`pynamic-structure-factor`: a `python` program to calculate dynamic structure factors in the classical limit

Tyler C. Sterling *

Department of Physics, University of Colorado Boulder, Boulder CO, 80309, USA

April 22, 2023

Contents

1 Introduction

Neutron and X-ray scattering are the flag-ship methods for measuring atomic dynamics and structure in condensed matter. Thermal neutrons and X-rays have wave-lengths that are $\sim$ Å, i.e. of the same order as the inter-atomic spacing [1]. In a neutron scattering or X-ray scattering experiment, we measure the *doubly-differential cross section*, $d\sigma/d\mathbf{\Omega}dE$, which basically tells us the fraction of the incident beam that is scattered in a solid-angle element, $d\mathbf{\Omega}$, with a change in energy in the interval $E + dE$. The vector $\mathbf{\Omega}$ points in the direction the particles scatter [1–3].

*ty.sterling@colorado.edu

If we don't care about the energy dependence, we can integrate the doubly-differential cross section over energy: this is equivalent to simply counting all neutrons scattered into the angle element $d\boldsymbol{\Omega}$ in an experiment. Integrating over energy gives us the differential cross section $d\sigma/d\boldsymbol{\Omega}$. The doubly-differential cross section is what is measured in *inelastic* scattering experiments and the differential cross section is what is measured in diffraction experiments. Integrating the differential cross section over all solid angles, we get the cross section, σ. It is called a *cross section* because it corresponds to the cross-sectional area of the incident beam that is scattered.

The doubly-differential cross section is the most fundamental quantity we measure since it can be used to determine $d\sigma/d\boldsymbol{\Omega}$ and σ. There is a tremendous amount of information encoded in the doubly-differential cross section and the data are extremely complicated. To make any progress understanding an experiment, we need a way to model the scattering. To that end, we note that the doubly-differential cross section is proportional to the *dynamic structure factor* (DSF) $S(\boldsymbol{Q},\omega)$ [1, 2]:

$$\frac{d^2\sigma}{d\boldsymbol{\Omega}dE} = \frac{k_i}{k_f}S(\boldsymbol{Q},\omega). \tag{1}$$

$k_i = |\boldsymbol{k}_i|$ and $k_f = |\boldsymbol{k}_f|$ are the magnitudes of incident and scattered neutron momenta[1]. respectively. $\boldsymbol{Q} = \boldsymbol{k}_f - \boldsymbol{k}_i$ is the change in momentum of the neutron: it is the momentum transferred into the scattering system. $E = \hbar\omega$ is the energy transferred into the scattering system by the particle. The point is, the doubly-differential scattering cross section, which is experimentally accessible, is simply related to $S(\boldsymbol{Q},\omega)$, which is a convenient quantity theoretically.

If we can calculate $S(\boldsymbol{Q},\omega)$, we can directly model a scattering experiment. Usually we want to do this because we have measured the atomic dynamics in a material and found an interesting signal, but are having a hard time understanding what is going on. $S(\boldsymbol{Q},\omega)$ is the Fourier transform of the (generalized) density-density correlation function, so is directly related to (the thermal average of) the real-space dynamics of the atoms in the scattering system [3]. In other words, given a set of experimental data, $S_{exp}(\boldsymbol{Q},\omega)$, we want to determine the atomic dynamics that generate the scattering signal: i.e. we want to solve an *inverse problem*.

A successful method to attack the inverse problem is reverse Monte Carlo (RMC) [4, 5]: we guess a configuration of the atoms and calculate the Boltzmann weight to deduce its likely hood. This is done for many configurations and they are averaged. If our guesses produce the right scattering intensity, then maybe we know what configurations produce the signal we care about. Still, RMC is based on ensemble averaging so we can really only probe the static microscopic structure: e.g. elastic diffuse and Bragg scattering. This is analogous to the situation in Monte Carlo atomic simulation methods where we lose information about the *dynamics* that lead to a physical observable by averaging over configurations on different phase-space trajectories. If knowing the dynamics is important, i.e. knowing the ω dependence, one often uses *molecular dynamics* (MD) simulations instead [6]. To calculate inelastic scattering, i.e. $S(\boldsymbol{Q},\omega)$, we need to use MD.

[1] Actually $\hbar\boldsymbol{k}$ is momentum, $\boldsymbol{k}$ is the wave-vector... but it is common to call the wave-vector $\boldsymbol{k}$ "momentum" too.

In this document, we learn how to calculate $S(\boldsymbol{Q}, \omega)$ from MD. See ref. [7] for related information. Important concepts for calculations are discussed, then my code for the calculations (`pynamic-structure-factor`) is introduced. Examples are provided for how to use it. In the appendix, a glossary of the parameters to run the code is provided and connection is made to the theory of neutron scattering from harmonic phonons.

2 The dynamic structure factor from molecular dynamics

We derive the equation used to calculate the DSF $S(\boldsymbol{Q}, \omega)$, from molecular dynamics here. Also see ref. [8].

The DSF is defined as [3]

$$\begin{aligned} S(\boldsymbol{Q}, \omega) &= \int \langle \hat{\rho}(\boldsymbol{r}, t) \hat{\rho}(0, 0) \rangle \exp(-i(\boldsymbol{Q} \cdot \boldsymbol{r} - \omega t)) dt d\boldsymbol{r} \\ &\equiv \int G(\boldsymbol{r}, t) \exp(-i(\boldsymbol{Q} \cdot \boldsymbol{r} - \omega t)) dt d\boldsymbol{r} \\ &\equiv \int F(\boldsymbol{Q}, t) \exp(i\omega t) dt. \end{aligned} \tag{2}$$

Eq. 2 is the time- and space-Fourier transform of the density-density correlation function (also called the *Van Hove* function [9]), $G(\boldsymbol{r}, t)$. $\hat{\rho}(\boldsymbol{r}, t)$ is the quantum mechanical density operator: in first quantization, it is a function of the position operators $\hat{\boldsymbol{r}}(t)$.

In general, positions do not commute at different times t, t' so evaluating eq. 2 is difficult. To simplify the notation, we do not write the explicit time dependence of $\hat{\boldsymbol{r}}$ except where it is needed. The usual method to evaluate eq. 2 for crystals is to expand the position operators, $\hat{\boldsymbol{r}}$, in terms of the phonon creation and annihilation operators [1] (see §B). This method works well when the harmonic approximation is sufficient, but at high-temperatures where anharmonicity matters and in molecular crystals where molecules rotate almost freely, this is won't work.

Instead, we approximate the positions as classical coordinates so that $\hat{\rho} \equiv \rho$ is classical and the classical positions $\boldsymbol{r}(t)$ can be determined using molecular dynamics simulations[2]. Importantly we have made no assumptions about the configuration of the material, so this method is valid for liquids, disordered compounds, molecular crystals, etc. The main error in the classical approximation is that the scattering function $S(\boldsymbol{Q}, \omega)$ does not satisfy the principle of detailed balance [1, 3, 10]. It is possible to add corrections that include quantum mechanical effects [10]; however, I do not pursue this here. In any case, the classical approximation becomes valid at high-temperature where quantum effects on nuclear motion are negligible and the particles follow Maxwell-Boltzmann statistics. Moreover, since we are sampling finite temperature trajectories, we naturally include anharmonic effects to all

[2] I believe that this can be shown to be a *stationary phase approximation* to the many-body correlation function in eq. 2... I am working on showing this else where. The important assumption that leads to validity of this approach is that the atoms are *distinguishable*, i.e. we can ignore quantum statistics. This is true above ~ 30 K atoms heavier than carbon. For helium, it is true above ~ 75 K. For deuterium and hydrogen (protium), it is true above ~ 175 K and ~ 300 K respectively.

orders. This is in contrast to using harmonic lattice dynamics where phonon life times are infinite.

For classical (i.e. commuting) positions, eq. 2 can be simplified. With

$$\delta(\boldsymbol{r}-\boldsymbol{r}_i)=\int \exp(i\boldsymbol{Q}\cdot(\boldsymbol{r}-\boldsymbol{r}_i))\frac{d\boldsymbol{Q}}{(2\pi)^3} \tag{3}$$

we can write $\rho(\boldsymbol{r},t)$ as [3]

$$\rho(\boldsymbol{r},t)=\sum_i^N b_i \int \exp(i\boldsymbol{Q}\cdot(\boldsymbol{r}-\boldsymbol{r}_i))\frac{d\boldsymbol{Q}}{(2\pi)^3}. \tag{4}$$

In eq. 4, b_i are the neutron scattering lengths. They are different for different types of atoms. The derivation for X-rays is more complicated, but the final result is nearly the same and is quoted below. i labels the atoms: it runs over all of the atoms in the simulation cell. Essentially the density as "seen" by the incident particles is the atomic density weighted by the probability of scattering off the atoms.

The classical expression for the Van Hove function, $\mathrm{G}(\boldsymbol{r},t)$ in eq. 2, is

$$G(\boldsymbol{r},t)=\langle\rho(\boldsymbol{r},t)\rho(0,0)\rangle=\int \rho(\boldsymbol{r}+\boldsymbol{r}',t+t')\rho(\boldsymbol{r}',t')d\boldsymbol{r}'dt'. \tag{5}$$

Inserting eq. 4 into eq. 5 and carrying out the integrals, we find

$$G(\boldsymbol{r},t)=\sum_i^N\sum_j^N b_ib_j\int \delta(\boldsymbol{r}-(\boldsymbol{r}_i(t+t')-\boldsymbol{r}_j(t')))dt'. \tag{6}$$

Similarly, inserting eq. 6 into eq. 2, we find:

$$F(\boldsymbol{Q},t)=\sum_i^N\sum_j^N b_ib_j\int \exp(-i\boldsymbol{Q}\cdot(\boldsymbol{r}_i(t+t')-\boldsymbol{r}_j(t')))dt'. \tag{7}$$

Next, we can rewrite $\exp(-i\boldsymbol{Q}\cdot\boldsymbol{r}(t))$ as

$$\exp(-i\boldsymbol{Q}\cdot\boldsymbol{r}(t))=\int \exp(-i\boldsymbol{Q}\cdot\boldsymbol{r}(\tau))\delta(\tau-t)d\tau. \tag{8}$$

Combining equations 7 and 8 with eq. 2, we can do all the integrals over exponentials:

$$S(\boldsymbol{Q},\omega)=\sum_i^N\sum_j^N b_ib_j\int\int \exp(-i(\boldsymbol{Q}\cdot\boldsymbol{r}_i(\tau)-\omega\tau))\exp(i(\boldsymbol{Q}\cdot\boldsymbol{r}_j(\tau')-\omega\tau'))d\tau d\tau'. \tag{9}$$

Finally, with $\tau \equiv t$, we can rewrite this as

$$S(\boldsymbol{Q}, \omega) = \Big| \sum_i^N b_i \int \exp\left(i \left(\boldsymbol{Q} \cdot \boldsymbol{r}_i(t) - \omega t\right)\right) dt \Big|^2 . \tag{10}$$

Equation 10 can be straight forwardly evaluated from molecular dynamics trajectories, $\boldsymbol{r}_i(t)$. This is the expression that `pynamic-structure-factor` (`PSF`) calculates. An expression similar to eq. 9 has been used in the past [11, 12]. However, the expressions in these references are not "squared" as they should be: for the scattering cross section (which is proportional to $S(\boldsymbol{Q}, \omega)$) to have the right dimensions, it must be proportional to b^2. Their expression is not (this is probably just a typo in [11] which was propagated into [12] which cites [11]). Moreover, their expressions are only valid for homogenous crystals with one atom type in the simulation (implicitly using units where $b_i \equiv b \equiv 1$).

If the experiment uses X-rays instead of neutrons, we need to replace the scattering lengths b_i in eq. 10 by the atomic form factors $f_i(Q)$. The form factors can be approximated by a sum of Gaussians:

$$f_i(Q) = \sum_j^4 p_{i,j} \exp\left(-q_{i,j} \left(\frac{Q}{4\pi}\right)^2\right) + s_i. \tag{11}$$

The parameters $p_{i,j}$, $q_{i,j}$, and s_i for X-rays and the scattering lengths b_i for neutrons are tabulated and can be looked up [13, 14]. The data in these references are what are used by `PSF`. The index i runs over all atoms in the simulation cell and $f_i(Q)$ is different for different atoms. Q is the (magnitude of-) momentum transferred from the incident (mono-chromatic) beam into the scattering system.

3 Finite size effects: an important concept

Computationally, the DSF is calculated as a time and space Fourier transform (FT) of the trajectories. The trajectory is spaced evenly on a grid in time, so the time FT is done using `scipy` FFT routines. However, since the positions aren't on a "grid" in space, it is not really practical[3] to do the space FT using FFT routines. Instead, the space FT is calculated directly using a highly vectorized algorithm[4].

There is an important "operational" distinction between FFT's and direct FT's. In the case of the time FFT, the frequency "grid" is fixed by the spacing of the time steps. It is possible to change this by "0-padding" and this might be useful to try to minimize spectral leakage, but I will not worry about this now. In the case of the direct FT, the user gives the

[3] We *could* put the scattering lengths onto an extremely fine grid: e.g. discretize the simulation cell and put 0's on all empty voxels and the scattering lengths on all voxels containing atoms. Since the coordinates are δ-functions, we need an *extremely* fine grid to do this accurately. We could then use numerical FFT's; however, this scheme would be prohibitively costly, both with computer memory and computational speed. This would be absurd.

[4] See the `psf.m_structure_factors.c_structure_factors` class if you want to check it out. In particular, look at the `_proc_loop_on_Q` method. If you can devise a better algorithm, I'm happy to replace mine with it!

$\boldsymbol{Q}$-points as arguments. This has advantages, e.g. you can only calculate on the $\boldsymbol{Q}$-points you care about rather than a huge grid. But there is a pitfall: *you* must pick $\boldsymbol{Q}$-points that are commensurate with the simulation cell or you will get non-sense data.

What does "commensurate" mean? Suppose that the density in your simulation cell is $\rho(x)$. $1d$ notation is used for convenience. Let the $1d$ box vector be R. We aren't making any assumptions about the system; it could be a crystal, liquid, etc. All we assume is the simulation uses periodic boundary conditions; the density satisfies $\rho(x) = \rho(r+R)$. The FT is given by $\rho(G) \propto \sum_x \rho(x) \exp(-iGx)$ and inverse FT by $\rho(x) \propto \sum_G \rho(G) \exp(iGx)$. Since $\rho(x) = \rho(x+R)$ is periodic, we require that $\exp(iGx) = \exp(iG(x+R)) = \exp(iGx)\exp(iGR)$. In other words, we require $G = 2\pi n/R$ with n an integer. In the limit of an infinite system, $\rho(G) = 0$ if n is *not* an integer; for a *finite* system, something else happens... if you want to know precisely what happens, I recommend finding a copy of "Diffuse Scattering and Defect Structure Simulations" by Neder and Profen [5] and reading §4.1. They explain the same concept in terms of "windowing functions" and it is more clear in that context what happens at incommensurate $\boldsymbol{Q}$-points. In any case, what happens at incommensurate $\boldsymbol{Q}$-points depends on the system size, so is called a "finite size effect".

Pragmatically, the requirement that the $\boldsymbol{Q}$-points be commensurate means that the $\boldsymbol{Q}$-points you want must be carefully chosen[5] with respect to the `lattice_vectors` you choose[6]. Let's see why: in the $1d$ case, suppose the box has length R and we choose a (fictitious) unitcell with $1d$ lattice vector $a = R/p$ with p an arbitrary positive number. In terms of the reciprocal lattice of the unitcell, the allowed wave vectors that satisfy periodic boundary conditions are

$$G = \frac{2\pi n}{R} = \frac{2\pi n}{pR/p} = \frac{n}{p}\frac{2\pi}{a} = \frac{n}{p}b$$

with $b = 2\pi/(R/p) = 2\pi/a$ the primitive reciprocal lattice vector of the unitcell. Then the wave vectors in r.l.u. of the unitcell are n/p with p an integer. So if $p = 1.31$ or something else stupid, then the *allowed* coordinates in r.l.u. are $1/p = 0.7634$, $2/p = 1.5267$, $3/p = 2.2901$, $\cdots$ this is clearly not a good choice. A better choice is something physically sensible.

Let us think about a $3d$ example: rutile TiO_2 (see the example, §7). Suppose we simulate a crystal that starts as a $16 \times 16 \times 24$ supercell formed by replicating the primitive unitcell. At finite temperature, the atoms move so periodicity with respect to the primitive cell is lost, but we don't expect a huge phase transition or anything, so we work in the reciprocal lattice units of the primitive cell. The primitive lattice vectors are

```
lattice_vectors = [[ 4.577, 0.000, 0.000], # angstroms
                   [ 0.000, 4.577, 0.000],
                   [ 0.000, 0.000, 2.949]]
```

For this supercell, the box vectors are

[5] The $\boldsymbol{Q}$-points are entered into the code in r.l.u. I suppose you could directly give them in Cartesian coordinates, but you would still have to ensure they are commensurate and I don't want to program support for multiple input options right now.

[6] The system still doesn't have to be a crystal; the lattice vectors are only used to determine the reciprocal lattice which then determines the $\boldsymbol{Q}$-points. For liquids or disordered systems, really we should calculate the spherical (powder) average of $S(\boldsymbol{Q}, \omega)$, but since the simulation uses periodic boundary conditions, the allowed $\boldsymbol{Q}$-points form a grid in reciprocal space. A natural way to represent this grid is with a lattice. If you want to spherical average the data after the fact, great!

```
box_vectors = [[ 16*4.577,    0.000,    0.000], # angstroms
               [    0.000, 16*4.577,    0.000],
               [    0.000,    0.000, 24*2.949]]
```

Along the x and y axes, the lattice vectors have $p_{xy} = 16$ and along z, $p_z = 24$. This means that the allowed wave vectors are Q_{xy} =0/16, 1/16, 2/16, ... and Q_z =0/24, 1/24, 2/24, ... i.e. $\boldsymbol{Q} = (n_x/16, n_y/16, n_z/24)$. The Bragg peaks are where $n_i = m_i \times p_i$ with m_i an integer since $Q_i = m_i \times p_i/p_i = m_i$ is an integer.

Rule of thumb: if you are simulating a crystal, or at least a system that is crystal-like (c.f. $CH_3NH_3PbI_3$), the lattice vectors should be the primitive lattice vectors of whatever the underlying unitcell is and the grid in reciprocal space should have spacing $\Delta Q \sim 1/N$, where N is the number of unitcell replicas along that axis. If your system is not a crystal... do something else... I dunno... I work on crystals.

4 Overview of the program

`PSF` isn't that hard to use. It is a `python` package that can be run in 2 "modes": (i) as a stand-alone program that calculates the DSF and writes it to a file and (ii) as an "API" that can be called from `python` scripts. Both modes are basically the same as will be seen below, with the API option as a convenient way to build a high through-put workflow. Examples using both methods are provided below.

A few disclaimers: I am self-taught programmer and work almost exclusively with `python`. Being self-taught, I probably don't follow programming best-practices and my code can almost certainly be sped-up and made more memory efficient by a more competent programmer. Working almost exclusively with `python`, I didn't write this code in a more appropriate language (I know *some* `FORTRAN`, but not enough to write this code as fast as in `python`). If either of these things bothers you, I am happy to help you improve my code or develop a new package to replace mine.

One particular issue I am aware of is that I, for some reason, chose to store arrays as column-major while `c` (what `numpy` and `python` are written in) is row-major... this is definitely a big slow down when slicing large arrays. It can be fixed pretty easily, but I haven't done it. The easiest fix is to add `order = "F"` to all arrays that matter when they are created.

4.1 Executable

Back to business: the program contains an "executable" `PSF.py` that can be used to run the program. This is the easiest and most straightforward way to use the code. The executable takes an input file as an optional argument: the *argument* is optional, the file is *mandatory* with the default file being `input_params.py`. If you name the file something else, e.g. `input.py`, you can run it from the command line by typing

```
python PSF.py -i input.py
```

All parameters in the file are optional and defaults are used for ones that are not given. The defaults are all declared in `$ROOT/psf/defaults.py`. Note, not all of the defaults are

sensible in general. It is your responsibility to correctly set them in the input file. The parameter names and meanings in the input file are discussed in the examples (§7) and in the glossary (§A).

In summary, the `PSF.py` code reads the input file, gets the trajectory from the specified trajectory file, then calculates $S(\boldsymbol{Q}, \omega)$ on the set of $\boldsymbol{Q}$-points specified by the user.

4.2 API

The main class in `PSF` is called `c_PSF` and is contained in `m_PSF.py`[7]. This class is a very simple interface that can be used to call the program from other scripts. Assuming the root directory containing the file `PSF.py` and the sub-directory `psf` are in your `python` path, you can just directly import the class by putting

```
from psf.m_PSF import c_PSF
```

in the header of your script. The input file is still required; if it's not given, the default is used. Or it can passed as an argument when instantiating `c_PSF`, e.g.

```
PSF = c_PSF(input_file = "input_params.py")
```

Then the calculation has to be "setup" before it can be run, i.e. the input file gets read, $\boldsymbol{Q}$-points, etc. get set up. This is done with

```
PSF.setup_calculation()
```

All of the arguments in the input file can be overwritten when setting up the calculations. e.g. if you want to use a different trajectory file than is specified in the input file, just pass in keyword arguments:

```
PSF.setup_calculation(trajectory_file = "pos.h5")
```

The precedence of what options are used is keyword arguments > input file > defaults.

Once the calculation is setup, it is run with

```
PSF.run()
```

This command also writes the files etc. so everything should finish willy-nilly without you having to do anything. The power of the API is that it can be used to chain calculations together, run batch jobs, etc. For example, suppose you want to calculate $S(\boldsymbol{Q}, \omega)$ on the same $\boldsymbol{Q}$-points etc. but for two different temperatures, e.g. 300K and 600K. You could run both calculations in one go with something like:

```
from psf.m_PSF import c_PSF

input_file = "input_params.py"
PSF = c_PSF(input_file = input_file)
```

[7] A general note: in my programs, the prefix "`c_`" denotes a class and "`m_`" denotes a module.

```
trajectory_file = "pos_300K.h5"
output_prefix = "sqw_300K.h5'
PSF.setup_calculation(trajectory_file = trajectory_file,
                      output_prefix = output_prefix)
PSF.run()

trajectory_file = "pos_600K.h5"
output_prefix = "sqw_600K.h5'
PSF.setup_calculation(trajectory_file = trajectory_file,
                      output_prefix = output_prefix)
PSF.run()
```

5 Installing the code

Like I said above, I am a bad `python` programmer. I don't know how to package this up into a "wheel" or whatever they are called in `pip`, or into a `conda` package. Rather, I just install it manually because that is what I am used to doing with compiled DFT and MD codes.

My code depends on `numpy`, `scipy`, and `hdf5`. Make sure these are installed and in your `python` path. I use `conda` for this.

Installing `PSF` is easy. The simplest and most elegant method I know is to just create a file with the extension ".pth" in the "site-packages" directory of your `python` installation. E.g. for me using the base environment in `anaconda`, I create "paths.pth" in the directory

```
/home/ty/anaconda3/lib/python3.9/site-packages
```

Inside "paths.pth" I add the path to the root directory of the `PSF` installation. E.g. for me, it is

```
/home/ty/research/repos/pynamic-structure-factor
```

`python` then looks inside this directory and in all of its sub-directories for the modules that are imported to run `PSF`.

A less elegant way to run the code is to just import the built-in `sys` library inside whatever script you are using to run `PSF` (e.g. `"PSF.py"`). Inside the script, add the command `sys.path.append("path")` where `"path"` is the path to the root directory of `PSF`.

6 Trajectories

The trajectory file contains the positions of the atoms as a function of time, the types of the atoms, and (sometimes) the "box vectors". These are the MD data that the code needs to calculate eq. 10. There are several supported "formats" for the input file as of now and more can be added (by me or you!) if you want them. The current formats and how to add new ones is detailed below.

For now, the only external MD package that is automatically supported is `lammps`. See below. Other packages can be supported by adding new allowed file formats to `PSF` or by externally converting the trajectories to the `user_hdf5` format.

For all formats, it is assumed that the positions are in Cartesian coordinates and that the order of the atoms in the file is the same for the entire trajectory; the latter is required since the atom types are only read for the first step in the file before looking up the scattering lengths or calculating form factors once and for all. The atom types are assumed to be consecutive integers starting at 1 and running up to the number of atom types in the file. You map the integers in the file to actual atom types using the variable `atom_types` in the `PSF` input file. See the glossary (§A).

6.1 `lammps_hdf5`

This my most frequently used file format. It is written out by `lammps` using something like the following commands in the `lammps` input file:

```
dump            POSITIONS all h5md 16 pos.h5 position species
dump_modify     POSITIONS sort id

...

undump          POSITIONS
```

See the `lammps` documentation for the details of these commands. The important aspects are that the data that are written are `position` and `species` and that `dump_modify ... sort id` is used to sort the atoms to the same order every time step. If you care, the data are accessed from the trajectory using the following paths:

box vectors: `"particles/all/box/edges/value"`

types: `"particles/all/species/value"`

positions: `"particles/all/position/value"`

The atom types should be consecutive integers starting at 1 and running up to the number of unique types in the simulation.

6.2 `user_hdf5`

This file type if for converting arbitrary trajectory data into a format `PSF` can read. The idea is that you write whatever code you need to convert your trajectory data into an `hdf5` file with the following fields:

types: `"types"`

positions: `"cartesian_pos"`

The box vectors are not read from a `user_hdf5` file since they are only used to "unwrap" the data and it is expected the data are already unwrapped in a custom generated file. If you don't know how to unwrap the data, contact me and I can give you some hints: it is a very instructive exercise on how periodic boundary conditions are handled computationally.

It expected that the steps in the file are evenly spaced and that the atoms are in the same order every step. The atom types should be consecutive integers starting at 1 and running up to the number of unique types in the simulation.

6.3 `external`

This allows you to pass positions and types directly into the code as `numpy` arrays. This method only works with the API mode and is specialized to some work I am doing: I am generating random configurations of a defected material and calculating diffuse scattering intensity. I am sort of trying to solve the "inverse problem" in a way that mirrors e.g. `DISCUS`. You really don't to need to worry about it unless this functionality unless it sounds like something you want... in which case, contact me directly.

6.4 Support for new file types

It is possible and mostly straightforward to add new file types to `PSF`. You have to do 3 things:

1. Modify the `psf.m_config.c_config._set_trajectory_format()` method to not crash if an "unknown" file format is requested. If you don't do this, your new format will be "unknown" and the program will crash. Add the string representing the new format to the list in the `if` statement.

2. Look at the `psf.m_trajectory.c_trajectory.get_atom_types()` method. Add an `if` statement to handle your new file format. Add a new a new method that opens your file and gets the type data from it. Your new method should be named `"_read_types_*"` where `"*"` is the string representing your file format. Your new method should only read the first time step because the atoms should be in the same order throughout the entire file. The types read from your file should be assigned to the `self.types` array attribute attached to the `c_trajectory` class. It is a $1d$ array with size `num_atoms` Look at the existing methods to read other formats for hints on how to write a new method. It's preferred but not required that your file is `hdf5`.

3. Look at the `psf.m_trajectory.c_trajectory.read_trajectory_block()` method. Add an `if` statement to handle your new file format. Add a new a new method that opens your file and gets the type data from it. Your new method should be named `"_read_pos_*"` where `"*"` is the string representing your file format. Your new method should take the array `"inds"` as an argument; `inds` contains the indices of the time steps that will be read from your file. The positions read from your file should be assigned to the `self.pos` array attribute attached to the `c_trajectory` class. The positions are in Cartesian coordinates and the array has the shape `num_block_steps` $\times$ `num_atoms` $\times$ `3`. If you also want to read the box vectors, assign them to the attribute `self.box_vectors`. It is a 3×3 array corresponding to the 3 *row* vectors representing the vectors spanning the simulation box. If the vectors are not constant during your simulation, average them. Look at the existing methods to read other formats for hints on how to write a new method. It's preferred but not required that your file is `hdf5`.

Keep in mind that reading some file types, e.g. a text file, if *much* slower than slicing directly from `hdf5`, especially if a stride is used to skip steps in the file.

7 Examples

In this section, we look at some examples of how to run the code, access the output, and understand the results. Two complete examples are provided: (i) rutile TiO_2 with and without vacancies and (ii) $CH_3NH_3PbI_3$ which has dynamic local ordering. For each case, we will look at both the excitations (i.e. $S(\boldsymbol{Q}, \omega)$) and diffuse scattering (i.e. $S(\boldsymbol{Q})$). For the diffuse scattering, we will see how to extract both the purely elastic data and energy integrated data. Finally, an "exercise" is suggested: calculate $S(\boldsymbol{Q}, \omega)$ in silicon with random germanium substitutions. It's not particularly interesting, but the calculation is easy. The "solution" to the exercise is provided. For each example, the input file will be discussed, but refer to the glossary (§A) for more details on all of the variables.

We will briefly mention the interesting physics involved, but this not a tutorial on diffuse or inelastic scattering. Rather, this is to showcase how to use the code. The physics is something you will have to work on yourself (though I am happy to chat with you about your physics problems if you want!).

Also be warned that the results calculated here are likely either under converged or wrong. **Do not use them for production calculations!** The systems are too small, the equilibration too short, and in the case of rutile, I haven't even tested that I am using the potential correctly. It is **your** responsibility to ensure that your MD calculations are converged and that you adequately sample to trajectory.

7.1 Rutile TiO_2

Rutile TiO_2 is a tetragonal crystal with lattice constants $a \sim 4.7$ Å and $c \sim 3.0$ Å. It is a polar insulator; the O atoms are anions, Ti are cations. Oxygen vacancies are a common defect that are technologically important, so let us think about those. When an O atom is removed from the crystal, the lattice around the vacancy site relaxes; i.e. there is strain. The strain results in deviation from the perfectly ordered crystal and will show up as diffuse scattering. For a low enough concentration of defects, we can still think in terms of phonons and Brillouin zones. The defects should also affect the phonon lifetimes and dispersion (i.e. lowering the thermal conductivity etc). The strain and renormalization of the phonons should show up in $S(\boldsymbol{Q})$ and $S(\boldsymbol{Q}, \omega)$ respectively.

I personally am rather more interested in the *ordering* of oxygen defects, i.e. the correlation between defects. But since I don't know *a priori* what the ordering is, let us just randomly place oxygen vacancies in the crystal and see what the scattering looks like.

7.1.1 MD simulations

This isn't a tutorial on MD, so I will just briefly mention what to do. Go to the directory `$ROOT/examples/rutile/lammps`. There are two sub-directories: `pristine` and `vacancies`. The `pristine` directory contains a structure file called `rutile_12x12x16.supercell` and a `lammps` input file called `lammps.in`. The structure is a $12 \times 12 \times 16$ replica of the 6 atom

primitive cell. The simulation contains 13824 atoms. The MD time step is 1 fs and the data are written to the file every 16 steps; i.e. the effective time step in the file is every 16 fs. The system is equilibrated in an NPT ensemble and then run in an NVE ensemble for 32000 steps. I.e. there are 2000 steps written to the file. The simulation temperature is 600 K. Go into this directory and run the simulation using `lammps`. I suggest using multiple processors; on my desktop with 16 cores, it only takes a few minutes. It should produce a trajectory file called `pos.h5`. Also read the `log.lammps` file and figure out what the lattice vectors are after NPT equilibration. You need them for the $S(\boldsymbol{Q}, \omega)$ calculation.

Now look in the `vacancies` directory. Here we simulate a rutile crystal with vacancies. The structure file is called `rutile_Ov_500_Tiv_250_12x12x16.supercell`. It is produced by the script `make_vacancies.py`. There are 500 oxygen vacancies and 250 Ti vacancies (to maintain charge neutrality). The `lammps` simulation is the same except there are only 13074 atoms. Run the simulation. It should produce a trajectory file called `pos.h5`.

7.1.2 Phonons

First, let us calculate the phonons, i.e. $S(\boldsymbol{Q}, \omega)$. Recall, there are two ways to do it: (i) executable mode or (ii) API. Let's see how to do both. First, we cover using the executable. Go into the phonons directory called `$ROOT/examples/rutile/phonons`. You should see a file called `input_pristine.py`. This file contains all of the settings needed to be set to calculate $S(\boldsymbol{Q}, \omega)$ for the pristine structure on a particular set of $\boldsymbol{Q}$-points. Open the file. The first few lines

```
# options for where to get data/preprocessing
trajectory_format = "lammps_hdf5"
trajectory_file = "../lammps/pristine/pos.h5"
unwrap_trajectory = True
```

specify the format and path to the trajectory file. `unwrap_trajectory = True` means to move atoms that cross the periodic boundary box back to their original side; `lammps` automatically wraps them around.

The next lines

```
# options for splitting up trajectory
num_trajectory_blocks = 2
trajectory_blocks = None
```

specify how to "average" the data. It is done by splitting the trajectory into "blocks" of data in time. In this case, there are 2 of them. The actual blocks used can be specified or left as `None` which means use them all. Note that `num_trajectory_blocks` sets the length of the trajectory block, i.e. it also sets the frequency resolution. It should be large enough to accurately sample the frequencies you care about: at least as large as any correlation in time in your simulation.

Next,

```
# options for writing results
output_directory = "out"
output_prefix = "pristine_exe"
```

specifies where to write the results. This will create a directory called `out` in the current working directory and write a file called `pristine_exe_STRUFACS.hdf5` in it.

The following lines

```
# simulation inputs
md_time_step = 16     # femtoseconds; time step IN FILE
md_num_steps = 2000   # number of steps IN FILE
md_num_atoms = 13824
```

specify data needed from the MD simulation. These should match what was set in the MD simulation (see the discussion above).

The lattice vectors variable

```
# unit cell used to define Q-points in cartesian coords
lattice_vectors = [[ 4.577, 0.000, 0.000], # angstroms
                   [ 0.000, 4.577, 0.000],
                   [ 0.000, 0.000, 2.949]]
```

specifies what lattice vectors to use to calculate the reciprocal lattice. The $\boldsymbol{Q}$-points from the user are represented in r.l.u. of these vectors, so choose them wisely. Also see the section on finite size effects, §3. The box vectors are used to unwrap the trajectory.

```
# vectors spanning the simulation cell
box_vectors = None
```

If they are set to `None`, they are read from the trajectory file. If not being unwrapped, they can be omitted.

The next few lines

```
# experiment info
experiment_type = "neutrons"
atom_types = ["Ti","O"]
```

specify the experimental data. The type is either `"neutrons"` or `"xrays"`. The atom types, which are used to look up scattering lenghts or X-ray form factors, are given as strings and there should be one per type in the simulation; in the `lammps` simulation, type 1 is Ti and type 2 is O.

Next, we need to specify the $\boldsymbol{Q}$-point path in reciprocal space to calculate $S(\boldsymbol{Q}, \omega)$ on:

```
# options for how to generate Q-points for calculation
Qpoints_option = "path"
Q_path_start = [[0,0,0],
                [5,0,0],
                [2.5,0,0]]
Q_path_end = [[5,0,0],
              [5,5,0],
              [2.5,5,0]]
Q_path_steps = [60,60,60]
```

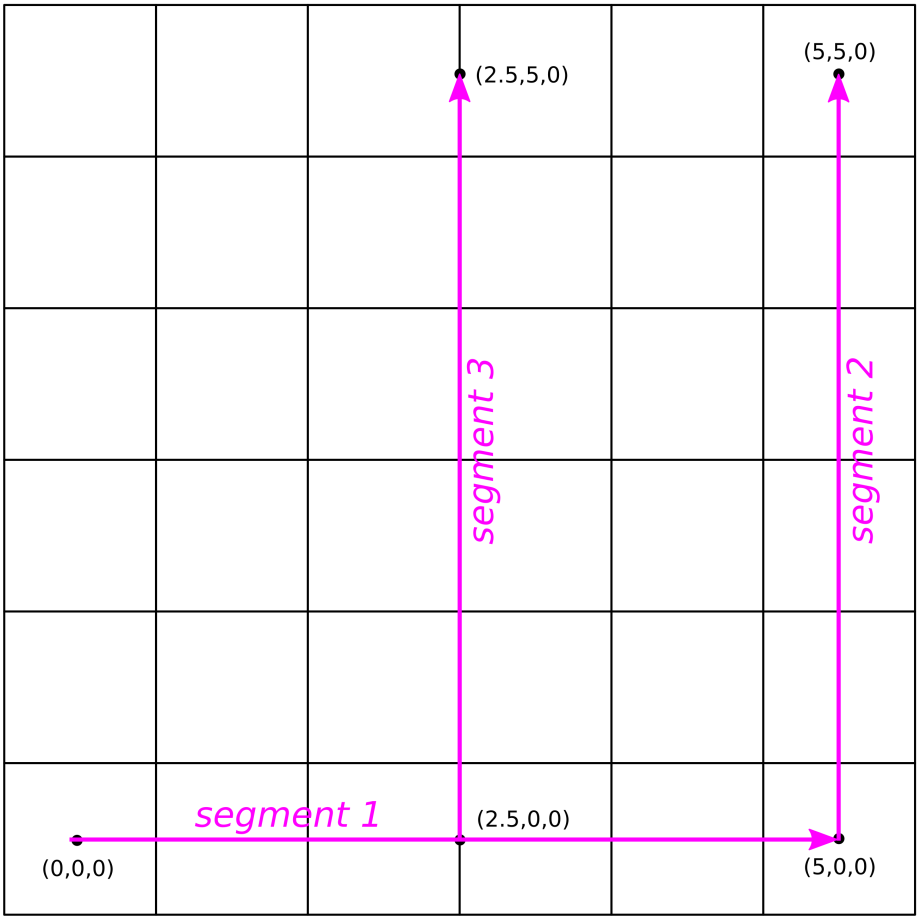

Fig. 1: Paths through reciprocal space used to calculate $S(\boldsymbol{Q}, \omega)$ in the examples. Each square is a Brillouin zone (BZ).

We tell the code we want to enter a path and then we enter the starting and ending vertices of each segment of the path. The *usual* use case is consecutive paths, so the *last* point on the segment is omitted to avoid duplicating $\boldsymbol{Q}$-points... note that in this case, the paths are *not* consecutive. The paths are $(0,0,0) \to (5,0,0)$, $(5,0,0) \to (5,5,0)$, and $(2.5,0,0) \to (2.5,5,0)$ (see fig. 1). Note that the last segment does not pass through any Bragg peaks, so there are no acoustic phonons along that segment. The last argument specifies the number of steps to take along each segment. Recall that we require $\boldsymbol{Q}$-points that are commensurate with the simulation box; our supercell is a $12 \times 12 \times 16$ replica of the unitcell. That means for paths along the x or y directions, we need steps that are spaced $\Delta Q = 1/12$ r.l.u. Here, we move 5 r.l.u. total along the x or y direction for each segment. Then we choose 60 steps since $5/60 = 1/12$.

Lastly,

```
# number of processes to split Q-point parallelization over
num_Qpoint_procs = 16
```

specifies the number of processes to use for the calculation. The calculation is parallelized

using `multiprocessing`. The $\boldsymbol{Q}$-points you request are automatically split over the number of processes you request. The trajectory file is read on one process and then copied to the others which calculate $S(\boldsymbol{Q}, \omega)$ on the subset of $\boldsymbol{Q}$-points assigned to them. `python` doesn't really support shared memory, so unless you have a lot of ram, you might want to use fewer processes.

Now that we know what is in the input file, we can run the calculation one of two ways. The first way is to copy the executable `PSF.py` into the working directory and run

```
python PSF.py -i input_pristine.py
```

The code should read the file, start running, then write the structure factors to the output file `out/pristine_exe_STRUFACS.hdf5`.

How do we access the data in the output file? I have provided utilities for this! They are in a class within my code: `psf.m_io.c_reader`. Look at the example plotting file `rutile/phonons/plots/simple_plot.py`. The following lines

```
from psf.m_io import c_reader

reader = c_reader("../out/pristine_exe_STRUFACS.hdf5")
reader.read_sqw()
Q_rlu = reader.Q_rlu
energy = reader.energy
sqw = reader.sqw
```

show how to access the data. `sqw` contains $S(\boldsymbol{Q}, \omega)$ and `Q_rlu` and `energy` contain the $\boldsymbol{Q}$-points and energies respectively. The rest of the script makes the plots, but I won't cover that. See §8 for details. You can try to run the script now and see what happens.

Cool. Now we are ready to use the API, which is more convenient for problems where we have multiple simulations to compare: e.g. pristine rutile vs rutile with vacancies. An example showing how to use the API is provided in `rutile/phonons/run_api.py`. Open it. The first few lines

```
from psf.m_PSF import c_PSF

# common configuration options for all calculations
input_file = "input_pristine.py"
PSF = c_PSF(input_file)
```

import the main class of `PSF` and tells it where to get common configuration options. Since we want to compare calculations between two different simulations, we use common options: e.g. the same $\boldsymbol{Q}$-point paths. Then some bonus options are set:

```
num_trajectory_blocks = 2
trajectory_blocks = None
num_Qpoint_procs = 16
```

These could be set in the input file, but I put them here to easily change on the fly.

The trajectory files for each simulation are in different locations, so we have to provide separate files for each. E.g.

```
PSF.setup_calculation(trajectory_file = "../lammps/pristine/pos.h5",
                      output_prefix = "pristine_neutrons",
                      experiment_type = "neutrons",
                      num_trajectory_blocks = num_trajectory_blocks,
                      trajectory_blocks = trajectory_blocks,
                      num_Qpoint_procs = num_Qpoint_procs)
PSF.run()
```

prepares to do a neutron scattering calculation on the pristine system and then runs it. We also want to compare neutron scattering to to X-ray scattering:

```
PSF.setup_calculation(trajectory_file = "../lammps/pristine/pos.h5",
                      output_prefix = "pristine_xrays",
                      experiment_type = "xrays",
                      num_trajectory_blocks = num_trajectory_blocks,
                      trajectory_blocks = trajectory_blocks,
                      num_Qpoint_procs = num_Qpoint_procs)
PSF.run()
```

For the vacancy calculation, we would use

```
vacancy_lattice_vectors = [[ 4.597, 0.000, 0.000], # angstroms
                           [ 0.000, 4.597, 0.000],
                           [ 0.000, 0.000, 2.961]]

PSF.setup_calculation(trajectory_file = "../lammps/vacancies/pos.h5",
                      output_prefix = "vacancies_neutrons",
                      experiment_type = "neutrons",
                      md_num_atoms = 13074,
                      num_trajectory_blocks = num_trajectory_blocks,
                      trajectory_blocks = trajectory_blocks,
                      num_Qpoint_procs = num_Qpoint_procs,
                      lattice_vectors = vacancy_lattice_vectors)
PSF.run()
```

The vacancy calculation has a different number of atoms (since some were removed) and different lattice vectors due to strain during relaxation in an NPT ensemble. We have to provide those data to `PSF`. We similarly set up a calculation for X-ray scattering from vacancies:

```
PSF.setup_calculation(trajectory_file = "../lammps/vacancies/pos.h5",
                      output_prefix = "vacancies_xrays",
                      experiment_type = "xrays",
                      md_num_atoms = 13074,
                      num_trajectory_blocks = num_trajectory_blocks,
                      trajectory_blocks = trajectory_blocks,
                      num_Qpoint_procs = num_Qpoint_procs,
```

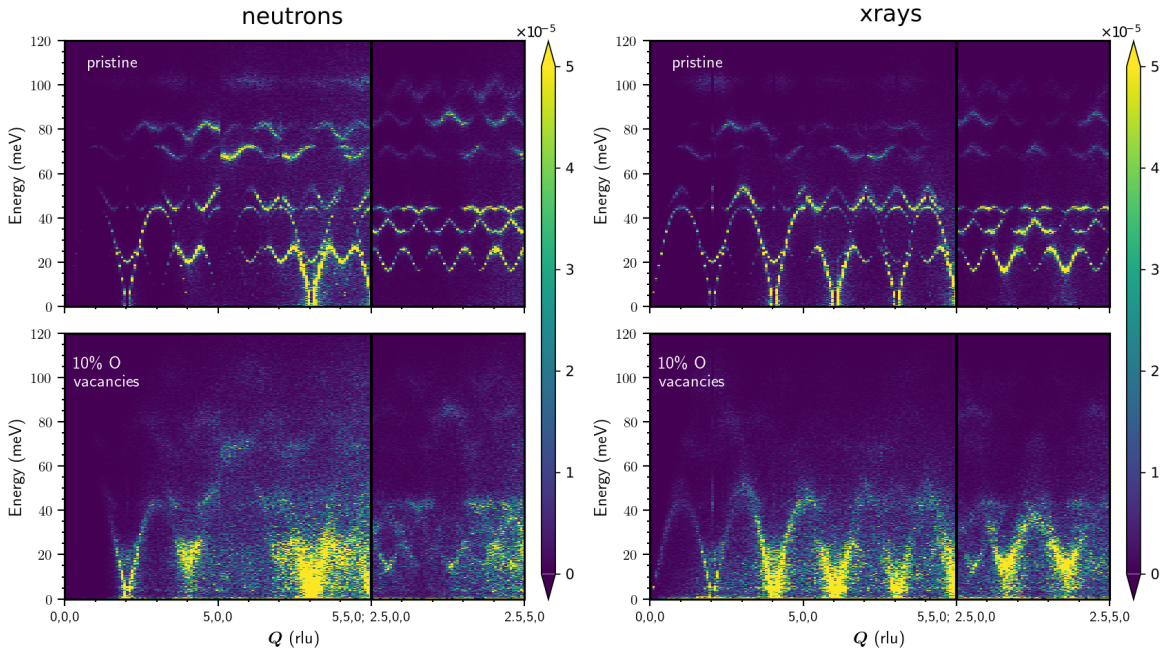

Fig. 2: Excitations $S(\boldsymbol{Q}, \omega)$ calculated using `PSF` for inelastic neutron scattering (left) and inelastic X-ray scattering (right). The top rows show scattering from a pristine crystal. The bottom rows show scattering from a crystal with 10% of the oxygen atoms as vacancies.

```
                                lattice_vectors = vacancy_lattice_vectors)
PSF.run()
```

All four calculations can be run in one go by executing

```
pyton run_api.py
```

This will write four output files:

```
out/pristine_neutrons_STRUFACS.hdf5,
out/pristine_xrays_STRUFACS.hdf5,
out/vacancies_neutrons_STRUFACS.hdf5,
out/vacancies_xrays_STRUFACS.hdf5.
```

Inside the directory `rutile/phonons/plots`, there are two files to read and plot $S(\boldsymbol{Q}, \omega)$ for the neutron and X-ray calculations. The files plot the pristine vs. vacancy calculations. The results are show in fig. 2. Try to reproduce these plots with my scripts or write your own!

Before we move on, let us briefly discuss the physics in fig. 2:

$\boldsymbol{Q}$-dependence: Let us momentarily focus only on the pristine neutron calculation (top left). It is clear that $S(\boldsymbol{Q}, \omega)$ depends sensitively on $\boldsymbol{Q}$ and ω. This dependence is due to the symmetry of the crystal and polarization of the phonons, e.g. longitudinal vs. transverse, optic vs. acoustic, etc. Acoustic phonons emerge from (some) Bragg peaks

and are linear near the Bragg peaks. Optic phonons are flat and at finite ($\omega \neq 0$) energy near Bragg peaks. The last segment is on BZ edges only, so contains no Bragg peaks, i.e. no phonons that go to $\omega = 0$. Also note that the intensity is larger for larger $|\boldsymbol{Q}|$. Actually, $S(\boldsymbol{Q}, \omega) \sim |\boldsymbol{Q}|^2$ with some other symmetry factors included (c.f. eq. 18). This is why the phonons are dim near $\boldsymbol{Q} = (0, 0, 0)$ and very bright near $\boldsymbol{Q} = (5, 5, 0)$. Here are some exercises: (i) Along the first segment, the acoustic phonons are steeper and disperse to higher energy than along the second segment. These are acoustic modes with different polarization. Which phonons have which polarization? How do you know? (hint: see the discussion following eq. 18). (ii) Why are the acoustic modes in the first segment steeper than the second? Think in terms of how the bonds in the crystal stretch due to different polarizations.

pristine vs. vacancies: Now let us compare pristine vs. vacancy systems from neutron scattering (left column). The scattering from the pristine system is sharp, i.e. the *width* of the modes is narrow. This means the phonons have long *lifetimes* ($\sim$1/width). The system with vacancies shows very broad phonons. What does that mean about the lifetimes? We usually assume that a system with defects has reduced thermal conductivity. How are our conclusions about the lifetimes in the pristine vs. vacancy defected system consistent with this assumption?

neutrons vs. X-rays: Now we focus on the differences between neutron and X-ray scattering. Look at the pristine systems in the top row. There are obvious differences: the X-ray calculation mostly shows intensity at lower energy and also shows phonons at different $\boldsymbol{Q}$ than in the neutron calculation. What gives?! As mentioned earlier, the X-ray calculation uses X-ray form factors; the neutron calculation uses scattering lengths. X-rays scatter from the *charge*, so the probability of an X-ray scattering off an atom scales with the atomic number. This means that very light elements are nearly invisible to X-rays when compared to heavy ones. On the other hand, neutrons scatter off the *nuclei* of atoms. The interaction is through the strong nuclear force; there is no good intuitive model for this interaction. The scattering lengths are measured in nuclear experiments: what we do know is that the scattering lengths vary wildly from atom to atom and from isotope to isotope. With these facts in mind, consider the following: (i) The X-ray scattering only shows low energy modes: Heavier atoms move slower, so phonons involving motion of mainly heavy atoms are lower frequency (i.e. lower energy). Between Ti and O, which is heavier? Which has a larger atomic number? How is the X-ray data consistent with these facts? (ii) In the X-ray data, there is intensity at different $\boldsymbol{Q}$-points than in neutron scattering. This is due to the unitcell form factor (eq. 15). Why is the form factor different between neutrons and X-rays? Hint: think in terms of "interference". If some atoms are "invisible" to X-rays, then there is no scattering from these atoms and waves that aren't scattered can't interfere. (iii) In neutron scattering, what are the scattering lengths of Ti and O? (Try finding them on Google: `https://www.ncnr.nist.gov/resources/n-lengths/` or in ref. [13].) One of them is negative but, since $S(\boldsymbol{Q}, \omega) \sim b^2$, this doesn't matter. In contrast to X-ray scattering, we see both low and high energy phonons. Is the consistent with the scattering lengths you looked up? Which phonons are due (mainly) to which

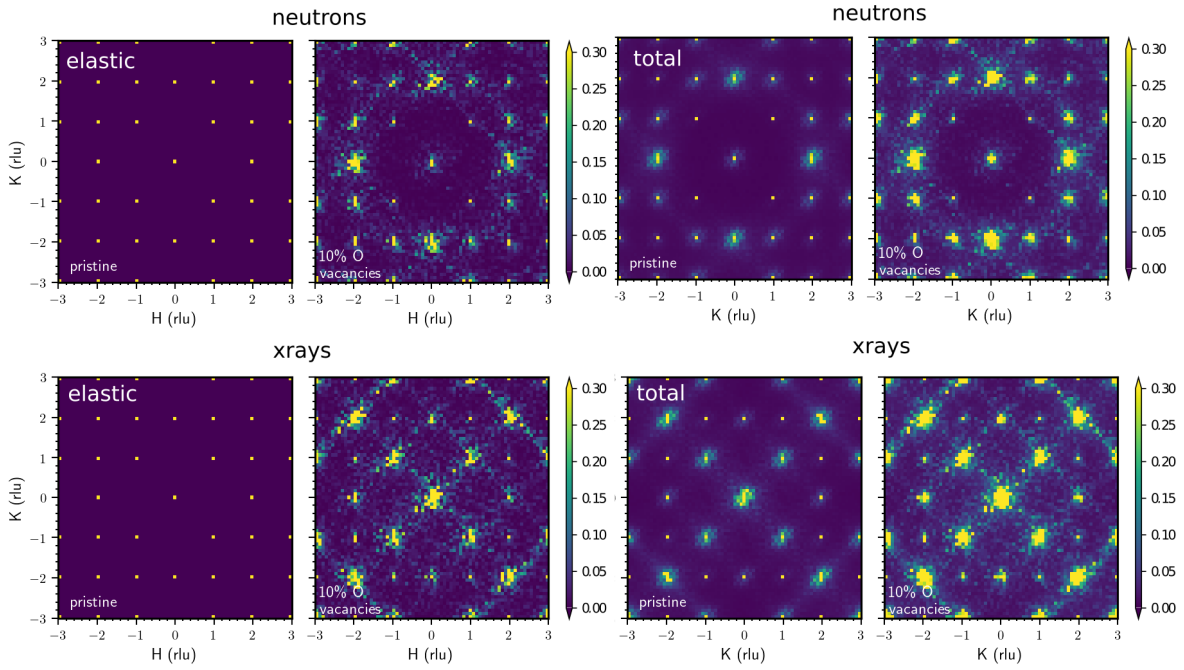

Fig. 3: Diffuse elastic scattering (left) and total scattering (right) from rutile with and without oxygen vacancies. The top rows show neutron diffuse scattering and the bottom rows show X-ray diffuse scattering.

atoms?

7.2 Diffuse scattering

Now let us see what the diffuse scattering looks like. We are going to calculate both elastic and total (energy integrated) diffuse scattering. We already covered the executable mode in the last section, so let us focus on the API from here on out. Go to the directory `examples/rutile/diffuse`. Look in the input file `input_pristine.py`. The main difference here vs. the phonon calculation is the following lines:

```
Qpoints_option = 'mesh'

Q_mesh_H = [-3,3,73]
Q_mesh_K = [-3,3,73]
Q_mesh_L = 2
```

This means we will calculate $S(\boldsymbol{Q})$ on a mesh spanning $-3 \leq Q_{H,K} \leq 3$ r.l.u. along the H and K directions [8] We only calculate at $Q_L = 2$ r.l.u., but we could specify a range for L too in which case we would get $S(\boldsymbol{Q})$ on a dense $3d$ volume in $\boldsymbol{Q}$-space. Also note that it would be more sensible to only calculate from $0 \leq Q_{H,K} \leq 3$ since we can use symmetry to

[8] Neutron scatterers call the x, y, and z directions h, k, and l respectively. Capital letters mean extended Brillouin zones; lower case mean in the first Brillouin zone. E.g. $-1/2 \leq h < 1/2$ while H can be anything.

relate $-Q_{H,k} = Q_{H,K}$. But this would complicate things right now, so I don't want to cover it. You should use common sense when you pick your $\boldsymbol{Q}$-point mesh. Here, we pick a mesh that is inefficient but is convenient for learning.

The code in `run_api.py` calculates neutron and X-ray diffuse scattering for the pristine and vacancy defected crystals. Note that now we use more trajectory blocks:

```
num_trajectory_blocks = 4
trajectory_blocks = None
num_Qpoint_procs = 8
```

This lowers the energy resolution a bit, but that is fine since we are not looking at excitations. You should set `num_Qpoint_procs` to something sensible; it should run as quickly as possible without running out of memory. Run the calculation. Go into the plotting directory `rutile/diffuse/plots` and make the plots. There are scripts to plot elastic scattering for neutrons and X-rays and total scattering for neutrons and X-rays. The results are in fig. 3. Try to reproduce these plots.

Let us briefly discuss the physics. This problem is less interesting than the hybrid perovskite below, so we won't belabor it. The main things to note are: (i) in the elastic calculations, the scattering from the pristine crystal is extremely sharp. There are Bragg peaks and nothing else. This is because we are simulating an ordered crystal with no defects. (ii) In the defected crystal, the Bragg peaks are broad and there is a diffuse "back ground." The background is from the ordering of vacancies and the relaxation of atoms around the defects. In our case, the vacancies are randomly distributed so the back ground is uniform. The strain, on the other hand, is not random. The point of diffuse scattering is that deviations from perfect order, e.g. defects, generate diffuse scattering away from the Bragg peaks. In other words, the shape of the strain field shows up in the shape of the diffuse scattering. However, since this isn't a particularly interesting problem, let's not think too hard about what it means.

Now compare total to elastic scattering. Recall, total scattering is integrated over all energies; elastic scattering is only at $\omega = 0$. Exercise: even the pristine data has diffuse intensity away from Bragg peaks in the total scattering. Where does this intensity come from? Hint: it is usually called "thermal" diffuse scattering.

7.3 Hybrid perovskite $CH_3NH_3PbI_3$

Now let us move on to another example. This example closely follows ref. [8]. We will simulate the hybrid perovskite $CH_3NH_3PbI_3$ (MAPI). MAPI is a popular material for solar energy conversion applications and is interesting for numerous basic physics reasons. People are trying hard to understand the electronic physics in MAPI, but until recently [8], no one even knew what the atoms were doing. It turns out that there is *local* ordering away from the expected cubic structure. $2d$ "pancakes" of locally correlated, tetragonally rotated octahedra exist throughout the otherwise primitive cubic crystal. The pancakes are large ($\sim$nm) and dynamic, so are likely to affect the electronic structure by dynamically modifying the potential seen by the electrons.

In this section, we will briefly look at the inelastic scattering $S(\boldsymbol{Q}, \omega)$ then spend some time seeing how to calculate the diffuse scattering and make sense of the results.

7.3.1 MD simulations

Go to the directory `$ROOT/examples/mapi/lammps`. There is only one directory, since we are interested in local ordering within the pristine structure. I.e. we won't simulate an explicitly defected crystal. The structure is a $12 \times 12 \times 12$ replica of the 12 atom primitive cubic cell. The simulation contains 20736 atoms. The MD time step is 0.25 fs; it must be small to accurately capture the motion of hydrogen. The data are written to the file every 200 steps; i.e. the effective time step in the file is every 50 fs. The system is equilibrated at 600K in an NPT ensemble, quenched to 350 K, and then run in an NVE ensemble for 200000 steps. There are 1000 steps written in the trajectory file. Go into this directory and run the simulation using `lammps`. I suggest using multiple processors; on my desktop with 16 cores, it takes a few minutes. It should produce a trajectory file called `pos.h5`. Also read the `log.lammps` file and figure out what the cubic lattice vectors are after quenching. You need them for the $S(\boldsymbol{Q}, \omega)$ calculation.

7.3.2 Phonons

Go into the phonons directory `$ROOT/examples/mapi/phonons`. Look at the input file `input.py`. It is mostly the same as the one for rutile with the main difference being the atom types have changed:

```
atom_types = ["C","N","H","H","Pb","I"]
```

The $\boldsymbol{Q}$-path is the same as before.

We use this file as input for the API. Look at the API script `run_api.py`. You should see some words like "protonated" and "deuterated". Protonated hydrogen, or "protium" is the most common hydrogen isotope: the nucleus is a single proton with no neutrons. Deuterium is less common but is stable: it has both a proton and a neutron. It turns out that protium is a strong absorber of neutrons (they form bound states with the nuclei) so scattering from crystals with hydrogen is weak and experiments are difficult. Clever experimentalists in ref. [8] synthesized MAPI crystals with hydrogen (protium) replaced by deuterium and measured the neutron diffuse scattering intensity from the deuterated crystal. To compare to this experiment, we need to account account for the fact that deuterium has a different scattering length than protium: this is as simple as changing from protium (`"H"`) to deuterium (`"2H"`) in the arguments to `PSF`:

```
atom_types = ["C","N","2H","2H","Pb","I"]
```

For convenience, we only do one MD simulation with protium in the system (i.e. using the mass of hydrogen in the equations of motion). The isotope effects (i.e. mass effects) due to replacing protium with deuterium have been investigated with neutron scattering. The effects are, at best, disputed. There are no MD potentials for deuterated MAPI, but I did check simulations using both the mass of hydrogen and the mass of deuterium and didn't find significant effects in the low energy phonons, so we should be fine using just protium in our MD calculations. Still, use caution when interpreting the results!

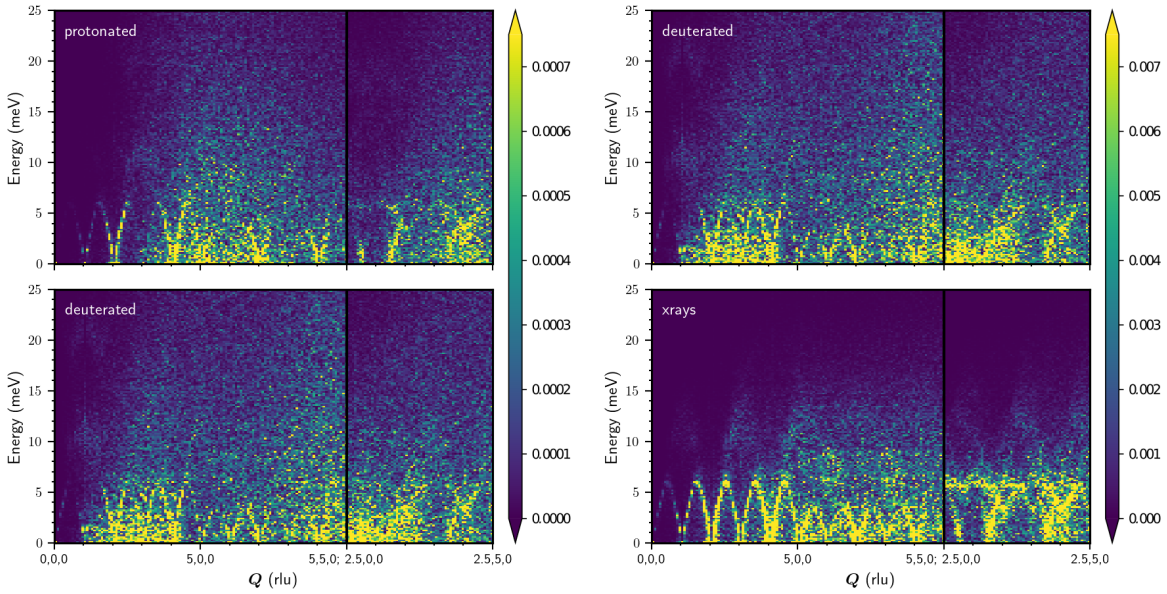

Fig. 4: Excitations $S(\boldsymbol{Q}, \omega)$ calculated using `PSF` for inelastic neutron scattering (left) and inelastic X-ray scattering (right). The left figure compares neutron scattering from protonated MAPI to deuterated MAPI. The right figure compares neutron scattering from deuterated MAPI to X-ray scattering.

For completeness, we are going to do neutron scattering calculations for both protium and deuterium. Our protonated calculations won't include the effects of absorption (classical scattering doesn't form bound states), but we might as well calculate scattering from protonated MAPI anyway.

For the X-ray calculation, we just leave hydrogen in the crystal. X-rays only see the charge of the nucleus which is the same for protium and deuterium, so there will no difference due to different scattering lengths.

Everything discussed above is already set up in `run_api.py`. Go and run it! It should run successfully and write some out put files. Go into the `mapi/phonons/plots` directory. There are scripts to plot the comparison of $S(\boldsymbol{Q}, \omega)$ for neutron scattering from protonated vs. deuterated MAPI and to plot the comparison of $S(\boldsymbol{Q}, \omega)$ for neutron scattering from deuterated MAPI vs X-ray scattering from MAPI. The results are shown in fig. 4. See if you can reproduce them!

Let us briefly discuss the physics in fig. 4:

$\boldsymbol{Q}$-dependence: Just like with rutile, observe how the intensity varies with $\boldsymbol{Q}$ in all of the plots. Compare the segments and consider which polarizations should show up where. Also note that the last segment, which is along BZ edges only, has $\omega = 0$ modes. There are no fundamental Bragg peaks at any of these $\boldsymbol{Q}$-points? This intensity is called a "soft mode." It corresponds to a phonon eigenvector that gets "frozen" in, resulting in a structural distortion. This is a signature of a displacive phase transition. Some things to think about: why is the name "soft" in soft mode appropriate? Phonons

are "oscillators," so what sense are these modes frozen in? Hint: what is frequency of the oscillation? If you want to understand what the actual eigenvector looks like, good luck! We usually resort to lattice dynamics calculations for these types of things, but MAPI is a very hard material to model: since the MA are disordered in the cubic phase, it is hard to even justify the concept of *phonons* makes sense. Still, see refs. [15, 16] for details.

protonated vs. deuterated: There isn't really anything interesting to say here. The sign of the protium scattering length is negative; the sign of the deuterium scattering length is positive. Waves scattered from protium have their phase shifted by π with respect to waves scattered from deuterium. These effects show up in the plot, but are not particularly profound.

neutrons vs. X-rays: Now look at the right figure comparing neutron scattering from deuterated MAPI to X-ray scattering. There are some obvious differences: in the neutron calculation, the acoustic phonons are only faintly distinguishable and there is a big "wall" of intensity above the acoustic phonons that we can see. In the X-ray calculation, we can see the acoustic modes more easily and the "wall" is less pronounced. Exercises: (i) How "big" are the X-ray form factors of the atoms in the CH_3NH_3 (MA) molecules compared to Pb and I? Hint: which atoms are heavier? With this in mind, what atoms contribute most of the scattering in the X-ray calculation? To belabor this point, imagine that the light atoms are completely invisible. You can enforce this by replacing their types with `"none"` in the input file. Try it in a neutron calculation leaving only Pb and I intact and compare those results to the X-ray calculation. Do they look similar? Why? (ii) The neutron calculation has a "wall" of intensity; notably, we don't really see "phonons". Conceptually, phonons are small oscillations of atoms around fixed, ordered positions. In cubic MAPI, the MA molecules rotate almost freely. Is this consistent with the interpretation of small oscillations around fixed positions? If not, should we even expect to see "phonons" in $S(\boldsymbol{Q}, \omega)$?

7.3.3 Diffuse scattering

Go into the `mapi/diffuse` directory. Open the input file `input.py` and note the $\boldsymbol{Q}$-point mesh and MD parameters. These will be the same for all diffuse scattering calculations. In particular, note that $Q_L = 2.5$ r.l.u. so that there are *no* Bragg peaks in the data.

Now look at the API script `run_API.py`. Try to understand what is being done there. Run it once you have it figured out. It should run successfully and produce three output files in `mapi/diffuse/out`. Now go into `mapi/diffuse/plots`. There are a bunch of files here... there are scripts to plot purely elastic scattering, $S(\boldsymbol{Q})$ integrated over $-1 \leq \hbar\omega \leq 1$ meV, and total scattering. For each, there are scripts to compare neutron diffuse scattering from protonated to deuterated MAPI and to compare neutron scattering from protonated MAPI to X-ray scattering from protonated MAPI. The results of all of these are shown in fig. 5. Try to reproduce these figures.

Note, there is also a script that compares the X-ray scattering calculation to the X-ray diffuse scattering experimental data in ref. [8]. This comparison is shown in §9, fig. 8.

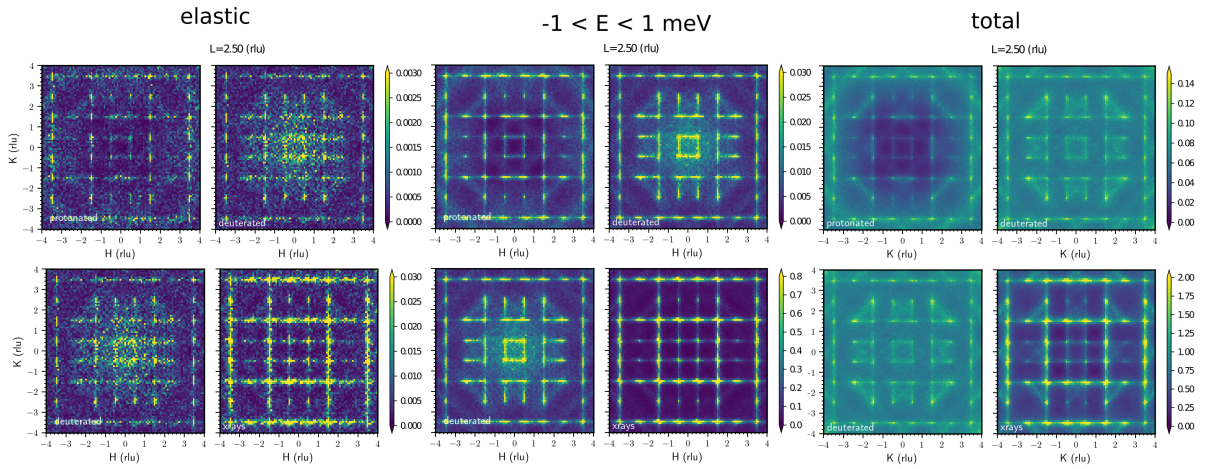

Fig. 5: Elastic (left), energy integrated (middle), and total (right) diffuse scattering from MAPI. The top rows compare neutron scattering from protonated to deuterated MAPI. The bottom rows compare neutron scattering from deuterated MAPI to X-ray scattering from protonated MAPI. The energy integrated data are integrated $-1 \leq \hbar\omega \leq 1$ meV.

Let us first think about the results in fig. 5. The elastic data are at $\omega = 0$ and are on a zone boundaries: there should be *no* intensity here in a perfectly ordered crystal. The fact that there is intensity is a clue that something cool is going on! There are no "defects" in the sense of vacancies, substitutions, etc... so what gives? The conclusion is that there are distortions within the otherwise pristine crystal away from the perfect order we ordinarily expect. We will look at that more carefully soon: for now, let us flesh out the difference between the different figures in fig. 5. The elastic data look exceptionally grainy: this is because we used a relatively short trajectory and did not "block average" much. It is a numerical artifact that can be fixed by better converging the calculation. Still, the purely elastic data show qualitatively the diffuse scattering features present in ref. [8]. The energy integrated data in the middle column look similar to the elastic data, but are much less "noisy": this is because we are integrating over multiple frequency bins, averaging out some of the noise. Note that experiments don't have perfect energy resolution and so are naturally integrated over a small energy window centered on $\hbar\omega = 0$ meV. The energy integrated data represent elastic diffuse scattering as would be seen in an experiment and the agreement with the elastic data says the experiment is a good representation of the purely elastic data (cf. fig. 8). On the other, a *diffraction* experiment measures total scattering. This is shown in the right column. Note that there is now a broad, diffuse background masking the sharp features in low energy data. Do you know where this background come from?

Let us now try to understand the diffuse scattering "rods" seen in the low energy data in fig. 5. To make sense of the diffuse scattering, we need to think in terms of Fourier transforms. Recall that $S(\boldsymbol{Q})$ is the Fourier transform of the real space density-density correlation function: let's call it $G(\boldsymbol{r})$. The *shape* of $S(\boldsymbol{Q})$ is related to the *shape* of $G(\boldsymbol{r})$: a well known example of this is that exponential real-space correlations show up as Lorenztian line shapes in $S(\boldsymbol{Q})$. Heuristically, since $S(\boldsymbol{Q})$ is a function of $\boldsymbol{Q}$, which has dimensions

$[\boldsymbol{Q}] = 1/L$, widths in $\boldsymbol{Q}$ are inversely proportional to the correlation lengths (i.e. widths in real space). Call widths in $\boldsymbol{Q}$-space Γ and correlation lengths γ. Then $\gamma \propto 1/\Gamma$.

Along the directions *perpendicular* to the rods in fig. 5, the diffuse scattering features are relatively sharp. You can tell the features are "rods" by noting that equivalent rods to the ones shown in the planes can be seen from a "top-down" view in either the $H = 2.5$ r.l.u. or $K = 2.5$ r.l.u. planes intersecting the planes in fig. 5. Anyway, these features are rods that are narrow and circular. Let's call this width $\Gamma_\perp$. Since they are circular in $\boldsymbol{Q}$-space, we deduce the correlations are circular in real space. Since the widths are narrow, the correlations lengths $\gamma_\perp = 1/\Gamma_\perp$ are large.

Note that for infinitely narrow features in $\boldsymbol{Q}$-space, $\Gamma \to 0$ and $\gamma \to \infty$. This is a physically relevant limit signalling *long ranged order*. Recall that Bragg peaks show up as very narrow features in scattering from perfectly ordered crystals!

If we look at the "width" of the rod along it's axis, i.e. the length of the rod, we note that it is very, very large. I.e. $\Gamma_\parallel \to \infty$. Then we conclude the correlation length along the axis of the rod is vanishingly small: $\gamma_\parallel = 1/\Gamma_\parallel$. We have deduced that the peculiar shape of the intensity in $S(\boldsymbol{Q})$ arises from $2d$ correlations: they have large but finite correlation lengths in the directions perpendicular to the rods, but are uncorrelated along the direction of the rods. This is why we call them "pancakes".

Still, we need to figure out *what* the correlations are. I leave this as an exercise! You can easily look up the answers by reading ref. [8]. To make progress, look at the energy integrated data in the middle column of fig. 5. In particular, compare the neutron scattering calculation for deuterated MAPI to the X-ray calculation. Do the rods look identical in the X-ray and neutron calculations? What atoms contribute to the scattering in the X-ray calculation? What about in the neutron calculation? The beauty of atomistic calculations is that, once you know what atoms to think about, you can look at the details of the simulation to make progress. That is how we solved this problem: we figured out what atoms contribute strongly to the X-ray scattering. Combined with our knowledge of the soft modes in halide perovskites (c.f. [15, 16]), we knew what to look for. We directly plotted the MD trajectory and the relevant correlations were so obvious once we knew what to look for!

7.4 Exercise: Si

As a final exercise, I challenge you to simulate silicon with `lammps` and calculate $S(\boldsymbol{Q}, \omega)$ along the same paths as we did above for rutile and MAPI. Then simulate the same system with 10% of the sites swapped with germanium. I did this with the Tersoff potential. I used the 8 atom conventional cell of silicon replicated $12 \times 12 \times 12$ times. My `lammps` scripts to run these simulations are in the directory `$Root/examples/silicon/lammps`. You can use these or write your own input files.

Once you have done the `lammps` simulations, prepare files to run `PSF`. You can use either the executable or API. Keep in mind, you need to copy the relevant MD data into the input file, define a $\boldsymbol{Q}$-point path that is commensurate with the simulation box, and set the atom types correctly. Give it a try! Calculate $S(\boldsymbol{Q}, \omega)$ for both neutrons and X-rays and for both the pristine and Ge-defected crystals. Extract the data from the output files and plot it. My scripts to simulate $S(\boldsymbol{Q}, \omega)$ are in the `silicon/phonons/answers` directory. Try to write your own first, then compare to these if you have problems.

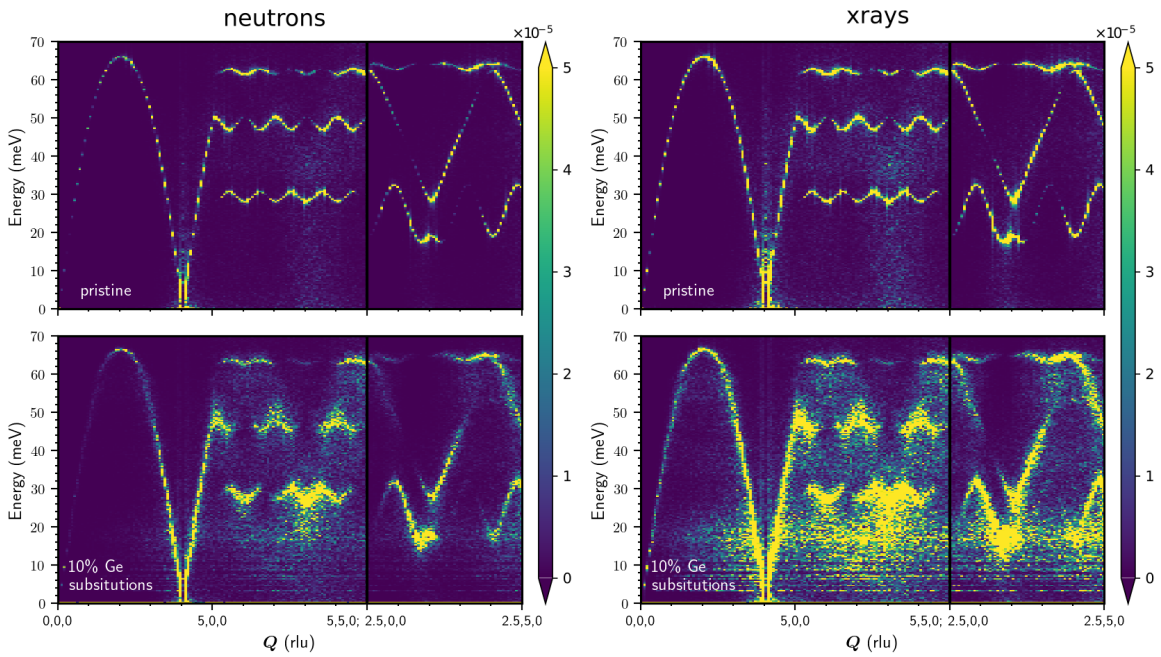

Fig. 6: Excitations $S(\boldsymbol{Q}, \omega)$ calculated using `PSF` for inelastic neutron scattering (left) and inelastic X-ray scattering (right). The top rows show scattering from a pristine silicon crystal. The bottom rows show scattering from a crystal with 10% of the silicon atoms replaced by germanium.

My results are shown in fig. 6 and yours should be qualitatively identical if you run equivalent simulations to mine. I also suggest tweaking things and seeing what changes: Pick a different $\boldsymbol{Q}$-point path. Use a $\boldsymbol{Q}$-point mesh. Pick a wrong $\boldsymbol{Q}$-point grid that is incommensurate. Use more or fewer trajectory blocks. Turn off scattering from the Ge atoms in the defect simulation (set the type to `"none"`). Each time you play with some setting, try to understand why the results look the way they do.

Also try to understand why the data in fig. 6 look the way they do. The pristine calculations are sharp while the defected calculations have broad phonons. Why? What does these mean for the lattice thermal conductivity? The pristine neutron scattering calculation and the pristine X-ray calculation look nearly identical. Why? In the defect systems, there are now two atom species and the neutron and X-ray calculations are no longer identical. Why not?

8 Accessing the output

I provide a "reader" class with methods to read the output into `numpy` arrays for you to do whatever you want with. The class is `psf.m_io.c_reader`. The reader finds the output file you specify and different data are read using different methods. The following example shows how to use the class:

```
from psf.m_io import c_reader
```

```
input_file = "psf_STRUFACS"
reader = c_reader(input_file = input_file)

reader.read_sqw()
H = reader.H; K = reader.K; L = reader.L
energy = reader.energy
sqw = reader.sqw

reader.read_energy_integrated_sqw(e_min = 1, e_max = 1)
sq_integrated = reader.sq_integrated

reader.read_energy_integrated_sqw()
sq_total = reader.sq_integrated

reader.read_elastic()
sq_elastic = reader.sq_elastic
```

An explanation of each of these methods is given below:

read_sqw(): This method reads in the entire $S(\boldsymbol{Q},\omega)$ array and the $\boldsymbol{Q}$-points and energy arrays (in meV). $S(\boldsymbol{Q},\omega)$ is inside the attribute `sqw`. The energies are in `energy`. If the data were calculated on a mesh of $\boldsymbol{Q}$-points, e.g. for a diffuse scattering cut, then the $\boldsymbol{Q}$-points spanning each edge of the mesh are read in: they are called `H`, `K`, and `L`. These correspond to the 0^{th}, 1^{st}, and 2^{nd} axes of `sqw` respectively. They are in r.l.u. If the data were calculated on a path, then two arrays are read in: `Q_rlu` and `Q_cart`. These are self explanatory. In the case of a path, the 0^{th} axis of `sqw` correspond to the $\boldsymbol{Q}$-points. The *last* axis of `sqw` is always energy.

read_elastic(): This method reads in the $\boldsymbol{Q}$-points and the elastic $S(\boldsymbol{Q},\omega=0)$ datta. There is no energy axis, only $\boldsymbol{Q}$-points. They are the same as for `read_sqw()`. The elastic data are in the `sq_elastic` array attribute.

read_energy_integrated_sqw(e_min=None,e_max=None): This method reads in the $\boldsymbol{Q}$-points and integrates $S(\boldsymbol{Q},\omega)$ between whatever energy bounds you specify. If you don't give any bounds, the minimum and maximum are used as bounds and *total* scattering is returned. The energy integrated data are in the array attribute `sq_integrated`.

9 Validation

Here, we validate the results of `PSF` by comparing to two other sources: $S(\boldsymbol{Q},\omega)$ from harmonic lattice dynamics and $S(\boldsymbol{Q})$ from experiment. For the first, we look back at the silicon calculations in the examples. In particular, consider the pristine neutron scattering calculation. $S(\boldsymbol{Q},\omega)$ was calculated on the same path as the examples using `PSF`. An analogous calculation was done with `euphonic` [2].

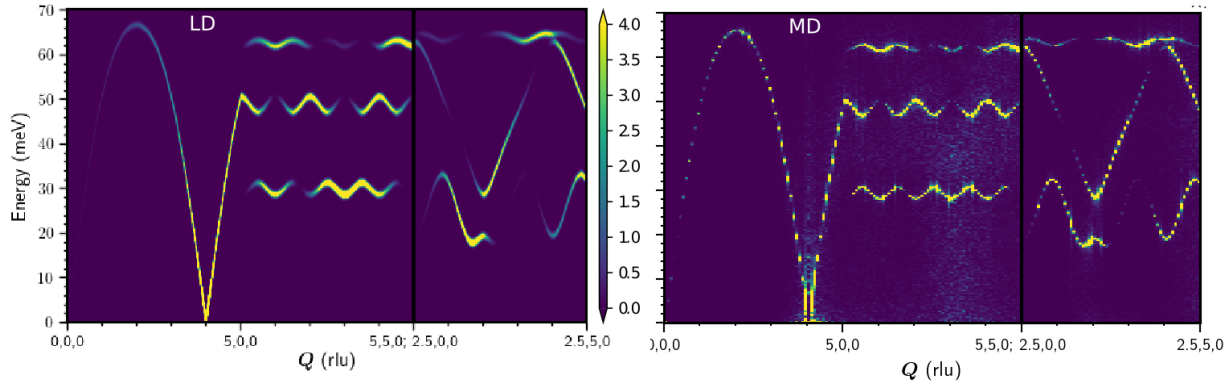

Fig. 7: $S(\boldsymbol{Q},\omega)$ calculated from harmonic lattice dynamics (left) and from `PSF` (right). The harmonic lattice dynamics calculation was done with `euphonic` [2]. The phonons are infinitely sharp, i.e. δ-functions in energy, so we broaden them with a Gaussian with FWHM=1 meV to approximate the finite widths in the MD calculation. In both methods, the temperature was 300 K.

`euphonic` calculates eq. 18. The usual input data for this method are `ab-initio` force consonants. However, to compare to the classical MD potential, I hacked an interface together to mesh `lammps` and `phonopy` to calculate the harmonic force constants. I used these force constants with `euphonic` to calculate $S(\boldsymbol{Q},\omega)$ on the same $\boldsymbol{Q}$-point path as calculated with `PSF`. The phonon peaks in $S(\boldsymbol{Q},\omega)$ are δ-functions in energy. To approximate the finite widths due to anharmonicity in the MD calculation, the `euphonic` results are broaded by a Gaussian with FWHM = 1 meV. The results of these calculations are shown in fig. 7. The agreement is perfect! The only notable difference is that the `PSF` results are *grainy* compared to the lattice dynamics results. This is due to the finite size effects mentions in 3. The lattice dynamics calculation can be done on an arbitrarily dense grid using Fourier interpolation, so it does not suffer from finite size effects.

The other way to validate my code is to compare to experiment. Here, we compare theoretical X-ray diffuse scattering results from to X-ray scattering data from experiment. See fig. 8. The calculation is the same as done for X-rays in the examples §7. The agreement is so good! I won't belabor this. See ref. [8] for a detailed assessment of the agreement between `PSF` and experiment.

10 Limitations

Only orthorhombic simulation cells are fully supported for now... the reason is that unwrapping is currently done in Cartesian space and this is difficult to do for non-orthorhombic box vectors. If `unwrap_trajectory = False`, there should be no issues but I haven't really tested this. **Use caution!** This issue can be fixed by unwrapping in reduced ("crystal") coordinates. It is actually easier in reduced coordinates, but this requires converting from Cartesian to reduced coordinates and back at every time step which is time consuming... I haven't implemented it yet.

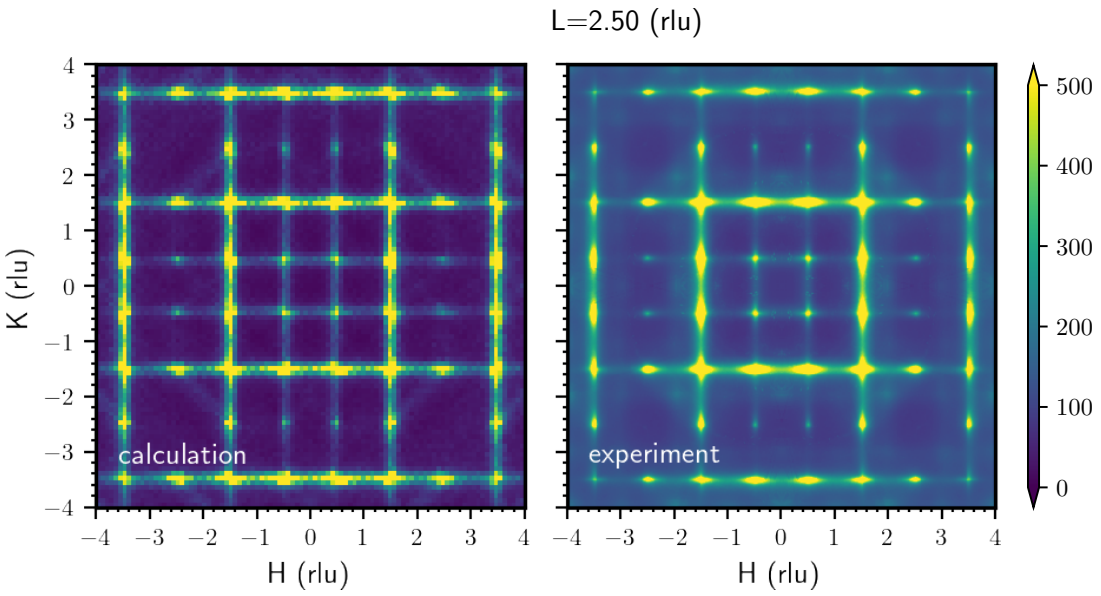

Fig. 8: Theoretical energy integrated diffuse scattering from MAPI compared to experimental X-ray diffuse scattering data from ref. [8]. The energy integrated data are integrated $-2 \leq \hbar\omega \leq 2$ meV which is roughly equal to the energy resolution in the experiment.

Another limitation is that, if the simulation box volume or shape changes throughout the simulation, then the reciprocal lattice vectors calculated at from the `lattice_vectors` argument won't really be valid. The problem is that the $\boldsymbol{Q}$-points calculated from the reciprocal lattice vectors will no longer be commensurate with the simulation cell. If the shape or volume change is small, this should not be a big problem. Just calculate the *average* lattice vectors and provide them as an argument. If the change is large... I don't know. Try running an `nvt` simulation instead... its really a "your science problem" (when it happens to me, it's a "my science problem!"). One possible solution is to read the box vectors for the whole simulation and calculate the lattice vectors from them by defining the supercell multiplicity. But I don't want to implement this.

A Glossary of parameters

atom_types:

description : The atoms types in the simulation. These are used to set the neutron scattering lengths or X-ray form factors. The code expects the types in the file are consecutive integers starting at 1 and that there is unique number for each type of atom. You are free to set these to whatever you want! e.g. if you want to calculate scattering from deuterium rather than hydrogen, set the type to 2H rather than H etc. If you want to turn scattering from an atom type off, set that type as `"none"`.

type : list of str

default : `atom_types = ["Ti", "O"]`

---

box_vectors:

description : "Lattice vectors" of the simulation box. The code assumes periodic boundary conditions and these are the vectors spanning the "unit cell" containing the simulation. In other words, these are the supercell lattice vectors. These are used to unwrap the trajectory (see `unwrap_trajectory`), so are ignored if `unwrap_trajectory = False`. If `box_vectors = None`, they are read from the file. For now, `box_vectors` are only read if `trajectory_format = "lammps_hdf5"`, so these must be given if a different format is used. The code doesn't care if the simulation is constant pressure: it calculates the time-averaged vectors from the file and uses those. If `box_vectors` are given manually for a constant pressure simulation, just use the average.

type : 3x3 list of floats or None

default : `box_vectors = None`

---

calc_sqw:

description : Whether or not to calculate the dynamic structure factor, $S(\boldsymbol{Q}, \omega)$. The alternative is to only calculate elastic scattering, $S(\boldsymbol{Q}, \omega = 0)$, which saves (a little) time since calculating $S(\boldsymbol{Q}, \omega)$ requires FFT's. Elastic scattering is always calculated.

type : bool

default : `calc_sqw = True`

---

experiment_type:

description : The experiment type: either `experiment_type = "neutrons"` for neutron scattering or `experiment_type = "xrays"` for X-ray scattering. In the case of X-ray scattering, the form factors are pre-calculated for all the $\boldsymbol{Q}$-points that you request. This is time consuming for a large number of $\boldsymbol{Q}$-points, so it is parallelized using the `multiprocessing` library for `python`. The number of processes is set by `num_Qpoint_procs`.

type : str

default : `experiment_type = "neutrons"`

---

lattice_vectors:

description : Lattice vectors of unitcell used to calculate the reciprocal lattice vectors, i.e. `lattice_vectors` determine the Cartesian coordinates of the $\boldsymbol{Q}$-points. They can be arbitrary, but keep in mind the $\boldsymbol{Q}$-points have to be commensurate with the supercell (see `box_vectors` below) to get sensible data. It is your responsibility to ensure this. These should be in whatever length units are in the trajectory file.

type : 3x3 list of floats

default :

```
lattice_vectors = [[4.59, 0.00, 0.00],
                   [0.00, 4.59, 0.00],
                   [0.00, 0.00, 2.96]]
```

---

md_num_atoms:

description : The number of atoms in the simulation. **Note: the default is not correct in general, so this should always be set by the user.**

type : int

default : `md_num_atoms = 6480`

---

md_num_steps:

description : The full number of steps in the trajectory file. Do not subtract any steps that will be omitted with `trajectory_stride`, `trajectory_skip`, or `trajectory_trim`: the code does this automatically. **Note: the default is not correct in general, so this should always be set by the user.**

type : int

default : `md_num_steps = 6250`

---

md_time_step:

description : Time step in the *file* in femtoseconds. (not the Verlet integration time step in the *simulation*!) This variable should not include the `trajectory_stride` if used. The effective step is `md_time_step`×`trajectory_stride`, but this is calculated inside the code and the user doesn't have to care about it. **Note: the default is not correct in general, so this should always be set by the user.**

type : float

default : `md_time_step = 16.0`

---

num_Qpoint_procs:

description : Number of processes to split $\boldsymbol{Q}$-points over. $\boldsymbol{Q}$-points are split-up round-robin style and then processes are spawned using the `multiprocessing` library. Note that `python` doesn't really support shared memory, so use this with caution if the trajectory is large (in the sense of how much disk space it requires) since each process has to store the entire block of trajectory data in RAM.

type : integer

default : `num_Qpoint_procs = 1`

---

output_directory:

description : Where to write the output file. If the directory doesn't exist, it is created. If `ouput_directory = None`, the current working directory is used.

type : str or None

default : `ouput_directory = None`

---

output_prefix:

description : Prefix prepended the output file. The actual file name will be whatever you define with "_STRUFACS.hdf5" appended to it. e.g. if `ouput_prefix = "psf"`, then the output file name will be "psf_STRUFACS.hdf5"

type : str

default : `ouput_prefix = "psf"`

Q_file:

description : csv file with $\boldsymbol{Q}$-points in it. It actually is delimited by spaces and not commas, so don't put any commas in it! The data in the file should have shape Nx3. It is read by `numpy.loadtxt()` so make sure it will work with that.

type : str

default : `Q_file = "Qpts.dat"`

Q_mesh_H:

description : Used if `Qpoints_option = "mesh"`. Set grid of $\boldsymbol{Q}$-points with 1^{st}-component spanning from `Q_mesh_*[0]` to `Q_mesh_*[1]` with `Q_mesh_*[2]` steps along this edge of the grid. For only one $\boldsymbol{Q}$-point along this edge, just give a single number.

type : either list [float,float,int] or float

default : `Q_mesh_H = [-3,3,41]`

Q_mesh_K:

description : Used if `Qpoints_option = "mesh"`. Set grid of $\boldsymbol{Q}$-points with 2^{nd}-component spanning from `Q_mesh_*[0]` to `Q_mesh_*[1]` with `Q_mesh_*[2]` steps along this edge of the grid. For only one $\boldsymbol{Q}$-point along this edge, just give a single number.

type : either list [float,float,int] or float

default : `Q_mesh_K = [-3,3,41]`

Q_mesh_L:

description : Used if `Qpoints_option = "mesh"`. Set grid of $\boldsymbol{Q}$-points with 3^{rd}-component spanning from `Q_mesh_*[0]` to `Q_mesh_*[1]` with `Q_mesh_*[2]` steps along this edge of the grid. For only one $\boldsymbol{Q}$-point along this edge, just give a single number.

type : either list [float,float,int] or float

default : `Q_mesh_L = 1`

Q_path_start:

description : The starting vertices on the path thru $\boldsymbol{Q}$ space. Each segment of the path starts at the $\boldsymbol{Q}$-point listed in `Q_path_end` and ends at the corresponding $\boldsymbol{Q}$-point listed in `Q_path_end`. Note: if you want consecutive segments thru $\boldsymbol{Q}$-space, just set the next vertex in `Q_path_start` to the previous one in `Q_path_end`. Also see `Q_path_steps`.

type : Nx3 list of floats

default :

```
Q_path_start = [[0,0,0],
                [2,0,0]]
```

Q_path_end:

description : The ending vertices on the path thru $\boldsymbol{Q}$ space.Each segment of the path starts at the $\boldsymbol{Q}$-point listed in `Q_path_end` and ends at the corresponding $\boldsymbol{Q}$-point listed in `Q_path_end`. Note: if you want consecutive segments thru $\boldsymbol{Q}$-space, just set the next vertex in `Q_path_start` to the previous on in `Q_path_end`. Also see `Q_path_steps`.

type : Nx3 list of floats

default :

```
Q_path_end = [[2,0,0],
              [2,0,2]]
```

Q_path_steps:

description : Number of steps along each segment defined by `Q_path_start` and `Q_path_end`. Since the most common case is consecutive segments thru $\boldsymbol{Q}$-space, the end of each segment doesn't actually include the $\boldsymbol{Q}$-point in `Q_path_end`. That way, the $\boldsymbol{Q}$-point isn't repeated in paths with consecutive segments. If you want non-consecutive segments, then you either have to live with not actually getting the last $\boldsymbol{Q}$-point of each segment, or do separate calculations for the individual segments and put the data together yourself.

type : list of integers

default :

```
Q_path_steps = [21,21]
```

Qpoints_option:

description : How the set of $\boldsymbol{Q}$-points will be generated. Allowed values:

- `"mesh"` : Calculate $S(\boldsymbol{Q},\omega)$ on and automatically generated uniform 3D mesh of $\boldsymbol{Q}$-points. See the `Q_mesh_*` arguments.
- `"path"` : Calculate $S(\boldsymbol{Q},\omega)$ on an automatically generated path thru $\boldsymbol{Q}$-space. See the `Q_path_*` arguments.
- `"file"` : Calculate $S(\boldsymbol{Q},\omega)$ on an arbitrary set of $\boldsymbol{Q}$-points read from a csv file. See the `Q_file` argument.

type : str

default : `Qpoints_option = "mesh"`

---

trajectory_file:

description : Path to the trajectory file. Must be an `hdf5` database written in on of the formats specified in §6.

type : str

default : `trajectory_file = "pos.h5"`

---

trajectory_blocks:

description : Which blocks of trajectory data to actually use. The trajectory is split into `num_trajectory_blocks` blocks, but only the ones given in this list are used. This is useful to e.g. skip every other block of data to ensure that the results in each block that are averaged together are uncorrelated. If `trajectory_blocks = None`, all blocks are used.

type : list or None

default : `trajectory_blocks = None`

---

trajectory_file:

description : Path to the trajectory file. Must be an `hdf5` database written in on of the formats specified in §6.

type : str

default : `trajectory_file = "pos.h5"`

---

trajectory_format:

description : Format of the trajectory data file. Current supported args:

`"lammps_hdf5"` :
Must be written using commands like:

```
dump pos all h5md 10 pos.h5 position species
dump_modify pos sort id
```

in your `lammps` input file. Note that `lammps` wraps the positions by default, so `"unwrap_trajectory" = True` should be set if you are using `"lammps_hdf5"` as the trajectory format. See §6 for the format.

`"user_hdf5"` :
A standard format made by converting text files written by `lammps`, `vasp`, etc. I don't provide any converters as of now, but you should be able to make one fairly easily if you try and I will be glad to help if not. You could unwrap the trajectories while making the files so you don't have to do it again while running the code. See §6 for the format.

`"external"` : Arrays of positions and atom types are passed into the code as external arguments. Only works with the API.

type : str

default : `trajectory_format = "lammps_hdf5"`

---

trajectory_skip:

description : Skip this many steps at the beginning of the trajectory. Useful to e.g. skip all steps in the beginning of the file if volume fluctuations are large during equilibration. (Really, you shouldn't write the data in the transient regime to the file, but if you do, use this).

type : int

default : `trajectory_skip = 0`

---

trajectory_stride:

description : Only use every n^{th} step. Eseful if the sampling in the file is more frequent than needed.

type : int

default : `trajectory_stride = 1`

---

trajectory_trim:

description : Skip this many steps at the end of the trajectory.

type : int

default : `trajectory_trim = 0`

unwrap_trajectory:

description : Whether or not to "unwrap" the trajectory. By default, `lammps` moves atoms that cross the simulation box boundary back to the other side. This can cause the trajectory to jump discontinuously by a box vector, causing issues with the Debye-Waller factor among other things.

type : bool

default : `unwrap_trajectory = True`

B Dynamic structure factor from harmonic lattice dynamics

Here, we quote the expressions for the DSF that can be derived from harmonic lattice dynamics. The derivations aren't particularly instructive, so I wont provide them. For details, see refs. [1, 2, 17].

The DSF is

$$S(\boldsymbol{Q},\omega) = \sum_{ij} b_i b_j \int \frac{dt}{2\pi} \exp(i\omega t) \langle \exp(-i\boldsymbol{Q}\cdot\hat{\boldsymbol{r}}_i(t)) \exp(i\boldsymbol{Q}\cdot\hat{\boldsymbol{r}}_j(0)) \rangle. \tag{12}$$

In the classical approximation, we replaced $\hat{\boldsymbol{r}}_i(t) \to \boldsymbol{r}_i(t)$ with classical positions. As mentioned, the other common way to approach this equation is assume displacements are small, i.e. $\hat{\boldsymbol{r}}_i(t) \to \hat{\boldsymbol{r}}_{ia} = \boldsymbol{r}^{(0)}_{ia} + \hat{\boldsymbol{x}}_{ia}(t) = \boldsymbol{R}_i + \boldsymbol{\tau}_a + \hat{\boldsymbol{x}}_{ia}(t)$ with i labeling the unitcell and a labeling the atom in the unitcell. Then $\hat{\boldsymbol{x}}_{ia}(t)$ is the displacement operator, which can be expressed in terms of phonon creation/annihilation operators. Plugging this in and simplifying, we eventually arrive at a term that looks like

$$\begin{aligned} S(\boldsymbol{Q},\omega) &= \sum_{\boldsymbol{R}ab} b_a b_b \exp(W) \times \\ \exp(-i\boldsymbol{Q}\cdot\boldsymbol{R}) \exp(-i\boldsymbol{Q}\cdot(\boldsymbol{\tau}_a - \boldsymbol{\tau}_b)) &\int \frac{dt}{2\pi} \exp(i\omega t) \exp(\langle \hat{U}_{\boldsymbol{R}a} \hat{V}_{\boldsymbol{0}b} \rangle) \end{aligned} \tag{13}$$

with $\hat{U}_{\boldsymbol{R}a} = -i\boldsymbol{Q}\cdot\hat{\boldsymbol{x}}_{\boldsymbol{R}a}(t)$ and $\hat{V}_{\boldsymbol{0}b} = i\boldsymbol{Q}\cdot\hat{\boldsymbol{x}}_{\boldsymbol{0}b}(0)$; see §3.3 and 3.4 in Squires [1]. $\exp((\langle \hat{U}^2_{\boldsymbol{R}a} \rangle + \langle \hat{V}^2_{\boldsymbol{0}b} \rangle)/2) \equiv \exp(W)$ is the *Debye-Waller* factor (DWF); it corresponds to the attenuation of intensity from thermal motion of the atoms. I.e. as the atoms wiggle around, they stray away from the average positions and the intensity at e.g. Bragg peaks, which is scattering from the average positions, decreases. This expression is exact (in the harmonic approximation), but is intractable as it stands. Following §3.5 in Squires, we expand

$$\exp(\langle \hat{U}_{\boldsymbol{R}a} \hat{V}_{\boldsymbol{0}b} \rangle) = \sum_n \frac{\langle \hat{U}_{\boldsymbol{R}a} \hat{V}_{\boldsymbol{0}b} \rangle^n}{n!} = 1 + \langle \hat{U}_{\boldsymbol{R}a} \hat{V}_{\boldsymbol{0}b} \rangle + \frac{1}{2} \langle \hat{U}_{\boldsymbol{R}a} \hat{V}_{\boldsymbol{0}b} \rangle^2 + \cdots \tag{14}$$

Each term of order n corresponds to a scattering process where n phonons are created or annihilated. E.g. the first term, $n = 0$, corresponds to creating 0 phonons; this is Bragg scattering. The next term, $n = 1$, is one-phonon scattering. This corresponds to creating/annihilating one phonon in the m^{th} state; the momentum of the phonon is $\pm\boldsymbol{Q}$ and the energy is $\pm\omega$, changing the momentum of the neutron by $\mp\boldsymbol{Q}$ and its energy by $\mp\omega$. We will study the 0^{th} and 1^{st} order terms in detail. Higher order terms correspond to creating/annihilating multiple phonons at once which is a very low probability phenomenon. We won't consider terms beyond one-phonon.

B.1 Bragg scattering

The elastic structure factor, corresponding to 0-phonon processes, is

$$S_0(\boldsymbol{Q}) = \delta(\boldsymbol{Q} - \boldsymbol{G}) \left| \sum_a b_a \exp(i\boldsymbol{Q} \cdot \boldsymbol{\tau}_a) \exp(W_a) \right|^2 \tag{15}$$

with $W_a = -\langle (\boldsymbol{Q} \cdot \hat{\boldsymbol{x}}_{\boldsymbol{0}a}(0))^2 \rangle$ the DWF. The factor $f(\boldsymbol{Q}) \sim |\sum_a b_a \exp(-i\boldsymbol{Q} \cdot \boldsymbol{\tau}_a)|^2$ is the *unitcell form factor*; it is the Fourier transform of the ordering of the scattering centers in a single unitcell. It serves to modulate the intensity at the Bragg peaks in a way that depends on the symmetry (space group) of the crystal. The neutron is a wave; the phases of the waves scattered off each atom depend on the phase of the neutron wave at the atom. In other words, the scattered waves interfere. For particular $\boldsymbol{Q}$ and particular unitcells, it is possible to have purely constructive interference (very intense scattering peaks), purely destructive interference (no scattering), and something in between. The variation of the structure factor (via the form factor) with $\boldsymbol{Q}$ is what allows scientists to determine the space group of crystals.

B.2 One-phonon scattering

The one-phonon term is

$$\begin{gathered} S_1(\boldsymbol{Q}, \omega) = \sum_{\boldsymbol{R}ab} b_a b_b \exp(W) \times \\ \exp(-i\boldsymbol{Q} \cdot \boldsymbol{R}) \exp(-i\boldsymbol{Q} \cdot (\boldsymbol{\tau}_a - \boldsymbol{\tau}_b)) \int \frac{dt}{2\pi} \exp(i\omega t) \langle \hat{U}_{\boldsymbol{R}a} \hat{V}_{\boldsymbol{0}b} \rangle. \end{gathered} \tag{16}$$

In the matrix element

$$\langle \hat{U}_{\boldsymbol{R}a} \hat{V}_{\boldsymbol{0}b} \rangle = \langle (\boldsymbol{Q} \cdot \hat{\boldsymbol{x}}_{\boldsymbol{R}a}(t))(\boldsymbol{Q} \cdot \hat{\boldsymbol{x}}_{\boldsymbol{0}b}(0)) \rangle \tag{17}$$

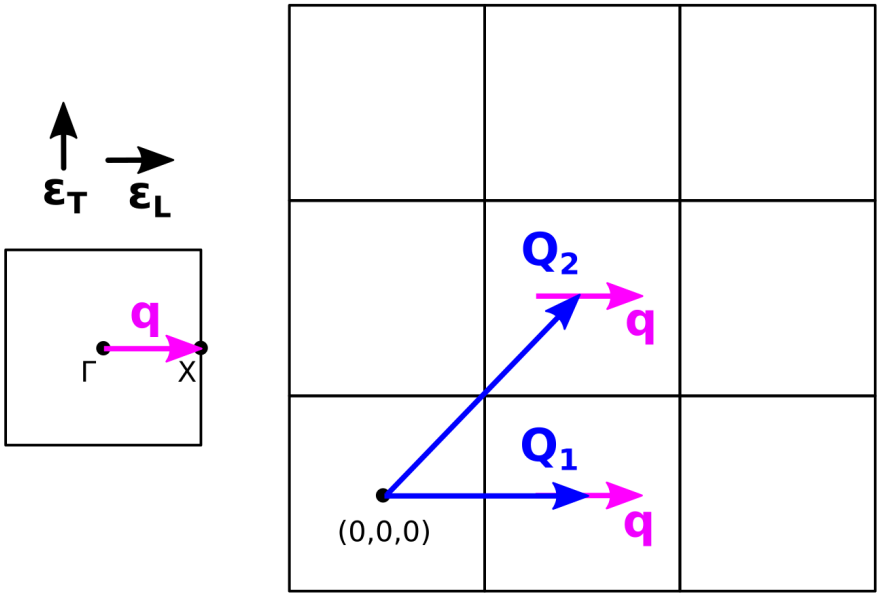

Fig. 9: Diagram showing $\boldsymbol{Q}$ and $\boldsymbol{\epsilon}_{\boldsymbol{q}\nu}$ for a few particular locations and modes.

we can use the known expressions for atom displacements in terms of phonon creation and annihilation operators and then evaluate all matrix elements. The result is

$$S_1^{(\pm)}(\boldsymbol{Q},\omega) = \sum_{\boldsymbol{q}\nu} \frac{1}{2\omega_{\boldsymbol{q}\nu}} \left| \sum_a \frac{\boldsymbol{Q}\cdot\boldsymbol{\epsilon}^a_{\boldsymbol{q}\nu}}{\sqrt{m_a}} b_a \exp(i(\boldsymbol{Q}\pm\boldsymbol{q})\cdot\boldsymbol{\tau}_a)\exp(W_a) \right|^2 \times \left(n_{BE}(\omega_{\boldsymbol{q}\nu}) + \frac{1}{2} \pm \frac{1}{2}\right) \delta(\omega \pm \omega_{\boldsymbol{q}\nu}) \delta(\boldsymbol{Q}\pm\boldsymbol{q}-\boldsymbol{G}). \tag{18}$$

where the upper (lower) sign corresponds to creating (annihilating) one phonon. $n_{BE}(\omega_{\boldsymbol{q}\nu})$ is the Bose-Einstein function. This makes sense because the probability of annihilating a phonon ought to depend on the probability of there already being a phonon etc. $\boldsymbol{\epsilon}_{\boldsymbol{q}\nu}$ is the phonon eigenvector and $\omega_{\boldsymbol{q}\nu}$ is the phonon frequency. $\boldsymbol{q}$ are "reduced" wave-vectors that are restricted to the first Brillouin zone. I.e $\boldsymbol{Q} = \boldsymbol{G} \pm \boldsymbol{q}$ is an extended wave-vector that contains two parts: (i) the nearest Bragg peak, $\boldsymbol{G}$ and the phonon wave-vector $\boldsymbol{q}$ which determines the phonon dispersions.

Note that the phase $\exp(\pm i\boldsymbol{q}\cdot\boldsymbol{\tau}_a)$ differs from Squires eq. 3.120. This is due to the phase convention I use for Fourier transforms which are the same as what Phonopy uses: `https://phonopy.github.io/phonopy/dynamic-structure-factor.html`.

The interpretation of these components of eq. 18 is fairly straightforward. Just like in the Bragg scattering case, the $f(\boldsymbol{Q}) \sim |\sum_a b_a \exp(-i(\boldsymbol{Q}\pm\boldsymbol{q})\cdot\boldsymbol{\tau}_a)|^2$ unit-cell form factor term depends on the symmetry of the crystal (and of the phonon displacement). The symmetry dependence contains a vast amount of information, but we already discussed this above when looking at Bragg scattering so let's not dwell on it.

The term $\sim |\boldsymbol{Q}\cdot\boldsymbol{\epsilon}^a_{\boldsymbol{q}\nu}|^2$ is equally important. $\boldsymbol{\epsilon}_{\boldsymbol{q}\nu}$ is only defined in the first Brillouin

zone, while $\boldsymbol{Q}$ can be anything. Suppose we want to measure dispersion from Γ to X in a cubic crystal. See fig. 9: the path from Γ to X is shown in the 1st BZ on the left. The reduced wave vector $\boldsymbol{q}$ points towards X. How do we figure out which branch is which in $S(\boldsymbol{Q}, \omega)$? Consider $\boldsymbol{Q}_1$: $\boldsymbol{Q}_1 \cdot \boldsymbol{\epsilon}_{\boldsymbol{q}\nu}$ is zero for eigenvectors that are perpendicular to $\boldsymbol{Q}_1$ and non-zero for ones that point at least somewhat in the direction of $\boldsymbol{Q}_1$. In this case, $\boldsymbol{Q}_1 \parallel \boldsymbol{q}$, so eigenvectors with $\epsilon_{\boldsymbol{q}\nu} \parallel \boldsymbol{q}$ show up strongest in $S(\boldsymbol{Q}, \omega)$ at $\boldsymbol{Q}_1$. Recall that $\epsilon_{\boldsymbol{q}\nu} \parallel \boldsymbol{q}$ means "longitudinal"! So if $\boldsymbol{Q} \parallel \boldsymbol{q}$, the most intense phonons are longitudinal. On the other hand, consider $\boldsymbol{Q}_2$. In this case, $\boldsymbol{Q}_2$ isn't quite parallel or perpendicular to $\boldsymbol{q}$. Still, $\boldsymbol{Q}_2 \cdot \boldsymbol{\epsilon}_{\boldsymbol{q}\nu}$ is strongest for eigenvectors parallel to $\boldsymbol{Q}_2$, so we know that the brightest phonons at $\boldsymbol{Q}_2$ are the ones with $\boldsymbol{\epsilon}_{\boldsymbol{q}\nu}$ that point along $\boldsymbol{Q}_2$. This logic can be extended to arbitrary $\boldsymbol{Q}$ and is an extremely useful concept when interpreting inelastic scattering data.

The final thing to note is that $S(\boldsymbol{Q}, \omega) \sim \boldsymbol{Q}^2$. The intensity grows as $\boldsymbol{Q}^2$, so the strongest signal is far from $\boldsymbol{Q} = (0, 0, 0)$. This is useful to know, but there is a complication: the DWF is $\exp(W) \sim \exp(-(\boldsymbol{Q} \cdot \boldsymbol{x})^2)$, i.e. is Gaussian in $\boldsymbol{Q}$. For large $|\boldsymbol{Q}|$, the DWF squashes the intensity. In neutrons scattering, something else that usually kills the intensity before the DWF does: conservation of energy! There are called *kinematic constraints* due to conservation of total energy of the neutron+crystal system. Different results are true for X-ray scattering and I won't go into the details here. Rather, just be aware that the $\boldsymbol{Q}$-range actually accessible in a experiment if finite and, in-fact, usually only a subset of the kinematically accessible $\boldsymbol{Q}$-points are measured at all: neutron scattering is time consuming and focus is usually paid to physically interesting regions of $\boldsymbol{Q}$-space. With this is mind, I suggest you consider using simulated $S(\boldsymbol{Q}, \omega)$ to inform your experiments!

References

[1] Gordon Leslie Squires. *Introduction to the theory of thermal neutron scattering*. Courier Corporation, 1996.

[2] Rebecca Fair et al. "Euphonic: inelastic neutron scattering simulations from force constants and visualization tools for phonon properties". In: *Journal of Applied Crystallography* 55.6 (2022).

[3] Martin T Dove and Martin T Dove. *Introduction to lattice dynamics*. 4. Cambridge university press, 1993.

[4] Z. J. Morgan et al. "Rmc-Discord: Reverse Monte Carlo Refinement of Diffuse Scattering and Correlated Disorder from Single Crystals". In: *Journal of Applied Crystallography* 54.6 (Dec. 2021), pp. 1867–1885. ISSN: 1600-5767. DOI: 10.1107/S1600576721010141.

[5] Reinhard B Neder and Thomas Proffen. *Diffuse Scattering and Defect Structure Simulations: A cook book using the program DISCUS*. Vol. 11. OUP Oxford, 2008.

[6] Michael P Allen and Dominic J Tildesley. *Computer simulation of liquids*. Oxford university press, 2017.

[7] Wei Jin et al. "Dynamic structure factor and vibrational properties of SiO 2 glass". In: *Physical Review B* 48.13 (1993), p. 9359.

[8] Nicholas J. Weadock et al. "The nature of dynamic local order in CH3NH3PbI3 and CH3NH3PbBr3". In: *Joule* (2023). ISSN: 2542-4351. DOI: https://doi.org/10.1016/j.joule.2023.03.017. URL: https://www.sciencedirect.com/science/article/pii/S2542435123001290.

[9] Léon Van Hove. "Correlations in space and time and Born approximation scattering in systems of interacting particles". In: *Physical Review* 95.1 (1954), p. 249.

[10] Thomas F Harrelson et al. "Computing inelastic neutron scattering spectra from molecular dynamics trajectories". In: *Scientific reports* 11.1 (2021), pp. 1–12.

[11] Tomofumi Zushi et al. "Effect of a SiO 2 layer on the thermal transport properties of¡ 100¿ Si nanowires: A molecular dynamics study". In: *Physical Review B* 91.11 (2015), p. 115308.

[12] Shiyun Xiong et al. "Native surface oxide turns alloyed silicon membranes into nanophononic metamaterials with ultralow thermal conductivity". In: *Physical Review B* 95.18 (2017), p. 180301.

[13] F Sears Varley. "Neutron scattering lengths and cross sectioirn". In: *Neutron news* 3.3 (1992), pp. 29–37.

[14] PJ Brown et al. "Intensity of diffracted intensities". In: (2006).

[15] Tyson Lanigan-Atkins et al. "Two-dimensional overdamped fluctuations of the soft perovskite lattice in CsPbBr3". In: *Nature materials* 20.7 (2021), pp. 977–983.

[16] Federico Brivio et al. "Lattice dynamics and vibrational spectra of the orthorhombic, tetragonal, and cubic phases of methylammonium lead iodide". In: *Physical Review B* 92.14 (2015), p. 144308.

[17] K Sturm. "Dynamic structure factor: An introduction". In: *Zeitschrift für Naturforschung A* 48.1-2 (1993), pp. 233–242.