest this for the splines function of the gnuplot to work properly), you should be able to get something like the following:



This plot was generated using the following commands in gnuplot (the file graphite.txt must first be cleaned up in order to obtain only three columns with the values of a, c/a, and energy):

```
set dgrid3d splines 1000,1000
set pm3d map
splot [] [] [:-22.93] "graphite.txt"
```

The '`splines`' keyword provides a smooth interpolation between our discrete set of datapoints. The plotting range along the energy axis is restricted in order to highlight the location of the energy minimum.

Here we see that the energy minimum is very shallow along the direction of the c lattice parameter, while it is very deep along the direction of the lattice parameter a. This corresponds to the intuitive notion that the bonding in graphite is very strong within the carbon planes, and very weak in between planes.

By zooming in a plot like the one above you should be able to find the following equilibrium lattice parameters:

calculation: $a = 2.443$ Å and $c = 6.677$ Å with $c/a = 2.733$

to be compared with the experimental values:

experiment: $a = 2.464$ Å and $c = 6.712$ Å with $c/a = 2.724$

From these calculations we can see that the agreement between DFT/LDA and experiments for the structure of graphite is excellent. This result, however, is somewhat an artifact: most DFT functionals generally overestimate the interlayer distance in graphite due to the inadequate description of van der Waals correlation effect; since LDA generally tends to overbind (as we have seen in all examples considered so far), this overbinding compensates for the lack of van der Waals interactions, and the cancellation of errors leads to a good agreement with experiment.

Visualization of crystal structures

In this exercise, we want to see how the structure of graphite that we have in our output file looks like in a ball-and-stick model. Let us assume that we have run a calculation with our converged lattice parameters with the output name: `graphite-4.617-2.734.out`

Using jdftx script: `createXSF`, we can first convert the output file to .xsf file:

```
$ createXSF graphite-4.617-2.734.out graphite.xsf
```

We will do the visualization on our local computer. For this, `scp` the `graphite.xsf` file to your computer:

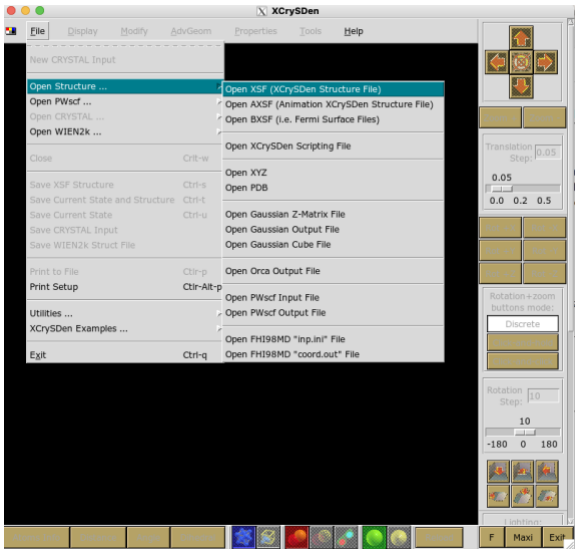
```
$ scp paradim-bpamuk@paradim.arch.jhu.edu:/scratch/paradim/paradim-
bpamuk/HandsOn4/graphite.xsf .
```

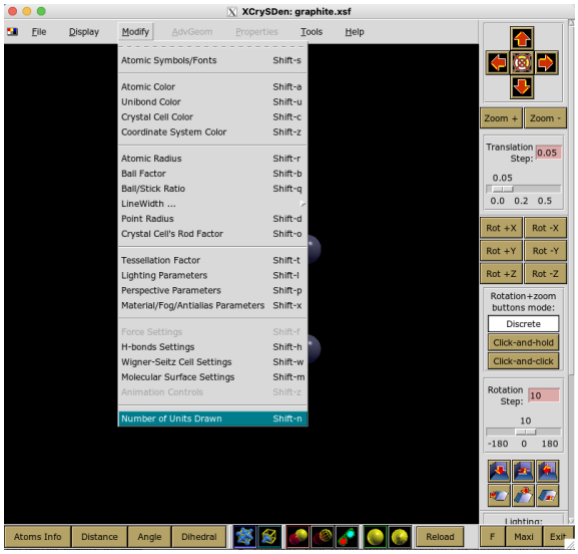
The software `xcryden` can open files with the XSF format to visualize the atomistic structures. General info about this project can be found at
http://www.xcrysden.org.

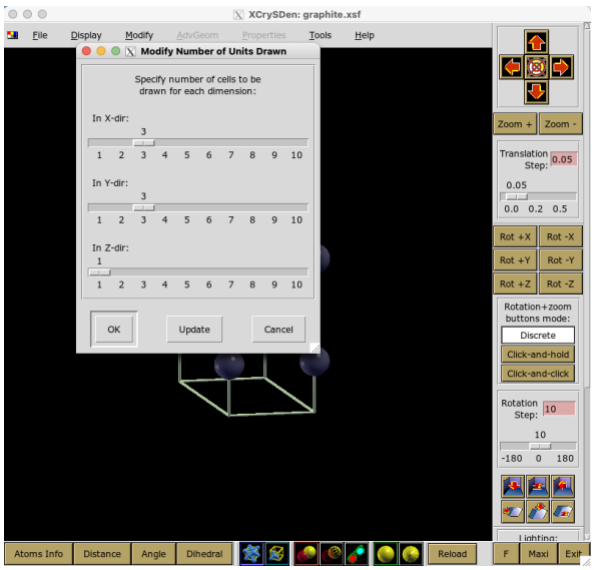
We launch xcryden on the local desktop by typing:

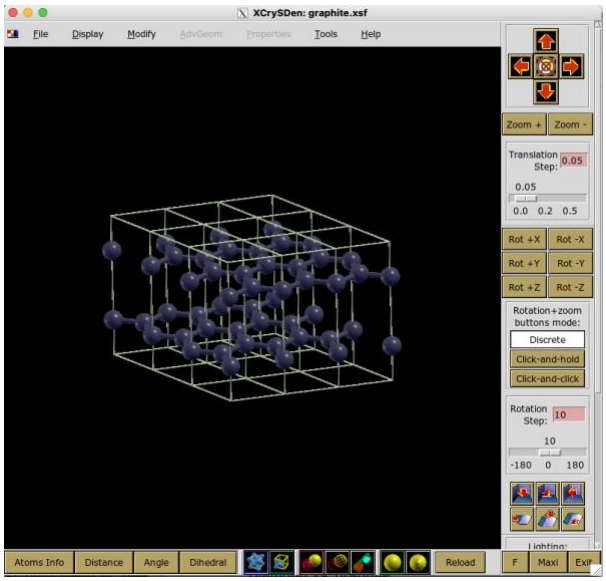
```
$ xcrysden
```

The user interface is very simple and intuitive. The following snapshots will help you to get started with the visualization.


XCrySDen
File
Display
Modify
AdvGeom
Properties
Tools
Help
New CRYSTAL Input
Open Structure ...
Open PWscf ...
Open CRYSTAL ...
Open WIEN2k ...
Close
Save XSF Structure
Save Current State and Structure
Save Current State
Save CRYSTAL Input
Save WIEN2k Struct File
Print to File
Print Setup
Utilities ...
XCrySDen Examples ...
Exit
Open XSF (XCrySDen Structure File)
Open AXSF (Animation XCrySDen Structure File)
Open BXSF (i.e. Fermi Surface Files)
Open XCrySDen Scripting File
Open XYZ
Open PDB
Open Gaussian Z-Matrix File
Open Gaussian Output File
Open Gaussian Cube File
Open Orca Output File
Open PWscf Input File
Open PWscf Output File
Open FHI98MD "inp.ini" File
Open FHI98MD "coord.out" File
Translation Step:
Rotation+zoom buttons mode:
Discrete
Rotation Step:
F
Maxi
Exit

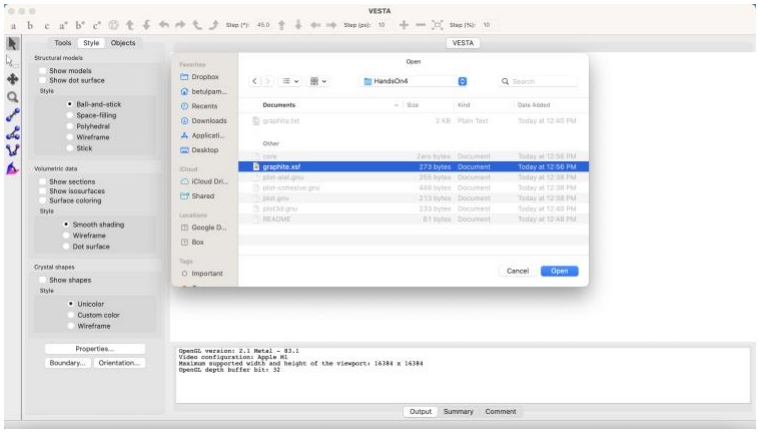

XCrySDen: graphite.xsf
File
Display
Modify
AdvGeom
Properties
Tools
Help
Atomic Symbols/Fonts
Atomic Color
Unibond Color
Crystal Cell Color
Coordinate System Color
Atomic Radius
Ball Factor
Ball/Stick Ratio
LineWidth ...
Point Radius
Crystal Cell's Rod Factor
Tessellation Factor
Lighting Parameters
Perspective Parameters
Material/Fog/Antialias Parameters
Force Settings
H-bonds Settings
Wigner-Seitz Cell Settings
Molecular Surface Settings
Animation Controls
Number of Units Drawn
Zoom +
Zoom -
Translation Step:
Rot +X
Rot -X
Rot +Y
Rot -Y
Rot +Z
Rot -Z
Rotation+zoom buttons mode:
Discrete
Click-and-hold
Click-and-click
Rotation Step:
Atoms Info
Distance
Angle
Dihedral
Reload
F
Maxi
Exit

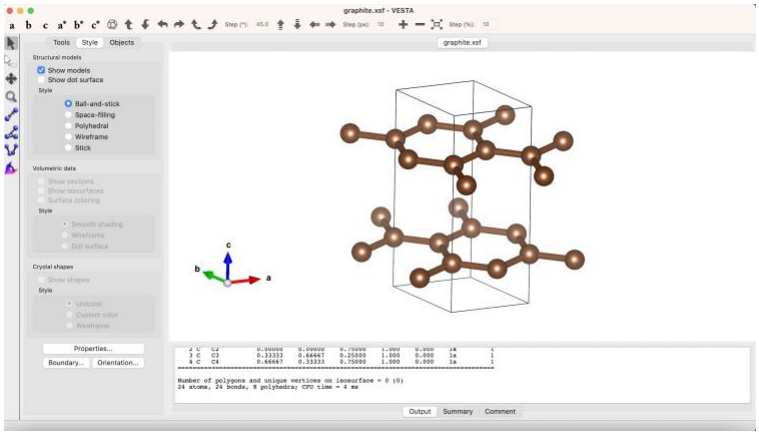

XCrySDen: graphite.xsf
File
Display
Modify
AdvGeom
Properties
Tools
Help
Modify Number of Units Drawn
Specify number of cells to be drawn for each dimension:
In X-dir:
In Y-dir:
In Z-dir:
OK
Update
Cancel
Zoom +
Zoom -
Translation Step:
Rot +X
Rot -X
Rot +Y
Rot -Y
Rot +Z
Rot -Z
Rotation+zoom buttons mode:
Discrete
Click-and-hold
Click-and-click
Rotation Step:
Atoms Info
Distance
Angle
Dihedral
Reload
F
Maxi
Exit



Another software that you might want to consider for rendering structures is `Vesta`. In order to install Vesta on your desktop/laptop, you can visit the following page:
https://jp-minerals.org/vesta/en/download.html

Once installed, you can simply load the xsf file by File → Open → Select `graphite.xsf` file:





As an exercise, in Vesta, go to:
Edit → Edit Data → Unit cell → Transform
and generate a 3x3x1 supercell of graphite as we did with xcrysden.

Diamond vs. graphite

- Calculate the cohesive energy per atom of diamond and graphite, using the optimized lattice parameters as determined in the previous exercises.
- Based on your calculations, which carbon allotrope is more stable at ambient conditions, diamond or graphite?
- Compare your result with experiments by looking up online the cohesive energies of diamond and graphite, e.g., from: https://doi.org/10.1063/1.4867544

Hands-On 5
Automatic Structure Optimization

We start the hands-on session by going to the scratch directory, and creating and moving into a folder for this session:

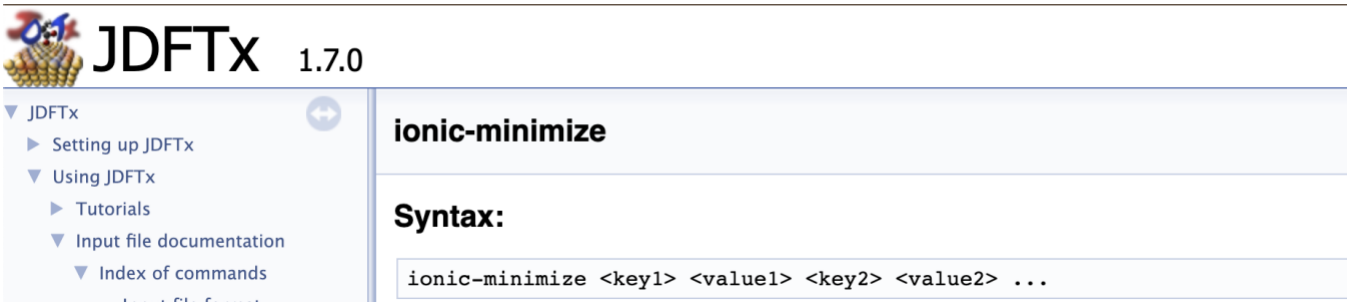
```
$ cd $SCRATCH
$ mkdir HandsOn5; cd HandsOn5
```

In this session we will learn how to find the ground-state structure using automatic optimization of atomic coordinates and crystal lattice.

Automatic optimization of atomic coordinates

In Hands-On 3 we discussed how to calculate the potential energy surface of a molecule, and how to determine equilibrium structures by locating the minima of that surface. In this section we consider an alternative route for finding the equilibrium geometry of Cl_2.

Now let us look at the documentation page for the JDFTx website. Under the input file documentation, we find the ionic-minimize command:



Now we add the ionic-minimize command at the end of our input file. In principle we could perform calculations without specifying these parameters; in this case the code will use some preset default values. For the sake of completeness let us specify some rather stringent criteria in the input file:

```
ionic-minimize \
    nIterations 10 \
    energyDiffThreshold 1e-6 \
    knormThreshold 1e-4  #Threshold on RMS cartesian force
```

which tells the code that we want 10 iterations of the atomic positions as well as the threshold on the total energy convergence and atomic forces. Note that one of the default values of `ionic-minimize` (and `lattice-minimize`) is "`nIterations 0 \`". So, it will end up simply doing a total energy calculation unless otherwise specified.

This choice instructs to automatically determine the equilibrium structure, starting from the given coordinates. In practice the code calculates the forces acting on the ions and updates the ionic positions in such a way as to minimize those forces. The equilibrium configuration will correspond to the situation where all forces are smaller than a given threshold, and where the total potential energy has changed less than a given threshold with respect to the previous iteration.

As a starting point for the atomic coordinates let us use a very large Cl-Cl separation:

```
coords-type cartesian     #Specify atom coordinates in terms of absolute atom position
in Bohr
ion Cl 0.00 0.00 0.00  1
ion Cl 5.00 0.00 0.00  1
```

The label 1 at the end of the atomic coordinates instructs the code to let the atoms move, while 0 does not allow the movement of the atoms.

Let us also ask the code to print out the converged atomic positions and other things that we might need to restart the calculation into separate files:

```
dump End State       #Output State i.e. everything needed to resume the calculation
```

which will create the `Cl2.ionpos` file with the information of the final atomic coordinates after 10 iterations.

Now we can run the calculation with the following edits in the job script file:

```
…
#SBATCH --ntasks-per-node=6     #number of cores/tasks per node
…
mpirun -n 6 jdftx < Cl2.in > Cl2.out
```

While the calculation is running, you can check the output file with the following command:

```
$ tail -f Cl2.out
```

The command `tail` shows you the last 10 lines of any text file. By using the flag `-f` this command 'follows' the file, so it keeps printing the lines whenever the file is modified.

In order to search for a word in vi we simply press `/` and type the word. We search for 'Forces' and obtain:

```
# Forces in Cartesian coordinates:
force Cl   0.061674768915473  -0.000000000000000   0.000000000000000 1
force Cl  -0.061674768915473  -0.000000000000000   0.000000000000000 1
```

These lines are telling us that, in the initial configuration, the two Cl atoms experience forces directed along the Cl-Cl axis and pointing towards the other Cl atom. This was to be expected since we started with the atoms at a distance much larger than the equilibrium bond length. In vi, when we hit `n` to search for the "next" occurrences of the "Forces", we find the updated atomic positions for the next iteration and the forces on the atoms at this new position:

```
# Ionic positions in cartesian coordinates:
ion Cl   0.431723382408312   0.000000000000000   0.000000000000000 1
ion Cl   4.568276617591688   0.000000000000000   0.000000000000000 1

# Forces in Cartesian coordinates:
force Cl   0.051560049348982  -0.000000000000000   0.000000000000000 1
force Cl  -0.051560049348981   0.000000000000000   0.000000000000000 1
```

Clearly the atoms are being displaced towards each other. At the end of the iterations, we can see something like the following:

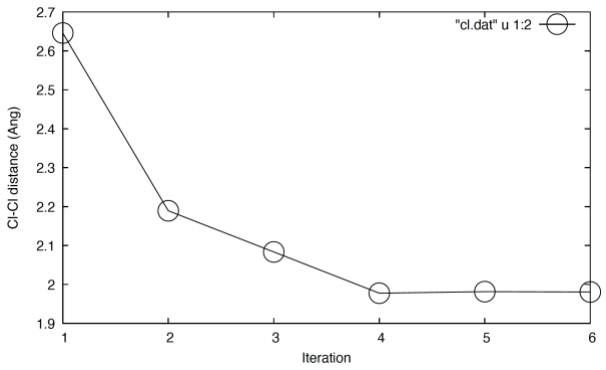
```
# Ionic positions in cartesian coordinates:
ion Cl   0.628583764855240   0.000000000000000   0.000000000000000 1
ion Cl   4.371416235144760   0.000000000000000   0.000000000000000 1

# Forces in Cartesian coordinates:
force Cl  -0.000049475558772  -0.000000000000000   0.000000000000000 1
force Cl   0.000049475558770   0.000000000000000   0.000000000000000 1
```

Here we see that the forces are practically vanishing, therefore we reached the equilibrium configuration. The total energy at equilibrium is -30.1274486305767333 Ha and the bond length is 3.743 Bohr = 1.98 Å. These values are in agreement with what we had found in HandsOn3 by explicitly looking for the minimum of the potential energy surface.

If we want to see how the atomic coordinates evolved towards the equilibrium configuration, we can simply issue:

```
$ grep "ion Cl" Cl2.out
```

From the atomic positions at each iteration, determine the Cl-Cl distance in Å and plot the distance vs. iteration number. You should obtain something like the following:



Automatic optimization of atomic coordinates and unit cell

In addition to the optimization of atomic coordinates, it is also possible to optimize the vectors of the primitive unit cell. In order to simplify life in the later stages of this calculation, we modify the input file such that the atomic coordinates and lattice parameters are read from external files.

We create a file for the lattice parameters:

```
$ more graphite.lattice
lattice \
   4.000000000000000   -2.000000000000000    0.000000000000000  \
   0.000000000000000    3.464101615137755    0.000000000000000  \
   0.000000000000000    0.000000000000000    8.000000000000000
```

and we create a file for the atomic positions:

```
$ more graphite.ionpos
# Ionic positions in lattice coordinates:
ion C   0.000000000000000   0.000000000000000   0.250000000000000 1
ion C   0.000000000000000   0.000000000000000   0.750000000000000 1
ion C   0.333333333333333   0.666666666666667   0.250000000000000 1
ion C   0.666666666666667   0.333333333333333   0.750000000000000 1
```

Finally, we modify the input file to read these files:

```
$ more graphite.in
include graphite.lattice
include graphite.ionpos

kpoint-folding 6 6 2  #Gamma-centered by default
kpoint 0.5 0.5 0.5    #To offset
```

```
#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/$ID_lda.uspp
elec-cutoff 50  #wavefcn (density) cutoff Ha
elec-ex-corr lda

lattice-minimize nIterations 100  #Lattice and ionic geometry optimization

dump-name graphite.$VAR   #Output file name
dump End ElecDensity      #Output electronic charge density
dump End State            #Output the end state
dump Ionic State          #Output state every geometry (ionic/lattice) step
dump Ionic Lattice IonicPositions #Output every geometry (ionic/lattice) step
```

The command `lattice-minimize` instructs the code to perform lattice and ionic geometry optimization and asks the code to perform 100 iterations and stop if not converged. Now we are ready to run the calculation with the following in our job script file:

```
…
#SBATCH --ntasks-per-node=6     #number of cores/tasks per node
…

mpirun -n 6 jdftx < graphite.in >> graphite.out
```

At the end of the run, we can look inside the output file using vi and search for the following words:

```
/ Stress
```

We will see something like:

```
# Stress tensor in Cartesian coordinates [Eh/a0^3]:
[   -0.0176104            0            0  ]
[            0   -0.0176104            0  ]
[            0            0  -0.00682177  ]
```

This indicates that, as expected, in the first iteration the system is under very high pressure. Following this initial iteration, the code modifies the lattice vectors in the direction of lower pressure.

We can note that in this case the forces acting on each atom are all vanishing by symmetry:

```
# Forces in Lattice coordinates:
force C  -0.000000000000000   0.000000000000000   0.000000000000000 1
force C   0.000000000000000   0.000000000000000   0.000000000000000 1
force C  -0.000000000000000   0.000000000000000   0.000000000000000 1
force C  -0.000000000000000   0.000000000000000   0.000000000000000 1
```

As a result, this procedure will not modify the Wyckoff positions of the 4 C atoms in the unit cell, and the file the `graphite.ionpos` remains the same throughout the calculations.

After about 10 iterations, the calculation stops at lattice parameters of calculation: $a = 2.439$ Å and $c = 5.401$ Å , with the warning that the strain on the system is too big, so we need to resubmit the calculation.

Also note that from this calculation we have obtained a c/a ratio which is much smaller than the one determined in Hands-On 4 by studying the potential energy surface (2.214 here vs. 2.733 in Hands-On 4). The interlayer separation is approximately 19% shorter in the present calculation. This result is a calculation artifact. What is happening here is that the code modifies the crystal structure so as to minimize the energy. Now, the Hamiltonian describing the system is expressed in a basis of planewaves,

and the wavevectors of these planewaves along the c axis are multiples of $2\pi/c$. If, during the optimization, the c parameter undergoes a significant change (as it is the case here, since we start from $c/a = 2$ and we end up with $c/a = 2.733$), then we are effectively reducing our planewaves cutoff at each iteration. As a result, the calculation becomes less and less accurate. This indicates that the structure is not yet fully optimized. In order to avoid this problem, we should run a new calculation, starting from the latest lattice parameters.
Since we chose "`dump-name graphite.$VAR`", the file `graphite.lattice` is updated with the latest lattice parameters. So, when we rerun the calculation, we do not need to modify the input file anymore.

Now we can rerun the calculation:

```
$ mpirun -n 6 jdftx < graphite.in >> graphite.out
```

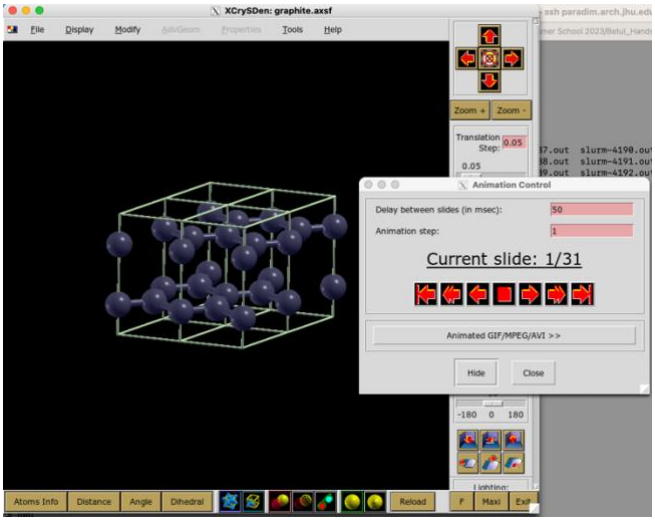
with `>>` flag to append to the `graphite.out` file to keep track of all the previous steps.

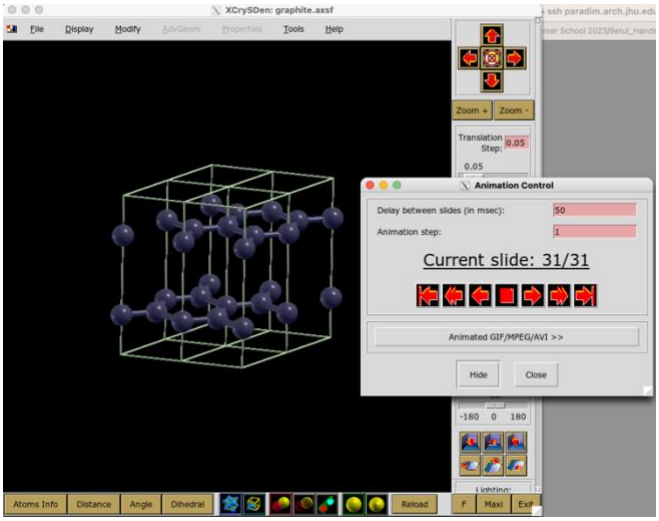
Perform a structural optimization for graphite once again, this time restarting from the lattice parameters $a = 2.439$ Å and $c = 5.401$ obtained at the previous step, as updated in the `graphite.lattice` file. Compare your new optimized lattice parameter with the results of Hands-On 4.

You can visualize how the structure changes at each iteration with `xcrysden` by creating an AXSF, animation xsf file:

```
$ createXSF graphite.out graphite.axsf Animated
```

and then open it File → Open Structure → Open AXSF





Hands-On 6
Elastic Constants

We start the hands-on session by going to the scratch directory, and creating and moving into a folder for this session:

```
$ cd $SCRATCH
$ mkdir HandsOn6; cd HandsOn6
```

In this session we will learn how to calculate elastic constants. We will start with the simple case of diamond, and then we will try to set up an entirely new calculation on $SrTiO_3$ (STO). For STO we will use the Materials Project database to find the initial geometry.

Elastic constants of diamond

We want to calculate the elastic constants of diamond. As we have seen in the lectures, for a cubic system like diamond there are only three independent elastic constants, namely C_{11}, C_{12} , and C_{44}.

Isotropic deformation. An isotropic deformation of the diamond structure corresponds to a uniform stretch of the lattice vectors:

$$a_i' = (1+\eta)a_i \text{ for } i = 1,2,3$$

The resulting change in the total potential energy from the equilibrium value U_0 is:

$$\frac{U-U_0}{\Omega} = \frac{3}{2}(C_{11}+2C_{12})\eta^2 \qquad (6.1)$$

The calculation of U and U_0can be performed by using the following input file for diamond, which has been adapted from Hands-On 4:

```
$ more diamond_3.533.in
latt-scale 6.676 6.676 6.676
lattice \
         -0.50    0.00   -0.50  \
          0.00    0.50    0.50  \
          0.50    0.50    0.00

coords-type lattice    #Specify atom coordinates in terms of the lattice vectors (fractional)
ion C 0.00 0.00 0.00  1
ion C 0.25 0.25 0.25  1

kpoint-folding 4 4 4  #Gamma-centered by default
kpoint 0.5 0.5 0.5    #To offset

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/$ID_lda.uspp
elec-cutoff 50          #wavefcn (density) cutoff Ha
elec-ex-corr lda
electronic-SCF          #Perform a Self-Consistent Field optimization

ionic-minimize \
    nIterations 10 \
    energyDiffThreshold 1e-6 \
    knormThreshold 1e-4  #Threshold on RMS cartesian force
```

```
dump-name diamond.$VAR   #output file name
dump End ElecDensity     #Output electronic density
dump End State           #Output State i.e. everything needed to resume the calculation
```

In this version of the input file, we are specifying that the unit cell vectors are given manually:

```
lattice \
         -0.50    0.00   -0.50  \
          0.00    0.50    0.50  \
          0.50    0.50    0.00
```

and these vectors are provided in units of the lattice parameter given by

```
latt-scale 6.676 6.676 6.676
```

The lattice parameters are printed also in the output file, note that each column represent a lattice vector in JDFTx:

```
R =
[        -3.338             0        -3.338  ]
[             0         3.338         3.338  ]
[         3.338         3.338             0  ]
```

And we use `ionic-minimize` to relax the system, so the atoms can find their equilibrium positions when we deform the unit cell.

By running the above calculation, we obtain the total energy in the ground state:

$U_0 = -11.4679253259808114$ Ha

Furthermore, we can read the volume of the unit cell by searching for: `/ volume`

This gives $\Omega = 74.3856$ bohr^3.

Now we can modify this input file in order to establish an isotropic deformation with $\eta = 0.002$ (let us call the new file diamond-1.in)

```
$ more diamond-1.in
…
lattice \
           -0.501    0.000   -0.501  \
            0.000    0.501    0.501  \
            0.501    0.501    0.000
…
```

With this new input file, we find the total energy:

$U = -11.4679037091361771$ Ha

Using Eq. (6.1) with $\eta = 0.002$ we obtain:

$C_{11} + 2C_{12} = 0.048434$ Ha/bohr$^3 = 1425$ GPa where 1 Ha/bohr^3 = 29421 GPa

Tetragonal deformation. From the lectures, we have the relation:

$$\frac{U - U_0}{\Omega} = 3(C_{11} - C_{12})\eta^2 \qquad (6.2)$$

and the tetragonal distortion of the unit cell can be realized by considering an expansion $(1+\eta)$ along *x* and *y*, and a contraction $(1-2\eta)$ along *z*. We copy the input file diamond-1.in into diamond-2.in and we modify this new file as follows:

```
lattice \
        -0.501    0.000   -0.501  \
         0.000    0.501    0.501  \
         0.498    0.498    0.000
```

This calculation gives:

$U = -11.4678965436591813$ Ha

Using Eq. (6.2) with $\eta = 0.002$ and remembering $U_0 = -11.4679253259808114$ Ha, we obtain:

$C_{11} - C_{12} = 0.032245$ Ha/bohr$^3 = 949$ GPa

By combining the two relations for C_{11} and C_{12} we obtain:

$C_{11} = 1108$ GPa and $C_{12} = 159$ GPa.

The corresponding experimental values are $C_{11} = 1482$ GPa, $C_{12} = 124$ GPa, from McSkimin & Andreatch, J. Appl. Phys. 43, 2944 (1972).

Note. These calculations are not accurate, and by refining our setup we can obtain better agreement with experiment. In particular, we are determining elastic constants using only 2 calculations in each case. Since elastic constants are second derivatives of the total energy, a much more accurate approach is to evaluate such derivatives using 3 total energy calculations. For more details, see the explicit calculation of C_{44} below.

Trigonal deformation. From the lectures, we have the relation:

$$\frac{U - U_0}{\Omega} = \frac{1}{2} C_{44} \eta^2 \tag{6.3}$$

A trigonal distortion of the unit cell can be realized by considering the following distortion of the lattice parameters:

$u_1 = u_2 = u_3 = u_4 = u_5 = 0$ and $u_6 = \eta$

We start from the undistorted lattice parameters:

$a_{1x} = -\frac{a}{2}$
$a_{1y} = 0$
$a_{1z} = \frac{a}{2}$
$a_{2x} = 0$
$a_{2y} = \frac{a}{2}$
$a_{2z} = \frac{a}{2}$
$a_{3x} = -\frac{a}{2}$
$a_{3y} = \frac{a}{2}$
$a_{3z} = 0$

Since $u_6 = 2\epsilon_6$ and $\epsilon_6 = \epsilon_{xy} = \epsilon_{yx}$, we can write:

$a'_{1x} = a_{1x} + \epsilon_{xy} a_{1y} = -\frac{a}{2}$

$a'_{1y} = a_{1y} + \epsilon_{yx} a_{1x} = -\frac{\eta}{2}\frac{a}{2}$
$a'_{1z} = a_{1z} = \frac{a}{2}$
$a'_{2x} = a_{2x} + \epsilon_{xy} a_{2y} = \frac{\eta}{2}\frac{a}{2}$
$a'_{2y} = a_{2y} + \epsilon_{yx} a_{2x} = \frac{a}{2}$
$a'_{2z} = a_{2z} = \frac{a}{2}$
$a'_{3x} = a_{3x} + \epsilon_{xy} a_{3y} = -\frac{a}{2}\left(1 - \frac{\eta}{2}\right)$
$a'_{3y} = a_{3y} + \epsilon_{yx} a_{3x} = \frac{a}{2}\left(1 - \frac{\eta}{2}\right)$
$a'_{3z} = a_{3z} = 0$

These relations indicate that, if we set $\eta = 0.002$, the input file must be modified as follows:

```
lattice \
          -0.5000    0.0005   -0.4995  \
          -0.0005    0.5000    0.4995  \
           0.5000    0.5000    0.0000
```

The calculation with this input file gives:

$U = -11.4679341426679802$ Ha

Using Eq. (6.3) with $\eta = 0.002$ and remembering $U_0 = -11.4679253259808114$ Ha, we obtain a negative value.

If U_0 is not exactly the energy minimum (e.g., due to incomplete numerical convergence), we can still correctly capture the curvature C_{44} using three total energy evaluations.

As in the case of cubic and tetragonal deformations, we are using the total energy computed for two configurations to evaluate a second derivative. To better understand what is happening, we can consider the Taylor expansion of the energy in terms of the strain, from (6.3):

$$U = U_0 + \frac{1}{2}\Omega C_{44}\eta^2$$

In our calculation we tried to determine C_{44} by simply inverting this relation. A more acurate approach consists of taking the second derivative of U with respect to η:

$$\frac{\partial^2 U}{\partial \eta^2} = \Omega C_{44}$$

Using central finite differences, this expression can be evaluated numerically as:

$$C_{44} = \frac{1}{\Omega}\frac{U(+\eta) - 2U_0 + U(-\eta)}{\eta^2} \tag{6.4}$$

Using this equation, the more accurate shear constant is found to be:

$C_{44} = 0.01822$ Ha/bohr^3 $= 536$ GPa

which is now more comparable to the corresponding experimental value is $C_{44} = 578$ GPa [McSkimin & Andreatch, J. Appl. Phys. 43, 2944 (1972)].

- Calculate the elastic constants C_{11}, C_{12}, and C_{44} of diamond, by following the steps illustrated above. Determine the elastic constants using two different sets of calculations: one for $\eta = 0.002$, and one for $\eta = 0.001$.

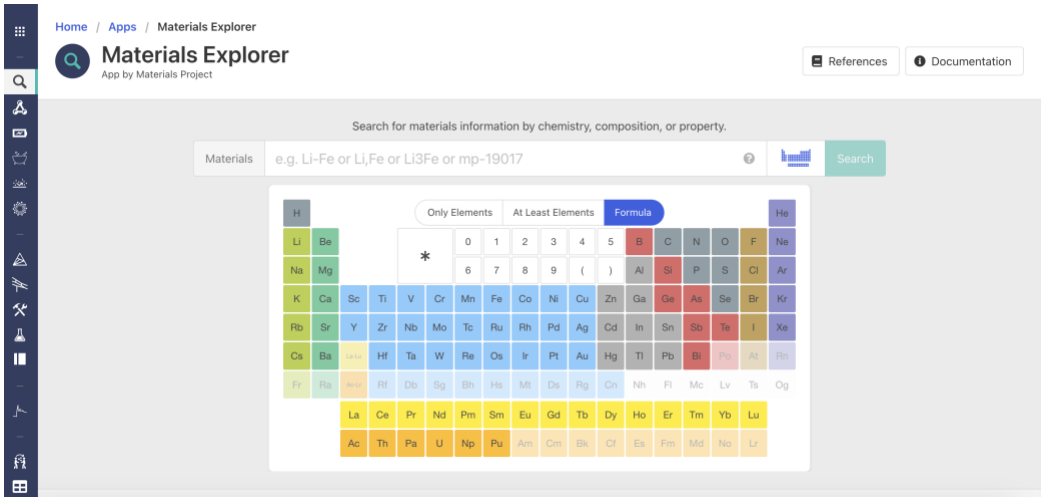
The bulk modulus B is a measure of the resistance of a material to hydrostatic compression. This quantity can be obtained from the elastic constants as:

$$B = \frac{1}{3}(C_{11} + 2C_{12})$$

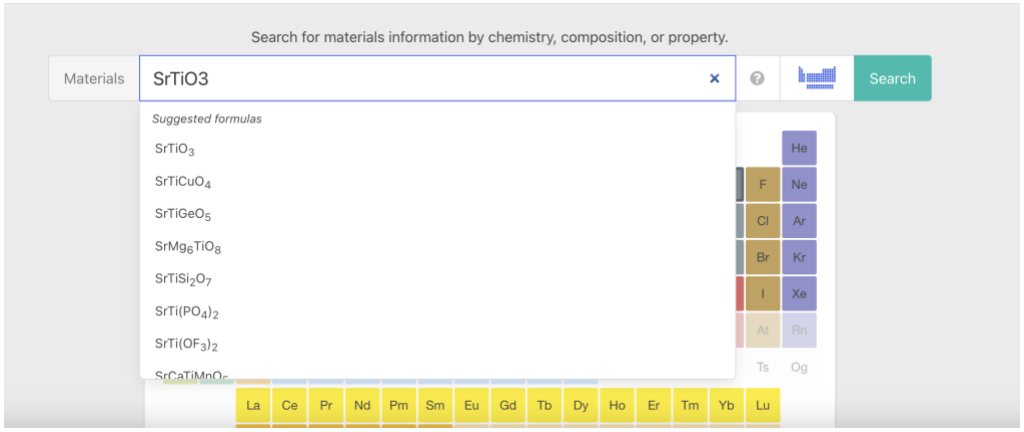
- Calculate the bulk modulus of diamond, using the data obtained in the previous step, and compare your result with the experiment.
- Investigate the sensitivity of the calculated bulk modulus of diamond to the choice of η, by repeating the calculations for $\eta = 0.1$, 0.001, and 0.0001.
- Repeat the calculation of the bulk modulus B using three evaluations of the total energy, along the same lines as in (6.4).

Elastic constants of STO

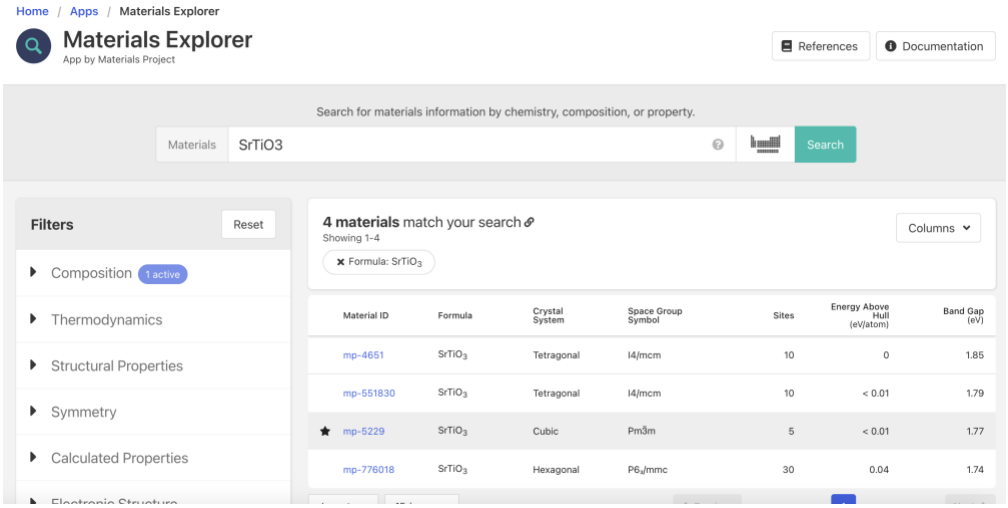
In this exercise we want to set up a simple input file to study $SrTiO_3$. In order to find an initial guess for the unit cell and atomic coordinates, we search the Materials Project database. In order to search the Materials Project we need to create an account. To this aim, please go to https://next-gen.materialsproject.org/ and click on "Login or Register". The registration process only requires an email address and a password, this should take less than a minute to complete. Then you can click on "Start Exploring Materials" and you should be able to see a periodic table like the following:



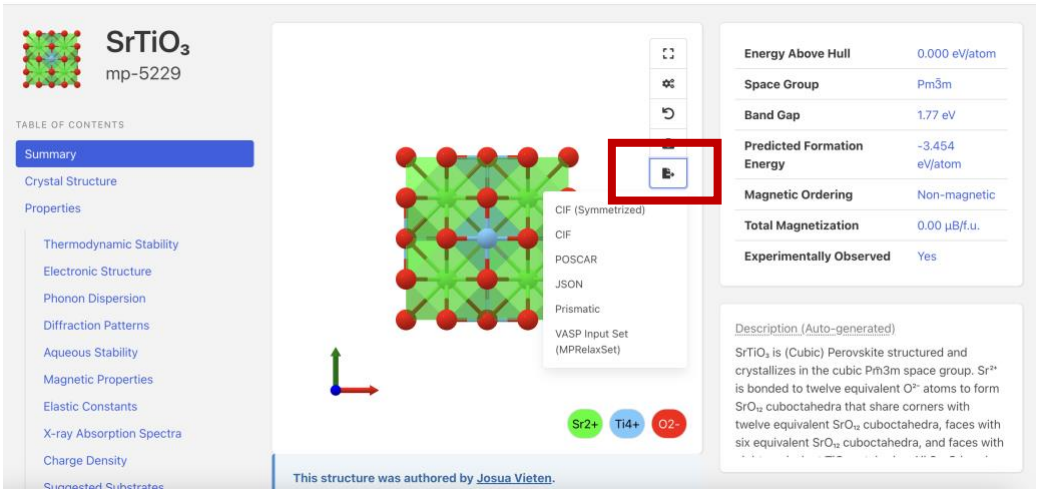
Now we can search for $SrTiO_3$ by entering it in the search bar:



We are looking for the $Pm\bar{3}m$ structure of $SrTiO_3$, which is the high-temperature cubic phase. We will study the cubic phase since it only contains 5 atoms, therefore calculations are relatively easy.



By clicking on the $Pm\bar{3}m$ field we are shown the properties of this structures that have been uploaded in the database:



All we need from this page is the structural data, which can be found in the POSCAR file. The 'poscar' format is the standard format of VASP, another widely used software package for DFT calculations that uses planewaves and pseudopotentials.
The 'poscar' file thus downloaded should look like the following:

```
Sr1 Ti1 O3
1.0
   3.9127013100000005    0.0000000000000000    0.0000000000000002
   0.0000000000000006    3.9127013100000005    0.0000000000000002
   0.0000000000000000    0.0000000000000000    3.9127013100000005
Sr Ti O
1 1 3
direct
   0.0000000000000000    0.0000000000000000    0.0000000000000000 Sr2+
   0.5000000000000000    0.5000000000000000    0.5000000000000000 Ti4+
   0.5000000000000000    0.0000000000000000    0.5000000000000000 O2-
   0.5000000000000000    0.5000000000000000    0.0000000000000000 O2-
   0.0000000000000000    0.5000000000000000    0.5000000000000000 O2-
```

Here the first line is a comment field, the second line contains the lattice parameter a in Å. Lines 3–5 contain the lattice vectors, scaled by the lattice parameter a: $a_1 = a$, $a_2 = a$, $a_3 = a$ (note that in this example the authors decided to set $a = 1$ so as to give the lattice vectors directly in Å). Line 6 contains the list of atoms in the unit cell, followed by the number of atoms of each type on line 7, in the same order. The keyword 'direct' on line 8 specifies that the atomic coordinates in lines 9-13 are expressed as fractional coordinates (units of 'direct' lattice), e.g., the Ti atom is at $(a_1 + a_2 + a_3)/2$.

- Construct the input file for a total energy calculation of $SrTiO_3$. You can start by modifying an input file from a previous exercise. If anything is unclear, please consult the documentation at http://jdftx.org/.

- Using the input file just created, say sto-1.in, perform a test run. This test is only to make sure that everything goes smoothly. For this test we can use some arbitrary convergence parameters, say `elec-cutoff 20` Ha and a Brillouin-zone sampling 4 4 4 with shifted k-points: `kpoint-folding 4 4 4` and `kpoint 0.5 0.5 0.5`

Convergence tests for STO

Now that we have a basic setup for $SrTiO_3$, we need to perform convergence tests.

- Determine the planewaves kinetic energy cutoff that is required to have the total energy converged to within 50 meV/atom. For this calculation you can use the same `kpoint-folding` and `kpoint` settings as in the previous exercise. As a reminder, we performed this kind of tests for silicon in HandsOn 02.

- Using the cutoff just obtained, determine the sampling of the Brillouin zone required to have the total energy converged to within 10 meV/atom. As a reminder, we performed this kind of tests for silicon in HandsOn 02.

Structure optimization for STO

- Using the convergence parameters obtained in Exercise 3, determine the optimized lattice parameter of $SrTiO_3$ by using a calculation of type `lattice-minimize`, as in HandsOn 5.

In this calculation you will note that the residual pressure at the end of the run is still nonzero. In order to fully optimize the lattice parameter, it is convenient to perform one or two additional runs using the optimized parameter as a starting point.

- Determine the optimized lattice constant of STO once again, this time by mapping the potential energy surface as a function of the lattice parameter, as in HandsOn 3.

- Which calculation provides the lowest total energy, the automatic optimization, or the manual calculation of the potential energy surface?

- Compare your optimized lattice parameter with the experimental value from Cao *et al*., PSSA 181, 387 (2000).

Bulk modulus of STO

As a sanity check, at the end of the last two exercises you should have obtained the following parameters:

```
…
kpoint-folding 4 4 4  #Gamma-centered by default
kpoint 0.5 0.5 0.5    #To offset
…
elec-cutoff 20        #wavefcn (density) cutoff Ha
…
```

- Use these parameters to calculate the bulk modulus of cubic $SrTiO_3$. In order to obtain reasonably accurate results, it is convenient to use the following relations:

$$B = \frac{1}{3}(C_{11} + 2C_{12}) = \frac{1}{9}\frac{1}{\Omega}\frac{\partial^2 U}{\partial \eta^2} \simeq \frac{1}{9}\frac{1}{\Omega}\frac{U(+\eta) - 2U(0) + U(-\eta)}{\eta^2}$$

This expression shows that we can calculate the bulk modulus by using the second derivative of the total energy with respect to the deformation parameter. The second derivative is then approximated using a finite-difference formula involving 3 points $(+\eta, 0, -\eta)$.

- Compare your calculated bulk modulus with the experimental values of Bell & Rupprecht, Phys. Rev. 129, 90 (1963). You should find that the deviation from experiment is smaller than 3%.

Hands-On 7
Vibrations and Phonon Dispersions

We start the hands-on session by going to the scratch directory, and creating and moving into a folder for this session:

```
$ cd $SCRATCH
$ mkdir HandsOn7; cd HandsOn7
```

In this session we will learn how to calculate the vibrational frequencies of molecules and solids and phonon dispersion relations.

Stretching frequency of a diatomic molecule

We start from the simplest possible system, the diatomic molecule Cl_2 studied in HandsOn3. For the input file, we use the converged parameters and optimized geometry from Hands-On3 or 5:

```
$ more Cl2.in
lattice Cubic 20

coords-type cartesian   #Specify coordinates in terms of absolute atom position in Bohr
ion Cl 0.000 0.000 0.000  0
ion Cl 3.743 0.000 0.000  0

kpoint-folding 1 1 1 #Gamma-centered by default

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/$ID_lda.uspp
elec-ex-corr lda
elec-cutoff 50 200   #Plane-wave KE cutoff for wfcn and charge density in Ha
electronic-SCF       #Perform a Self-Consistent Field optimization

dump-name Cl2.$VAR   #output file name
dump End ElecDensity #what you would like it to output
```

Let us run this calculation to make sure that everything runs smoothly.

In the lectures, we have seen that the vibrational frequency of a diatomic molecule can be calculated using:

$$\omega = \sqrt{\frac{2K}{M}}, K = \left.\frac{\partial^2 U}{\partial d^2}\right|_{d_0}$$

where M is the mass of the Cl nucleus, U is the total potential energy surface, d is the Cl-Cl distance, and d_0 is the equilibrium bond length.

By approximating the second derivative using finite differences, we have:

$$\hbar\omega \simeq \hbar\sqrt{\frac{2}{M}\frac{U(d_0+\delta) - 2U(d_0) + U(d_0-\delta)}{\delta^2}}$$

where δ is a small number, say $\delta = 0.001$ bohr.

We now calculate $U(d_0)$, $U(d_0+\delta)$, and $U(d_0-\delta)$ by creating two new input files where the coordinates of the second Cl atom are modified.

We can do this as usual using vi. Alternatively, we can use the following strategy, which is slightly faster:

```
$ sed "s/3.743/3.744/g" Cl2.in > Cl2_plus.in
$ sed "s/3.743/3.742/g" Cl2.in > Cl2_minus.in
```

Here we are replacing the distance 3.743 bohr by 3.744 and 3.742, respectively. It is convenient to extract the corresponding total energies from the output files on the fly. This can be done as follows within your job script:

```
mpirun -n 6 jdftx < Cl2.in | grep "Etot =" > U0.txt
mpirun -n 6 jdftx < Cl2_plus.in | grep "Etot =" > U_plus.txt
mpirun -n 6 jdftx < Cl2_minus.in | grep "Etot =" > U_minus.txt
```

In these expressions the vertical bar (`|`) 'pipes' the output from the command on the left (`mpirun -n 6 jdftx < Cl2.in`) into the input of the following command (`grep "Etot ="`). The output of `grep` is then 'redirected' (>) into the file on the right (`U0.txt`).

After executing these commands, we should see the following:

```
$ more U*.txt
::::::::::::::
U0.txt
::::::::::::::
     Etot =      -30.1274479260779380
::::::::::::::
U_minus.txt
::::::::::::::
     Etot =      -30.1274478268303980
::::::::::::::
U_plus.txt
::::::::::::::
     Etot =      -30.1274478133593959
```

At this point we can combine our results, considering that the mass of Cl is 35.45 amu (1 amu =1822.8885 m_e). We find:

$$\hbar\omega = 69.7 \text{ meV}$$

to be compared to the experimental value of 66.7 meV.

- Repeat the calculation of the stretching frequency of Cl_2, this time using (i) $\delta = 0.002$ bohr, (ii) $\delta = 0.01$ bohr.

Stretching frequency of a diatomic molecule with frozen phonon method in `vibrations`

The energies calculated so far have been at a specific set of atom coordinates, whether specified manually or optimized. These energies correspond to the Born-Oppenheimer surface, and do not include (free-)energy components due to the ionic degrees of freedom. This tutorial introduces the `vibrations` post-processing module with the example of the binding (free-)energy of Cl_2.

In order to execute `vibrations`, we first need to calculate the ground state properties of the system, so we run our input file with the converged atomic positions again:

```
$ more Cl2.in
lattice Cubic 20

coords-type cartesian   #Specify coordinates in terms of absolute atom position in Bohr
ion Cl 0.000 0.000 0.000  0
ion Cl 3.743 0.000 0.000  0

kpoint-folding 1 1 1 #Gamma-centered by default

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/$ID_lda.uspp
elec-ex-corr lda
elec-cutoff 50 200   #Plane-wave KE cutoff for wfcn and charge density in Ha
electronic-SCF       #Perform a Self-Consistent Field optimization

dump-name Cl2.$VAR   #output file name
dump End State #what you would like it to output
```

but this time we dump the final state of our system out.

```
mpirun -n 6 jdftx < Cl2.in > Cl2.out
```

Now, calculate the vibrational contributions by running jdftx on the input file:

```
$ more Cl2_vibrations.in
lattice Cubic 20

coords-type cartesian   #Specify coordinates in terms of absolute atom position in Bohr
ion Cl 0.000 0.000 0.000  0
ion Cl 3.743 0.000 0.000  0

kpoint-folding 1 1 1 #Gamma-centered by default

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/$ID_lda.uspp
elec-ex-corr lda
elec-cutoff 50 200   #Plane-wave KE cutoff for wfcn and charge density in Ha
electronic-SCF       #Perform a Self-Consistent Field optimization

initial-state Cl2.$VAR
dump End None
vibrations \
    dr 0.001 \
    centralDiff yes \
    translationSym yes \
    rotationSym yes \
    T 298
```

Edit the job script file to submit the jdftx calculation with vibrations:

```
mpirun -n 6 jdftx < Cl2_vibrations.in > Cl2_vibrations.out
```

The `vibrations` command calculates the normal modes of vibration by perturbing each atom by displacements of magnitude `dr` (which corresponds to the δ in the previous exercise) and constructs the force matrix optionally using a central-difference derivative formula (controlled by `centralDiff`). The `translationSym` and `rotationSym` commands control whether translation and rotational symmetries are used to constrain and optimize the force matrix calculations. (These symmetries should be used for molecular calculations but should not be used for solid or surface calculations.) `T` specifies the temperature for vibrational free energy calculation.

Now examine the generated output file. Notice that electronic minimization is performed for several perturbed configurations, and then normal modes obtained by diagonalizing the force matrix are reported. In this case, of the six degrees of freedom due to two Cl nuclei positions, five normal modes are zero (three translational and two rotational), and one normal mode, the Cl-Cl stretch, has a finite vibrational frequency of about 595 cm^{-1}.

The vibrational frequencies are usually given in units of cm^{-1}, i.e. as a wavenumber. The conversion is: 1 meV = 8.0655 cm^{-1}

The frequency obtained from this calculation of 73.8 meV differs from our result from the previous section because (1) acoustic sum rule, which modifies the dynamical matrix in such a way as to make sure that the molecule will not experience any restoring force when translated or rotated, also modifies the potential energy surface, and (2) the present calculations correspond to taking the second derivative of U in the limit $\delta \to 0$.

Finally, the output file reports the vibrational free energy components including the zero-point energy (ZPE), the vibrational energy including ZPE (`Evib`), the vibrational entropy (`TSvib`) and the net vibrational free energy (`Avib`).

Phonon dispersion relations of silicon with frozen phonon method in `phonon`

In this section we calculate the phonon dispersion relations of silicon. We begin by setting up the usual input file for diamond, from HandsOn3:

```
$ more Si_totalE.in
lattice face-centered Cubic 5.468
latt-scale 1.88973 1.88973 1.88973  #Convert lattice vectors from Angstroms to bohrs

coords-type lattice      #Specify atom coordinates in terms of the lattice vectors (fractional)
ion Si 0.00 0.00 0.00  0
ion Si 0.25 0.25 0.25  0

kpoint-folding 8 8 8  #Gamma-centered by default

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/$ID_lda.uspp
elec-cutoff 12  #wavefcn (density) cutoff Ha
elec-ex-corr lda
electronic-SCF       #Perform a Self-Consistent Field optimization

dump-name Si.$VAR
dump End State BandEigs #State and band eigenvalues
dump End ElecDensity    #Save the self-consistent electron density
dump End EigStats       #Get eigenvalue statistics
```

where we output the total energy and the final state of our system is now written as output files, to be used in the next step.

In periodic systems, the force matrix has a finite range that extends beyond a single unit cell of the system. In this case, Bloch's theorem implies that the eigenvectors are waves of atom vibrations, called phonons, and the eigenvalues exhibit a band structure term the phonon dispersion. Calculating forces due to atom perturbations beyond the range of one unit cell requires performing supercell calculations and mapping the resulting properties back to the unit cell. The vibrations command within `jdftx` only deals with unit cell calculations and that suffices for molecules. For solids, the phonon executable handles constructing supercells, performing perturbed calculations, and mapping the so-calculated phonon properties back to the original unit cell.

Once we obtain the total energy, we can set up the input file for the phonon calculation:

```
$ more Si_phonon.in
#Save the following to phonon.in
include Si_totalE.in        #Full specification of the unit cell calculation
initial-state Si.$VAR       #Start from converged unit cell state
dump-only                   #Don't reconverge unit cell state

phonon supercell 2 2 2      #Calculate force matrix in a 2x2x2 supercell
```

and run the phonon calculation with its own executable (note that this is separate from jdftx):

```
mpirun -n 6 phonon < Si_phonon.in > Si_phonon.out
```

The input file is extremely simple, relying on Si_totalE.in for the entire specification of the unit cell calculation and the corresponding converged unit cell results. The only addition is the phonon command, for which the only required argument is the supercell size in which to calculate the force matrix.

Note that phonon calculations are implemented in the JDFTx code for Gamma-centered total energy calculations only, so we chose a k-point mesh of 8x8x8 in this example. The supercell size must be a factor of the k-point folding: in this case, only 1, 2, 4, and 8 are acceptable for each dimension of the supercell. Also note that the calculation will be substantially more expensive as the supercell size is increased.

The output starts with listing of all the commands, followed by the usual initialization of a unit cell calculation under the heading "Unit cell calculation." This part ends on the energy evaluation in the unit cell at a fixed state (corresponding to dump-only).

Next, phonon constructs the supercell and determines its symmetries. It figures out all the atom displacements necessary to generate the force matrix: 2 atoms/cell x 3 Cartesian directions x 2 signs for central difference derivative = 12 for this example, and then finds the number of symmetry-irreducible perturbations, which happens to be just 1 in this case!

Then, for each symmetry-irreducible perturbation, it runs a supercell calculation with atom displaced (by $dr = 0.1$ bohr by default, see `phonon` command at http://jdftx.org/CommandPhonon.html to change this) under the heading "Perturbed supercell calculation X of N." The output format for each such calculation is also the usual JDFTx initialization followed by ElecMinimize (or SCF), but note that the lattice vectors are all twice as large, the bands and basis 8 times larger, while the k-point folding is halved, because this is a 2x2x2 supercell calculation.

At the end of each supercell calculation, phonon reports the energy and force change due to the perturbation per unit cell. Always make sure that these are one-two orders of magnitude larger than the convergence thresholds, either by adjusting those thresholds, or by adjusting dr in the phonon command. It is important to note that for accurate calculations, you should check that the phonon results are ***converged*** with respect to the dr parameter.

After all the supercell calculations, phonon collects force matrix contributions from each and outputs this to the binary `Si.phononOmegaSq` file (which we will use below). It also outputs the text file `Si.phononCellMap`, which lists the order of neighboring unit cells for the force matrix output (in exactly the same format as the `wannier.mlwfCellMap` that will be encountered in the Separated bands Wannier tutorial in HandsOn 9). Meanwhile, `Si.phononBasis` lists the order of atom perturbations in the phonon force matrix. The `Si.phononHsub` binary file contains electron-phonon matrix elements, which we will not discuss in this tutorial.

The code ends with a summary of the zero-point energy and vibrational free energy contributions, in exactly the same format as the output of the vibrations command, except it is now the (free-)energy contributions per unit cell of a periodic system. Note that phonon accounts for 3D periodicity for the Si example here, but will automatically switch to 2D or 1D periodicity, as appropriate, based on the geometry specified in the `coulomb-interaction` command.

Next, we list high-symmetry points in the Brillouin zone laying out a path along which we want the phonon dispersion:

```
$ more Si_bandstruct.kpoints.in
#Save the following to Si_bandstruct.kpoints.in
kpoint 0.000 0.000 0.000     Gamma
kpoint 0.000 0.500 0.500     X
kpoint 0.250 0.750 0.500     W
kpoint 0.500 0.500 0.500     L
kpoint 0.000 0.000 0.000     Gamma
kpoint 0.375 0.750 0.375     K
```

For common crystal structures, you can find the high-symmetry points easily on the web. For a more complete listing, you can consult a crystallography database e.g., the Bilbao database using the space group of your crystal.

From the above path specification, we can generate a sequence of points along the path and a plot script by running `bandstructKpoints` (in the `jdftx/scripts` directory) directly at the terminal (***not*** using a job script file in the queue, because this is a simple and fast script that does not require computationally expensive resources):

```
$ bandstructKpoints Si_bandstruct.kpoints.in 0.05 bandstruct
```

This should generate a file `bandstruct.kpoints` containing kpoints along the high-symmetry path and a gnuplot script `bandstruct.plot`. The second parameter, `dk`, of `bandstructKpoints` specifies the typical distance between kpoints in (dimensionless) reciprocal space; decreasing `dk` will produce more points along the path, which will take longer to calculate, but produce a smoother plot. Note that the last column of `banstruct.kpoints.in` is a label for the special Brillouin zone point which is used to label the plot.

Finally, we will calculate and plot the phonon dispersion from the force matrix output using this python script taken from the JDFTx tutorials website:

```
$ more PhononDispersion.py
#Save the following to PhononDispersion.py:
import numpy as np
from scipy.interpolate import interp1d

#Read the phonon cell map and force matrix:
cellMap = np.loadtxt("Si.phononCellMap")[:,0:3].astype(int)
forceMatrix = np.fromfile("Si.phononOmegaSq", dtype=np.float64)
nCells = cellMap.shape[0]
nModes = int(np.sqrt(forceMatrix.shape[0] / nCells))
forceMatrix = np.reshape(forceMatrix, (nCells,nModes,nModes))

#Read the k-point path:
kpointsIn = np.loadtxt('bandstruct.kpoints', skiprows=2, usecols=(1,2,3))
nKin = kpointsIn.shape[0]
#--- Interpolate to a 10x finer k-point path:
nInterp = 10
xIn = np.arange(nKin)
x = (1./nInterp)*np.arange(1+nInterp*(nKin-1)) #same range with 10x density
kpoints = interp1d(xIn, kpointsIn, axis=0)(x)
nK = kpoints.shape[0]

#Calculate dispersion from force matrix:
#--- Fourier transform from real to k space:
forceMatrixTilde  =  np.tensordot(np.exp((2j*np.pi)*np.dot(kpoints,cellMap.T)),  forceMatrix,
axes=1)
#--- Diagonalize:
omegaSq, normalModes = np.linalg.eigh(forceMatrixTilde)

#Plot phonon dispersion:
import matplotlib.pyplot as plt
meV = 1e-3/27.2114
```

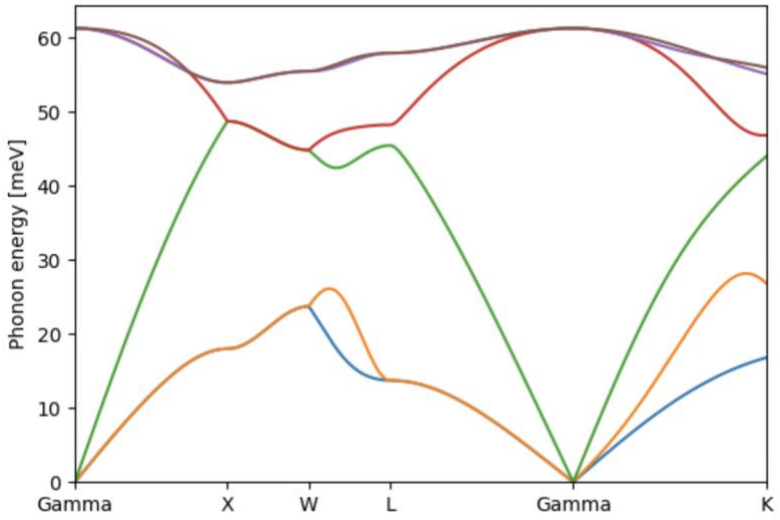
```
plt.plot(np.sqrt(omegaSq)/meV)
plt.xlim([0,nK-1])
plt.ylim([0,None])
plt.ylabel("Phonon energy [meV]")
#--- If available, extract k-point labels from bandstruct.plot:
try:
    kpathLabelText = ["Gamma", "X", "W", "L", "Gamma", "K"]
    kpathLabelPos = [0, 15, 23, 31, 49, 68]
    kpathLabelPos = [x * nInterp for x in kpathLabelPos]
    plt.xticks(kpathLabelPos, kpathLabelText)
except:
    print('Warning: could not extract labels from bandstruct.plot')
plt.show()
```

We will do this postprocessing in our local computers where visualization is available. For this, we will need to copy `Si.phononCellMap`, `Si.phononOmegaSq`, `bandstruct.kpoints` files to our local desktop, for example:

```
$ scp paradim-bpamuk@paradim.arch.jhu.edu:/scratch/paradim/paradim-
bpamuk/HandsOn7/Si/Si.phononCellMap .
```

Now we can create the plot running the python script above:

```
$ python3 PhononDispersion.py
```



Note that the phonon energies increase linearly from zero near the Gamma point, rather than quadratically as the electron energies do in band structure plots, because the phonon energy is the square root of the diagonalized quantity. This results in finite group velocities (the sound speeds) of phonons near Gamma-point.

Hands-On 8
Band Structures

We start the hands-on session by going to the scratch directory, and creating and moving into a folder for this session:

```
$ cd $SCRATCH
$ mkdir HandsOn8; cd HandsOn8
```

In this session we will learn how to calculate the electronic band structures of semiconductors and metals.

Band structure of silicon

This tutorial illustrates calculations of the electronic band structure, specifically, the variation of the Kohn-Sham eigenvalues along a special kpoint path in the Brillouin zone. It will also introduce an alternate algorithm for converging the electronic state, the self-consistent field (SCF) method.

First, let's run a total energy calculation as usual by copying the input and job script files from the previous tutorial:

```
$ cp $SCRATCH/HandsOn7/Si_totalE.in .
```

We add a new line in our Si_totalE.in to include the unoccupied bands in our calculation, too:

```
elec-n-bands 10             #Si has 4 occupied bands; asking for 6 unoccupied bands
```

```
$ cp $SCRATCH/HandsOn7/job.sh .
```

We will run with 4 processors today:

```
…
#SBATCH --ntasks-per-node=4     #number of cores/tasks per node
…
mpirun -n 4 jdftx < Si_totalE.in > Si_totalE.out
```

We also copy the high-symmetry points in the Brillouin zone from the previous example to generate the `bandstruct.kpoints` file:

```
$ cp $SCRATCH/HandsOn7/Si_bandstruct.kpoints.in .
$ bandstructKpoints Si_bandstruct.kpoints.in 0.05 bandstruct
```

and generate the path along which we want to calculate the electronic band structure.

Now we can run a band structure calculation along this path with the input file:

```
$ more Si_bandstruct.in
#Save the following to bandstruct.in
lattice face-centered Cubic 5.468
latt-scale 1.88973 1.88973 1.88973  #Convert lattice vectors from Angstroms to bohrs

coords-type lattice       #Specify atom coordinates in terms of the lattice vectors (fractional)
ion Si 0.00 0.00 0.00  0
ion Si 0.25 0.25 0.25  0

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/$ID_lda.uspp
elec-cutoff 12  #wavefcn (density) cutoff Ha
```

```
elec-ex-corr lda

include bandstruct.kpoints      #Get kpoints along high-symmetry path created above
fix-electron-density Si.$VAR    #Fix the electron density (not self-consistent)
elec-n-bands 10             #Number of bands to solve for
dump End BandEigs           #Output the band eigenvalues for plotting
dump-name bandstruct.$VAR   #This prefix should match the final parameter of bandstructKpoints
```

which we run using the edit in the job script:

```
mpirun -n 4 jdftx < Si_bandstruct.in > Si_bandstruct.out
```

which produces `bandstruct.eigenvals`.

We again do the plotting on our local computer, for this `scp` the files `bandstruct.eigenvals`, `bandstruct.kpoints`, and `bandstruct.plot`.

Finally running

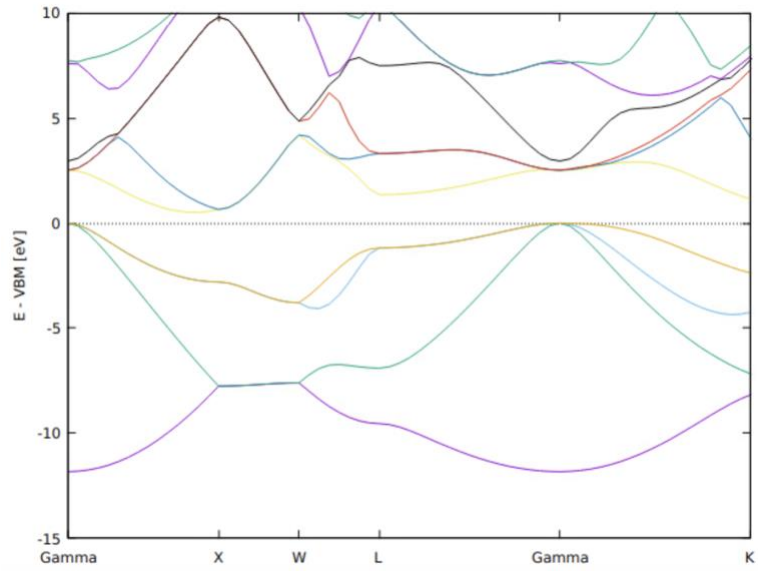
```
gnuplot --persist bandstruct.plot
```

will generate a plot of the electronic band structure.

Note that the y-axis (energy) is in Hartrees and its absolute value is not particularly meaningful because the electrostatic potential in a 3D periodic system does not have an unambiguous zero reference point. Typically, electronic band structure plots are referenced to the valence band maximum (VBM or HOMO) energy at zero for insulators, or the Fermi level (mu) for metals. Look at the `eigStats` output from the total energy calculation (`Si.eigStats`) to identify the VBM (HOMO) energy and replace the final line of the auto-generated `bandstruct.plot` with the following lines:

```
set xzeroaxis               #Add dotted line at zero energy
set ylabel "E - VBM [eV]"   #Add y-axis label
set yrange [*:10]           #Truncate bands very far from VBM
plot for [i=1:nCols] "bandstruct.eigenvals" binary format=formatString u 0:((column(i)-VBM)*27.21) w l
```

replacing `VBM` = 0.213073 in the last line with the relevant value from `Si.eigStats`. Note that this modification also converts the energy from Hartrees to eV for the plot. Now rerunning `gnuplot` produces the plot shown below.



Notice that at the Gamma point, the lowest band is single while the next three higher bands are degenerate: these line up with the *s* and *p* valence orbitals on the Silicon atoms. These degeneracies change in different parts of the Brillouin zone: the XW segment has two pairs of degenerate bands, while the WL and Gamma-K segments have no degeneracies. By looking for the valence band top at Gamma and the conduction band bottom along the Gamma-X line, we find that the band gap of silicon in DFT/LDA is E_g = 0.51 eV. The calculated band gap is much smaller than the experimental value of 1.2 eV.

Density of states of silicon

The previous tutorial showed one way of examining Kohn-Sham eigenvalues in a crystal, the electronic band structure, which plotted the eigenvalues along a path. This tutorial calculates the density of states in silicon, which is the probability distribution of eigenstates in energy (eigenvalue) space.

First, we calculate the total density of states using:

```
$ more Si_DOS.in
lattice face-centered Cubic 5.468
latt-scale 1.88973 1.88973 1.88973  #Convert lattice vectors from Angstroms to bohrs

coords-type lattice       #Specify atom coordinates in terms of the lattice vectors (fractional)
ion Si 0.00 0.00 0.00  0
ion Si 0.25 0.25 0.25  0

kpoint-folding 12 12 12  #Gamma-centered by default

#Select pseudopotential set:
ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/GBRV/$ID_lda.uspp
elec-cutoff 12  #wavefcn (density) cutoff Ha
elec-ex-corr lda
electronic-SCF       #Perform a Self-Consistent Field optimization

dump-name Si.$VAR
dump End None

elec-n-bands 10             #Si has 4 occupied bands; asking for 6 unoccupied bands
converge-empty-states yes   #Make sure that empty state eigenvalues are reliable

density-of-states Total     #Output total density-of-states

$ mpirun -n 4 jdftx < Si_DOS.in > Si_DOS.out
```

Starting from the total energy calculations of the previous two tutorials, all we needed to do is add the command `density-of-states`. Additionally, to get a few unoccupied states, we ask for more bands using `elec-n-bands` and make sure the empty states are converged at the end of the calculation using `converge-empty-states`. (Note that empty or unoccupied states that don't affect the total energy need not converge at all in a minimize calculation and may not converge fully in an SCF optimization.) We also cranked up the number of k-points to get smoother results.

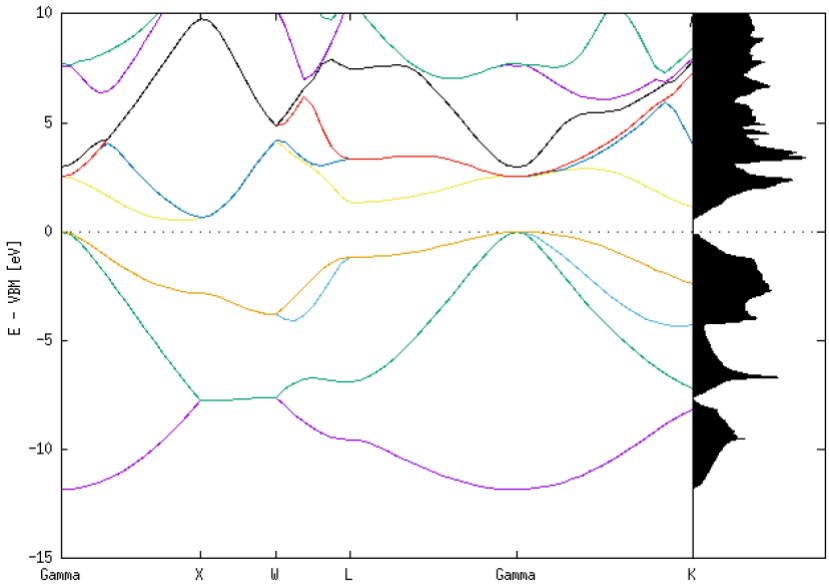
Running the above calculation produces a plain text file "`Si.dos`", which we can straightforwardly plot in `gnuplot`:

```
plot "Si.dos" u 1:2 w l
```

However, to make things a little more interesting (and show off some gnuplotting!), let's combine this with the band structure from the previous tutorial. Edit the last section of `bandstruct.plot` (including the edits from the previous tutorial) to be:

```
$ more bandstruct.plot
…
set xzeroaxis               #Add dotted line at zero energy
```

```
set ylabel "E - VBM [eV]"    #Add y-axis label
set yrange [*:10]            #Truncate bands very far from VBM
eV = 1/27.2114               #Value of eV in Hartrees
VBM = 0.213072               #VBM value (HOMO from totalE.eigStats)
xEnd = 68                    #Last x-tic value on k-path
set arrow \
  from first xEnd, graph 0 \
    to first xEnd, graph 1   #Draw separator

plot \
  for [i=1:nCols] "bandstruct.eigenvals" binary format=formatString u 0:((column(i)-VBM)/eV) w
l, \
  "Si.dos" u (xEnd+0.2*$2):(($1-VBM)/eV) with filledcurves y1=xEnd linecolor rgb "black"
```



This rotates the density-of-states and aligns its energy with the y-axis of the band structure, producing the plot shown above. Note the correlations between features in the band structure and the density of states. The highest density occurs where there are bands with low curvature, such as the flat section of the lowest bands in the XW segment and the low curvature for the lowest unoccupied bands near Gamma.

- Can you identify the band gap in the density of states? Correlating it with the band structure, is it a direct or indirect band gap? (That is are the highest occupied and lowest unoccupied states at the same k-point, or different ones?)

Next, let us examine contributions from various orbitals to the density of states. Modify the density-of-states commands in `Si.in` from `Total` to:

```
density-of-states \
    OrthoOrbital Si 1 s \
    OrthoOrbital Si 1 p
```

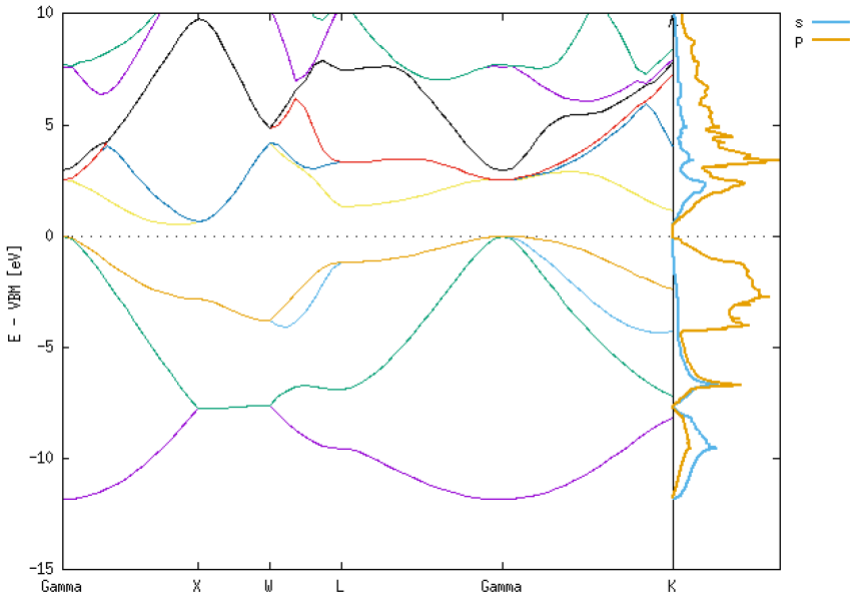
and rerun `jdftx` on that input file. Here we requested s and p orbital-projected contributions, which will both be saved to columns in the same file `Si.dos`. Update the final section of the latest version of `bandstruct.plot` to:

```
set key top right outside
plot \
   for [i=1:nCols] "bandstruct.eigenvals" binary format=formatString u 0:((column(i)-VBM)/eV)
w l title "", \
   "Si.dos" u (xEnd+0.5*$2):(($1-VBM)/eV) w l lw 2 title "s", \
   "Si.dos" u (xEnd+0.5*$3):(($1-VBM)/eV) w l lw 2 title "p"
```

and rerun:

```
gnuplot --persist bandstruct.plot
```

to get the plot shown below.



This time, instead of plotting a single total density of states, we plotted the contributions due to the *s* and *p* orbitals on the first Si atom. Note that the lower occupied bands exhibit *s* and *p* character, while the higher occupied bands have predominantly *p* character. The unoccupied bands have mixed character, but with a higher *p* fraction.

Band structure of aluminum

I recommend putting the results of Si and Al in their own folders, which can help in the following tutorial.

Silicon, the crystalline solid we have worked with is a semiconductor with a band gap which we identified in the previous section. This gap clearly demarcates occupied states from unoccupied states for all k-points, so electron 'fillings' are straightforward: the lowest (nElectrons/2) bands of all k-points are filled, and the remaining are empty. This is no longer true for metals, where one or more bands are partially filled, and these fillings (or occupation factors) must be optimized self-consistently. This tutorial introduces such a calculation for aluminum.

```
$ more Al_totalE.in
lattice face-centered Cubic 7.65

ion Al 0.00 0.00 0.00  0

ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/SG15/$ID_ONCV_PBE.upf
elec-cutoff 30

kpoint-folding 12 12 12

elec-n-bands 16
elec-smearing Fermi 0.01
electronic-SCF
converge-empty-states yes
```

```
dump-name Al.$VAR
dump End State BandEigs ElecDensity
density-of-states Total
dump End EigStats       #Get eigenvalue statistics
```

The parameters `elec-cutoff` and `kpoint-folding` are converged separately.

We will use a norm-conserving pseudopotential for aluminum because we will use a feature (dipole matrix elements) not supported for ultrasoft pseudopotentials in the last tutorial (which builds on this tutorial). Also note that this pseudopotential includes core electrons, so set a lower energy bound in the plots to filter these out and focus on the valence electrons.

The new keywords `elec-smearing Fermi 0.01` in red are needed whenever we deal with metals. These keywords instruct the code that we want to allow for fractional occupations of the Kohn-Sham states and that occupations are described using the Fermi-Dirac distribution, with a width of 0.01 Ha. We also ask the code to calculate 16 bands with `elec-n-bands 16` to account for the unoccupied bands of the metal.

Next, we can use the same path as the previous example, as the two cubic structures are similar, to generate the k-point path along which we want to plot the band structure:

```
$ cp Si_bandstruct.kpoints.in ./Al_bandstruct.kpoints.in
$ bandstructKpoints Al_bandstruct.kpoints.in 0.05 bandstruct
```

Now we are ready to run the band structure calculation with the input file:

```
$ more Al_bandstruct.in
lattice face-centered Cubic 7.65

ion Al 0.00 0.00 0.00  0

ion-species
/data/apps/extern/jdftx/1.7.0/usr/local/share/jdftx/pseudopotentials/SG15/$ID_ONCV_PBE.upf

elec-cutoff 30

include bandstruct.kpoints     #Get kpoints along high-symmetry path created above
fix-electron-density Al.$VAR   #Fix the electron density (not self-consistent)
elec-n-bands 16            #Number of bands to solve for
dump End BandEigs          #Output the band eigenvalues for plotting
dump-name bandstruct.$VAR  #This prefix should match the final parameter of bandstructKpoints
```

and after running the calculations by editing our job script file:

```
mpirun -n 4 jdftx < Al_totalE.in > Al_totalE.out
mpirun -n 4 jdftx < Al_bandstruct.in > Al_bandstruct.out
```

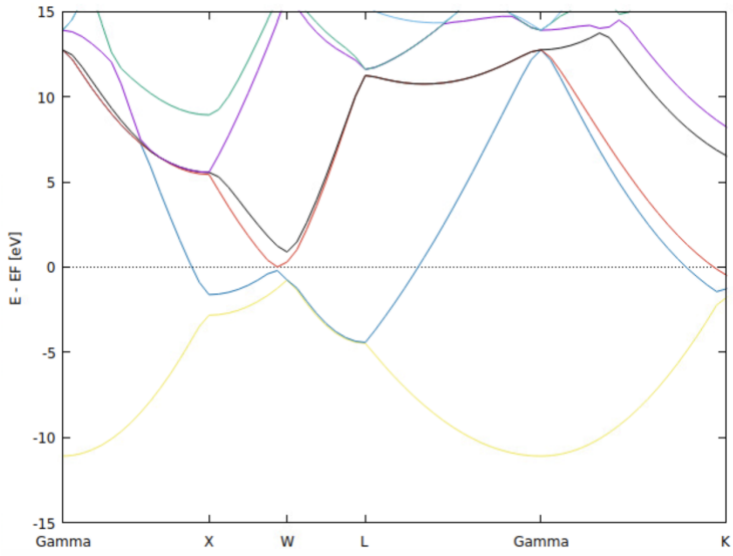
we are ready to plot the band structure using the `bandstruct.plot` with the appropriate edits as before.

- Edit bandstruct.plot to reference the energies relative to the Fermi level (or mu) (from the final `FillingsUpdate` line, or equivalently using `dump End EigStats`). and convert to eV.
- Also update the y-axis range to exclude energies far below the Fermi level, which we have set to zero in the plot and annotated with a dotted line.
- As a reference, the final script should look like:

```
#!/usr/bin/gnuplot -persist
set xtics ( "Gamma" 0,  "X" 15,  "W" 23,  "L" 31,  "Gamma" 49,  "K" 68 )
unset key
nRows = real(system("awk '$1==\"kpoint\" {nRows++} END {print nRows}' bandstruct.kpoints"))
```

```
nCols = real(system("wc -c < bandstruct.eigenvals")) / (8*nRows)
formatString = system(sprintf("echo '' | awk 'END { str=\"\"; for(i=0; i<%d; i++) str = str
\"%%\" \"lf\"; print str}'", nCols))
#plot for [i=1:nCols] "bandstruct.eigenvals" binary format=formatString u 0:i w l
set xzeroaxis               #Add dotted line at zero energy
set ylabel "E - EF [eV]"    #Add y-axis label
set yrange [-15:15]         #Truncate bands to see around the Fermi level
plot for [i=1:nCols] "bandstruct.eigenvals" binary format=formatString u 0:((column(i)-
0.398948)*27.21) w l
```

Run `bandstruct.plot` with gnuplot to get the electronic band structure shown below:



- Study the k-point convergence of this metal calculation following the outline of the Brillouin-zone sampling tutorial. How does the k-point convergence of the energy compare to the Silicon case?
- Try reducing the temperature in `elec-smearing Fermi` from 0.01 to 0.005 and redo the k-point convergence study.
- Compare the k-point convergence between the two temperatures.

Analytically, you expect the number of k-points per dimension to be inversely proportional to the Fermi temperature (smoothing width) in order to keep the accuracy (relative to the true infinite k-points limit) constant.

- Calculate the electronic density of states of Al and plot together with band structure as in the previous exercise.

Hands-On 9
Maximally Localized Wannier Functions

We start the hands-on session by going to the scratch directory, and creating and moving into a folder for this session:

```
$ cd $SCRATCH
$ mkdir HandsOn9; cd HandsOn9
```

In this session we will learn how to calculate the maximally localized Wannier functions (MLWFs) of semiconductors.

Separated bands of silicon

We will calculate maximally localized Wannier functions (MLWFs) for a set of bands that are separated in energies from other bands. This is applicable for the occupied valence bands of semiconductors and insulators. Here, we will construct and visualize Wannier functions for silicon and demonstrate band-structure interpolation using MLWF-based *ab initio* tight-binding models.

In a small molecule, the electron wavefunctions (Kohn-Sham orbitals) are localized around a few atoms, and we can easily visualize and interpret them. There is no corresponding example for the solids. This is, in part, because the Kohn-Sham orbitals now extend over the entire crystal, depend on the Bloch wave-vector $\boldsymbol{k}$ in addition to a band index (orbital number) and are, in general, complex-valued.

Wannier functions represent a Fourier transform of the Kohn-Sham orbitals over Bloch-wave vectors, which localizes them around one unit cell of the crystal. Now we no longer need to deal with each k separately: we have one Wannier function for each band centered at each unit cell; the Wannier functions are identical from one unit cell center to another.

However, these functions are not uniquely determined because the phase of the orbitals are independently variable for each k, and different combinations of phases generate different sets of Wannier functions.

Maximally-localized Wannier functions are a special case, where linear combinations and phases of orbitals from (i.e. unitary transformations of) a number of bands at each k are optimized to make the Wannier functions as localized as possible. MLWFs are still not unique, however, since once can still choose the combinations of bands that are linearly combined and the initial guess for the phases that determines which of several minima the optimization procedure converges to.

Let's start with the silicon calculations from HandsOn8:

```
$ cp $SCRATCH/HandsOn8/Si_totalE.in .
mpirun -n 4 jdftx < Si_totalE.in > Si_totalE.out

$ cp $SCRATCH/HandsOn8/Si_bandstruct.kpoints.in .
$ bandstructKpoints Si_bandstruct.kpoints.in 0.05 bandstruct

$ cp $SCRATCH/HandsOn8/Si_bandstruct.in .
mpirun -n 4 jdftx < Si_bandstruct.in > Si_bandstruct.out
```

Now, we can set up a Wannier input file:

```
$ more Si_wannier.in
#Save the following to wannier.in:
include Si_totalE.in

wannier \
    saveWfnsRealSpace yes
```

```
wannier-initial-state Si.$VAR
wannier-dump-name wannier.$VAR

wannier-center Gaussian 0.125 0.125 0.125
wannier-center Gaussian 0.625 0.125 0.125
wannier-center Gaussian 0.125 0.625 0.125
wannier-center Gaussian 0.125 0.125 0.625
```

This includes all the commands from the total energy calculation input file, and then adds commands specifying parameters for MLWF construction. The `wannier` command controls global options for the Wannier module, and here we specify that we want real-space wave function output (disabled by default since these files can be large!).

Next, the `wannier-initial-state` and `wannier-dump-name` commands specify which total-energy calculation Wannier should read the state from, and the file-name pattern for Wannier-specific outputs, respectively. Note that Wannier requires that the input total-energy calculation have at least `State` and `BandEigs` output.

Finally, the four `wannier-center` commands indicate the initial guesses ("trial orbitals") for constructing four MLWFs. The invocation above corresponds to Gaussian orbitals (of default sigma 1 bohrs) centered at the specified lattice coordinates: here we chose the centers of bonds from one Si atom to its nearest neighbors.

There are many other options for trial orbitals, including atomic orbitals and arbitrary linear combinations of Gaussian orbitals and atomic orbitals; see the `wannier-center` command documentation for details. By default, with four `wannier-centers` specified, the code will construct MLWFs from the four lowest bands in the system; this can be changed using the `wannier` command, as we will demonstrate in the next tutorial.

Now, generate the Wannier functions running the `wannier` code, this step is fast enough for us to run with one processor:

```
mpirun -n 1 wannier < Si_wannier.in > Si_wannier.out
```

Note that we are not using the `jdftx` executable here, but instead "`wannier`", which is also built in the same directory as jdftx.

Examine the output file `wannier.out`. The overall structure is very similar to jdftx output files.

The initialization section contains the usual parts followed by Wannier-specific parts.

After initialization, we find `WannierMinimize` lines which are optimizing a quantity called `Omega` (analogous to `ElecMinimize` optimizing Etot or F). This `Omega`, called the *spread function*, is the sum of variances of the Wannier functions, and minimizing it results in MLWFs.

At the end, the code reports the converged centers and spreads for each of the four MLWFs and ends with Wannier-specific outputs.

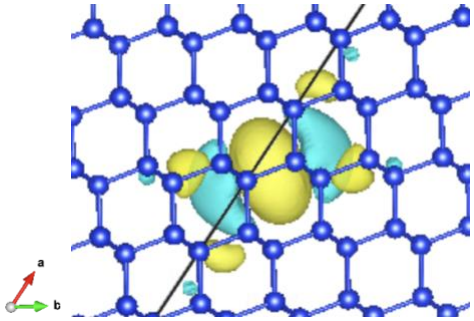
By default, the code dumps "`mlwfU`" which are the optimized unitary rotations at each k-point, "`mlwfH`" which is the Hamiltonian in the MLWF basis, and "`mlwfBandContrib`" specifying the ranges of bands and energies covered by each MLWF. Additionally, during initialization, the code dumps "`mlwfBandRanges`" which specifies the minimum and maximum energies of each band, which we will find useful in the next tutorial, and "`mlwfCellMap`" which specifies lattice vectors to surrounding unit cells in the order used for storing the Hamiltonian, which we will use below. Finally, since we asked for it, we also got real-space MLWF output in four files "`0.mlwf`", "`1.mlwf`" etc.

Visualization of Wannier functions:

You can use the `createXSF` script and VESTA to visualize Wannier functions as well:

```
$ createXSF wannier.out wannier.0.xsf wannier.0.mlwf
```

Now the output is on a supercell rather than the unit cell. The dimensions of this supercell depend on the k-point sampling and are 8 x 8 x 8 in this case. Use the boundary settings in VESTA to zoom in on the Wannier function, as shown below:



Note that the Wannier function is centered on the Si-Si bond as expected and has 3-fold symmetry around the bond because it is along a [111] axis.

Similarly, plot the other three Wannier functions and notice that they are identical except for being centered on a bond with a different direction (four bond directions over all for the tetrahedral lattice in a diamond structure).

Band structure with Wannier Hamiltonian:

Next, we'll generate the band structure using the Wannier Hamiltonian output. The "`mlwfH`" file contains the Hamiltonian in the optimized MLWF basis. It contains an array of matrices of dimensions nCenters x nCenters (4 x 4 in this case) corresponding to the matrix elements between each of the MLWFs at one lattice site with each of the MLWFs at a nearby lattice site.

The order of lattice sites is specified in the "`mlwfCellMap`" file.

The number of lattice sites depends on the k-point sampling and fills up a Wigner-Seitz cell of the effective supercell (8 x 8 x 8 in this case). Examine `wannier.mlwfCellMap`, it will have a somewhat larger number of lines than $8^3 = 512$ because it repeats boundary cells to maintain crystal symmetry. (You can visualize the cell map in `gnuplot` using: `splot "wannier.mlwfCellMap" u 4:5:6 )`

The above MLWF-basis Hamiltonian is a tight-binding model, but with multiple states on each lattice site (4 in this case), and connections to neighboring sites beyond just the nearest ones (as specified by the cell map). This tight-binding model reproduces the first-principles band structure from DFT exactly and is called an *ab initio* tight-binding model. The following Python script calculates band structure from MLWF Hamiltonian:

```
$ more WannierBandstruct.py
#Save the following to WannierBandstruct.py:
import numpy as np
from scipy.interpolate import interp1d

#Read the MLWF cell map, weights and Hamiltonian:
cellMap = np.loadtxt("wannier.mlwfCellMap")[:,0:3].astype(int)
Wwannier = np.fromfile("wannier.mlwfCellWeights")
nCells = cellMap.shape[0]
nBands = int(np.sqrt(Wwannier.shape[0] / nCells))
Wwannier = Wwannier.reshape((nCells,nBands,nBands)).swapaxes(1,2)
#--- Get k-point folding from totalE.out:
for line in open('Si_totalE.out'):
    if line.startswith('kpoint-folding'):
        kfold = np.array([int(tok) for tok in line.split()[1:4]])
kfoldProd = np.prod(kfold)
kStride = np.array([kfold[1]*kfold[2], kfold[2], 1])
#--- Read reduced Wannier Hamiltonian and expand it:
Hreduced = np.fromfile("wannier.mlwfH").reshape((kfoldProd,nBands,nBands)).swapaxes(1,2)
```

```
iReduced = np.dot(np.mod(cellMap, kfold[None,:]), kStride)
Hwannier = Wwannier * Hreduced[iReduced]

#Read the band structure k-points:
kpointsIn = np.loadtxt('bandstruct.kpoints', skiprows=2, usecols=(1,2,3))
nKin = kpointsIn.shape[0]
#--- Interpolate to a 10x finer k-point path:
xIn = np.arange(nKin)
x = 0.1*np.arange(1+10*(nKin-1)) #same range with 10x density
kpoints = interp1d(xIn, kpointsIn, axis=0)(x)
nK = kpoints.shape[0]

#Calculate band structure from MLWF Hamiltonian:
#--- Fourier transform from MLWF to k space:
Hk = np.tensordot(np.exp((2j*np.pi)*np.dot(kpoints,cellMap.T)), Hwannier, axes=1)
#--- Diagonalize:
Ek,Vk = np.linalg.eigh(Hk)
#--- Save:
np.savetxt("wannier.eigenvals", Ek)
```

The first two sections read in the MLWF data and set up the k-point path respectively; all the real action happens in two lines of code in the final block!

The Bloch Hamiltonian `Hk` is calculated by Fourier transforming the MLWF-basis Hamiltonian, and then we obtain the eigenvalues at each k (the band structure) by diagonalizing `Hk`.

Run the above script directly at the cluster:

```
$ python3 WannierBandstruct.py
```

and examine the output file "`wannier.eigenvals`": it is a text file with the eigenvalues for each band in a column.

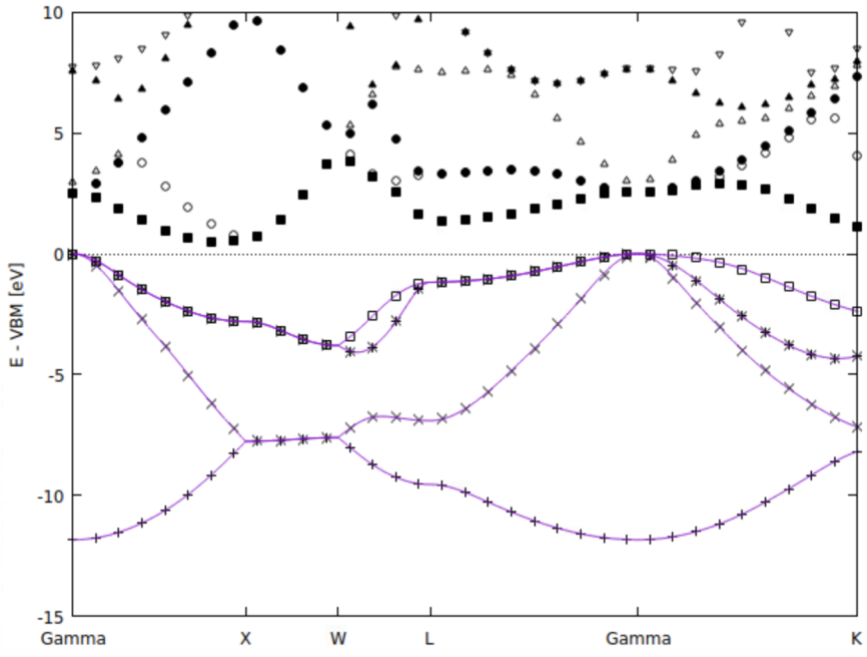
To obtain the bands from Wannier, `scp` the files `bandstruct.kpoints`, `bandstruct.plot`, and `wannier.eigenvals` from the ARCH cluster to your local computer.

Modify the end of `bandstruct.plot` to compare the DFT and Wannier band structure:

```
#Edit end of bandstruct.plot to the following:
VBM = 0.213073  #HOMO or mu from totalE.eigStats
eV = 1/27.2114  #in Hartrees
nWan = 4        #number of bands from Wannier functions
set xzeroaxis
set yrange [:10]
set ylabel "E - VBM [eV]"
plot for [i=1:nCols] "bandstruct.eigenvals" binary format=formatString u 0:((column(i)-VBM)/eV)
every 2 w p lc rgb "black"
replot for [i=1:nWan] "wannier.eigenvals" u (0.1*$0):((column(i)-VBM)/eV) w l lt 1
```

and run the gnuplot script:

```
$ gnuplot --persist bandstruct.plot
```



The Wannier band structure (plotted as lines) indeed reproduces the DFT band structure (plotted as points) exactly. Since we only constructed MLWFs from the lowest four (occupied) bands, we get the interpolated Wannier band structure only for those bands. Extending Wannier band structure interpolation to unoccupied bands and metals requires handling Entangled bands as we discuss in the next session.

Hands-On 10
Dielectric Function with Direct Transitions

We start the hands-on session by going to the scratch directory, and creating and moving into a folder for this session:

```
$ cd $SCRATCH
$ mkdir HandsOn10; cd HandsOn10
```

In this session we will learn how to calculate the maximally localized Wannier functions (MLWFs) of metals and other systems where the unoccupied states are of interest, with the example of Al.

Entangled bands of aluminum

We generalize from insulators and semiconductors where it is possible to single out sets of bands separated in energy from the rest to the more general case where the bands of interest are mixed in with other bands. This is relevant not only for metals, but also for semiconductors, when we are interested in representing the unoccupied states. Specifically, this tutorial constructs maximally localized Wannier functions (MLWFs) for aluminum and demonstrates accurate density-of-states calculations using Monte Carlo sampling.

Let us start with copying and running the total energy and band structure calculations of Al from the Hands-On 8:

```
$ cp $SCRATCH/HandsOn8/Al_totalE.in .
$ mpirun -n 4 jdftx < Al_totalE.in > Al_totalE.out

$ cp $SCRATCH/HandsOn8/Al_bandstruct.kpoints.in .
$ bandstructKpoints Al_bandstruct.kpoints.in 0.05 bandstruct

$ cp $SCRATCH/HandsOn8/Al_bandstruct.in .
$ mpirun -n 4 jdftx < Al_bandstruct.in > Al_bandstruct.out
```

If you ran this in a separate folder for Al, you could simply copy the folder here and continue instead of running it again.

We would like Wannier functions that capture the band structure of aluminum for a few eV surrounding the Fermi level, which is the relevant range for optical properties of the metal, for example. However, note that there are no sets of bands that cover this range, but are completely separated in energy from all other bands. We will therefore adopt an alternate strategy: we will require that the Wannier functions capture the band structure within a range of energies called the inner window exactly, but we will include bands from a larger range of energies called the outer window in the optimization procedure.

Specifically, let us work with 5 Wannier functions this time. In the band structure above, note that 5 bands can take us from the bottom of the valence bands approximately 11 eV below the Fermi level to just more than 9 eV above the Fermi level, where the sixth band starts (at the X point).

Therefore, we will set our inner window from 12 eV below the Fermi level (to include the bottom completely) to 8 eV above, which corresponds to the eigenvalue range [-0.042,0.693] Hartrees. This includes mu = 0.399 Hartrees; the ranges in the input file are absolute and not relative to mu.

For the outer window, we need to pick a range that contains at least five bands throughout the Brillouin zone: we can start at the same lower end as the inner window and extend to 17 eV above the Fermi level to satisfy this, which corresponds to the eigenvalue range [-0.042,1.024] Hartrees.

With these parameters we can set up the Wannier input file:

```
$ more Al_wannier.in
include Al_totalE.in

wannier \
    innerWindow -0.042 0.693 \
    outerWindow -0.042 1.024 \
    saveMomenta yes

wannier-center Gaussian -0.101403   0.435547   0.487558
wannier-center Gaussian -0.443339   0.055289  -0.446398
wannier-center Gaussian -0.361538  -0.010718   0.032397
wannier-center Gaussian -0.296948   0.326939  -0.400117
wannier-center Gaussian  0.030459  -0.158043  -0.194850

wannier-initial-state Al.$VAR
wannier-dump-name wannier.$VAR

wannier-minimize nIterations 300
```

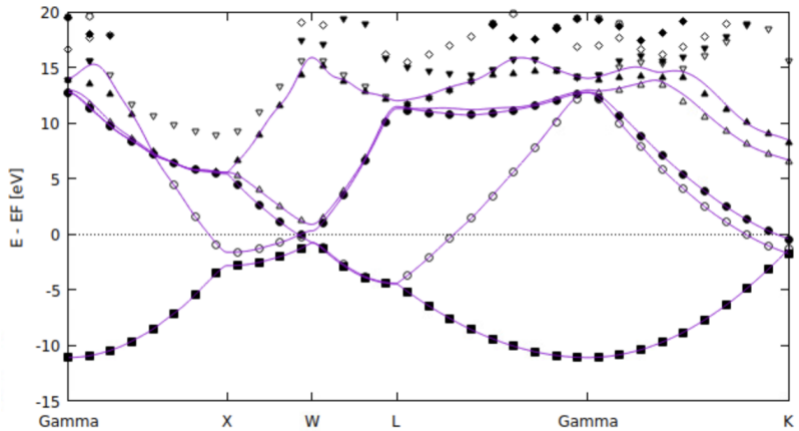
This time we have specified five wannier-center commands with trial orbitals centered at random coordinates. Note the window specifications in the `wannier` command. We additionally specified `saveMomenta` which will cause the code to output momentum matrix elements in a file suffixed "`mlwfP`". (We will use that output in the next tutorial.) Starting from random trial orbitals usually requires more steps to converge, which is why we increased `nIterations` using `wannier-minimize`.

Run the MLWF optimization:

```
$ mpirun -n 1 wannier < Al_wannier.in > Al_wannier.out
```

and examine `wannier.out`. Check the final converged center coordinates and spreads.

Use `WannierBandstruct.py` and `bandstruct.plot` from the previous tutorial to compare the DFT and MLWF-interpolated band structures. Remember to change the number of Wannier bands in bandstruct.plot and to reference the energies to mu instead of VBM.



- Try changing the initial random orbitals: do the converged centers change?
- Also try adjusting the windows: what effect does the window size have on the minimized spread?

- Note the constraints on the window sizes: the inner window must contain at most five bands throughout, while the outer window must contain at least five bands throughout (for five `wannier-centers`).

- Five was an arbitrary choice too: you could cover a greater energy range by including six or more `wannier-centers`.

Density of States with Wannier functions:

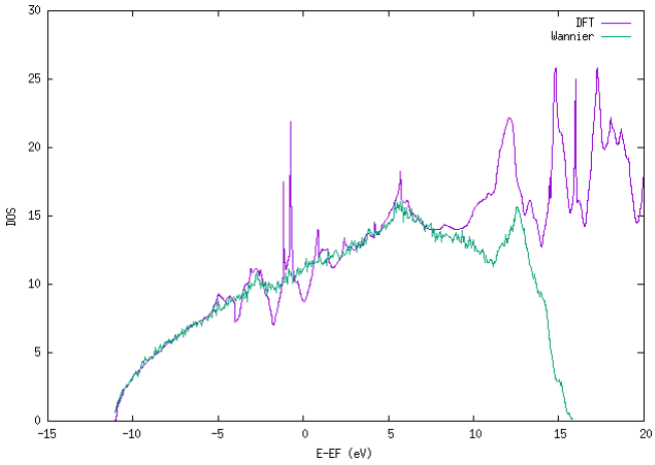
We can use the MLWF-basis Hamiltonian to calculate electronic properties accurately within the inner energy window. For example, the following script calculates the density of states:

```
$ more wannierDOS.py
#Save the following to WannierDOS.py:
import numpy as np

#Read the MLWF cell map, weights and Hamiltonian:
cellMap = np.loadtxt("wannier.mlwfCellMap")[:,0:3].astype(int)
Wwannier = np.fromfile("wannier.mlwfCellWeights")
nCells = cellMap.shape[0]
nBands = int(np.sqrt(Wwannier.shape[0] / nCells))
Wwannier = Wwannier.reshape((nCells,nBands,nBands)).swapaxes(1,2)
#--- Get k-point folding from totalE.out:
for line in open('Al_totalE.out'):
    if line.startswith('kpoint-folding'):
        kfold = np.array([int(tok) for tok in line.split()[1:4]])
kfoldProd = np.prod(kfold)
kStride = np.array([kfold[1]*kfold[2], kfold[2], 1])
#--- Read reduced Wannier Hamiltonian and expand it:
Hreduced = np.fromfile("wannier.mlwfH").reshape((kfoldProd,nBands,nBands)).swapaxes(1,2)
iReduced = np.dot(np.mod(cellMap, kfold[None,:]), kStride)
Hwannier = Wwannier * Hreduced[iReduced]

#Calculate DOS by Monte-Carlo sampling:
nBlocks = 100  #number of blocks
nK = 1000      #number of k per block
EkAll = np.zeros((nBlocks,nK,nBands))
for iBlock in range(nBlocks):
    kpoints = np.random.rand(nK,3)          #Generate random k
    Hk = np.tensordot(np.exp((2j*np.pi)*np.dot(kpoints,cellMap.T)), Hwannier, axes=1)
    Ek,Vk = np.linalg.eigh(Hk)              #Diagonalize
    EkAll[iBlock] = Ek                      #Save energies
    print("Block", iBlock+1, "of", nBlocks) #Report progress
#--- Histogram:
nBins = 400
hist,binEdges = np.histogram(EkAll, nBins, density=True)
hist *= (2*nBands)  #Integral should be 2 nBands but np.histogram normalizes to 1
bins = 0.5*(binEdges[1:]+binEdges[:-1])  #bin centers
#--- Save:
outData = np.vstack((bins,hist)).T #Put together in columns
np.savetxt("wannier.dos", outData)
```

The initial setup is the same as `WannierBandstruct.py`, but then we calculate eigenvalues on random k-points in the Brillouin zone. We are using a total of 100000 k-points above, but split it into 100 blocks of 1000 k-points each for reducing memory usage, especially to keep the size of the dot(kpoints, cellMap.T) matrix manageable. Finally, we histogram the computed eigenvalues to get the density of states.

```
$ python3 WannierDOS.py
```

will generate `wannier.dos` file which you can plot to obtain the density of states:



The Wannier DOS tracks the DFT one on the inner window, but not outside it. Within the inner window, notice that the Wannier DOS smooths over the spikes in the DFT DOS. Those spikes are artifacts of low-density regular k-point meshes; the Monte-Carlo sampling used 50 times as many k points and samples the Brillouin zone more evenly (no special planes / orientations), thereby avoiding these issues.

Direct transitions

This tutorial will introduce more complicated post-processing techniques using MLWF-based *ab initio* tight-binding models. Here, we will calculate the frequency-dependent imaginary part of the dielectric function for aluminum, accounting for direct interband transitions.

The simplest mechanism of light absorption in materials is the direct excitation of an electron from an occupied state to an unoccupied state, absorbing one photon. Energy conservation requires that the difference between these states equals the energy of the photon. In solids, momentum conservation implies that the initial and final state are at the same k-point because photons have negligible momentum at the electronic scale. Using perturbation theory and Fermi's golden rule, we can calculate the rate of this process for a given light intensity. From this rate, we can determine the corresponding imaginary part of the dielectric function as:

$$\mathrm{Im}\epsilon(\omega) = \frac{4\pi^2 \mathrm{e}^2}{3m_\mathrm{e}\omega^2\Omega}\int_{\mathrm{BZ}} \frac{2d\vec{k}}{\mathrm{V_{BZ}}} \sum_{n'n} (f_{\vec{k}n} - f_{\vec{k}n'})\delta(\varepsilon_{\vec{k}n'} - \varepsilon_{\vec{k}n} - \hbar\omega) \left|\langle\vec{p}\rangle^{\vec{k}}_{n'n}\right|^2$$

where Ω is the unit cell volume, $\varepsilon_{\vec{k}n}$ and $f_{\vec{k}n}$ are Kohn-Sham eigenvalues and occupations respectively, and $\langle\vec{p}\rangle^{\vec{k}}_{n'n}$ are the momentum matrix elements connecting band n and n' at Bloch wave-vector k. The integral averages over the Brillouin zone, and the delta function implements energy conservation. See paper by A.M Brown *et al.,* [ACS Nano, **10**, 1, 957–966 (2016)] for more details. Small differences here include a factor of 2 for spin degeneracy since we will use a non-relativistic calculation, and an average over spatial directions for an isotropic material that yields the factor of 3 in the denominator.

In the previous tutorial, we generated an *ab initio* tight-binding Hamiltonian for aluminum, which allows us to calculate the Kohn-Sham eigenvalues at arbitrary k. The occupation factors are 1 for eigenvalues smaller than mu, and 0 for those greater than mu. (The intermediate occupations for eigenvalues within k_BT of mu contribute negligibly for optical properties.) We also output the momentum matrix elements using the `saveMomenta` option of the `wannier` command. We will now use these outputs to calculate `ImEps` for aluminum using the `python` script below:

```
$ more WannierImEps.py
#Save the following to WannierImEps.py:
import numpy as np

#Read the MLWF cell map, weights and Hamiltonian:
cellMap = np.loadtxt("wannier.mlwfCellMap")[:,0:3].astype(int)
```

```
Wwannier = np.fromfile("wannier.mlwfCellWeights")
nCells = cellMap.shape[0]
nBands = int(np.sqrt(Wwannier.shape[0] / nCells))
Wwannier = Wwannier.reshape((nCells,nBands,nBands)).swapaxes(1,2)
#--- Get k-point folding from totalE.out:
for line in open('Al_totalE.out'):
    if line.startswith('kpoint-folding'):
        kfold = np.array([int(tok) for tok in line.split()[1:4]])
kfoldProd = np.prod(kfold)
kStride = np.array([kfold[1]*kfold[2], kfold[2], 1])
#--- Read reduced Wannier Hamiltonian, momenta and expand them:
Hreduced = np.fromfile("wannier.mlwfH").reshape((kfoldProd,nBands,nBands)).swapaxes(1,2)
Preduced = np.fromfile("wannier.mlwfP").reshape((kfoldProd,3,nBands,nBands)).swapaxes(2,3)
iReduced = np.dot(np.mod(cellMap, kfold[None,:]), kStride)
Hwannier = Wwannier * Hreduced[iReduced]
Pwannier = Wwannier[:,None] * Preduced[iReduced]

#Constants / calculation parameters:
mu = 0.399     #in Hartrees
eV = 1/27.2114 #in Hartrees
omegaMax = 6*eV #maximum photon energy to consider
Angstrom = 1/0.5291772 #in bohrs
aCubic = 4.05*Angstrom #in bohrs
Omega = (aCubic**3)/4  #cell volume

#Calculate ImEps by Monte-Carlo sampling of the BZ integral:
nBlocks = 100  #number of blocks
nK = 1000      #number of k per block
prefactor = 8*(np.pi**2)/(3.*Omega*nK*nBlocks) #Note me=e=hbar=1 in atomic units
omegaAll = []  #Frequencies of transitions
weightAll = [] #Corresponding weights
for iBlock in range(nBlocks):
    kpoints = np.random.rand(nK,3)         #Generate random k
    Hk = np.tensordot(np.exp((2j*np.pi)*np.dot(kpoints,cellMap.T)), Hwannier, axes=1)
    Pk = np.tensordot(np.exp((2j*np.pi)*np.dot(kpoints,cellMap.T)), Pwannier, axes=1)
    Ek,Vk = np.linalg.eigh(Hk)             #Diagonalize

    Pk = np.einsum("kba,kpbc,kcd->kpad",   #Sum using Einstein notation for
        Vk.conjugate(), Pk, Vk)            #transforming Pk to eigenbasis

    #Vectorized loop over pairs of states:
    Ei = np.repeat(Ek[:,:,np.newaxis], nBands, axis=2).flatten() #Initial energy
    Ef = np.repeat(Ek[:,np.newaxis,:], nBands, axis=1).flatten()   #Final energy
    omega = Ef - Ei                                        #Energy difference
    Psq = np.sum(np.abs(Pk)**2, axis=1).flatten()        #Matrix element squared

    sel = np.where(  #Select valid transitions:
        (Ei < mu) &        #initial state occupied
        (Ef > mu) &        #final state unoccupied
        (omega < omegaMax) #energy difference in range
    )[0]
    omegaAll.append(omega[sel])
    weightAll.append(prefactor * Psq[sel] / (omega[sel]**2))
    print("Block", iBlock+1, "of", nBlocks) #Report progress

#Histogram:
nBins = 200
omegaAll = np.concatenate(omegaAll)
weightAll = np.concatenate(weightAll)
hist,binEdges = np.histogram(omegaAll, nBins, weights=weightAll, density=True)
hist *= np.sum(weightAll)  #Integral should be sum(weights), but np.histogram normalizes to 1
bins = 0.5*(binEdges[1:]+binEdges[:-1])  #bin centers
#--- Save:
```

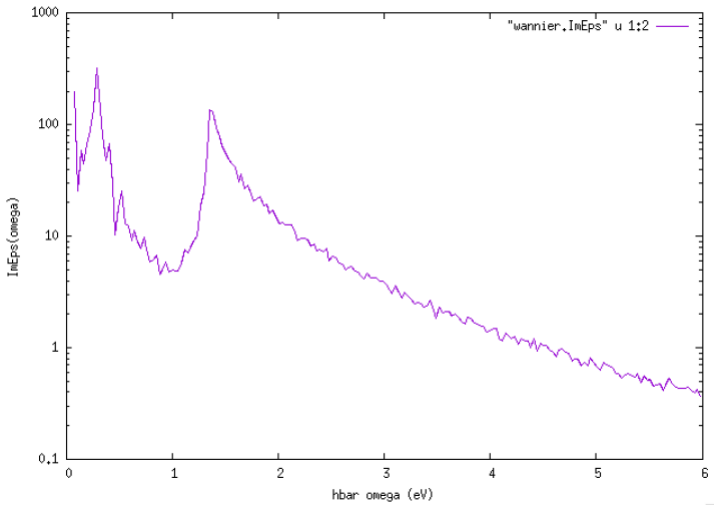
```
outData = np.vstack((bins/eV,hist)).T #Put together in columns, converting omega to eV
np.savetxt("wannier.ImEps", outData)
```

The overall structure is quite similar to `WannierDOS.py` from the previous tutorial. In addition to the Hamiltonian, now we also transform the momentum matrix elements, first from MLWF to *k* space, and then to the Kohn-Sham eigenbasis. Then the code considers pairs of states at each k-point and finds transitions that contribute to light absorption at relevant frequencies. Finally, these transitions are histogrammed by frequency, appropriately weighted by the matrix elements, to calculate ImEps as a function of frequency.

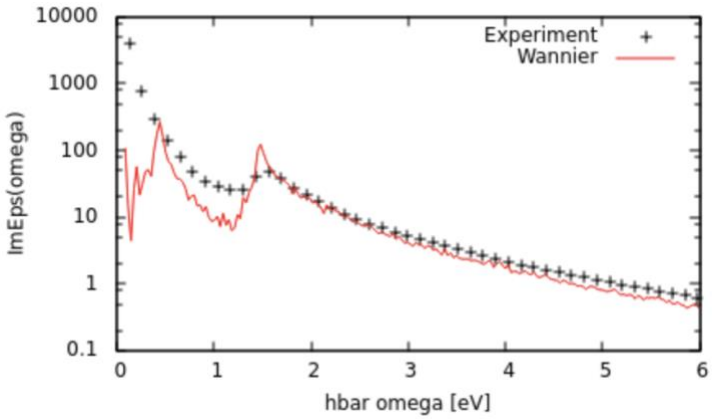
Running this python script:

```
$ python3 WannierImEps.py
```

generates the `wannier.ImEps` file, which can be ploted using `gnuplot`:



and can be compared with experiments as shown in figure from the jdftx tutorials website:



Note the excellent agreement with experimental results (from ellipsometry measurements; Palik handbook) for high frequencies. At low frequencies, the experimental measurements are larger because they also include contributions due to intraband transitions. Additionally, the peak at 1.5 eV is sharper in theory than experiment because we did not account for carrier broadening effects. Accounting for these additional mechanisms and broadening effects results in much better agreement with experiment as shown here.