PlasticSim Manual

1. Running PlasticSim

PlasticSim does not currently have a user interface and therefore requires the user to be comfortable editing Java source code. See §0

2. Editing the source code

To view the source code, navigate to https://www.github.com/ericdill/PlasticSim. The source code was developed with Java 1.7 in the Eclipse IDE (Kepler). To edit the source code, the Java Platform (JDK) and the Eclipse IDE must be installed. For ease of use it is recommended to install the Eclipse extension "EGit" as the github repository can be cloned directly into Eclipse and it will be trivial to update the source code as the main repository is edited.

2.1. Installing the Java Development Kit

This software was developed with Java 1.7 revision 21 (1.7u21). I advise working with Java 1.7, but different versions will likely not have any issues. Download your Java Development Kit (JDK) of choice from http://www.oracle.com/technetwork/java/javase/downloads/index.html and then install it like a standard program.

2.2. Installing Eclipse

This software was developed in the Eclipse Integrated Development Environment (IDE), Kepler edition. The software repository is configured to work with the Eclipse IDE and the setup is very easy if you decide to use Eclipse. Eclipse does not need to be installed, just downloaded and extracted from the zipped file. To download Eclipse, navigate to http://www.eclipse.org/downloads/ and choose your desired flavor. I use "Eclipse Standard" edition. Select the appropriate version (32-bit or 64-bit) and commence the download on the following page. Once downloaded, navigate to the download location and you should see a file named something like "eclipse-standard-kepler-SR1-win32-x86_64.zip" which needs to be extracted into a folder (try right-clicking on it and look for the word "extract"). Once extracted Eclipse is

operational and can be run by double-clicking on “eclipse.exe” within the extracted folder. In Windows, I like to copy the folder into my “Program Files” directory and then create a shortcut from “eclipse.exe” to the Desktop or Taskbar, but this is not a necessary step.

Run eclipse by double clicking on “eclipse.exe” or the shortcut that you created. You will be greeted by a sequence of screens where one will ask you to “Select a workspace.” For most users the default location is fine, so click the “Use this as default and do not ask me again” checkbox and click ok. If you decide that you would like a workspace in a different location, you can change that from within the IDE after it loads (**File**->**Switch Workspace**).

Eclipse will now load into the “Welcome to Eclipse” screen. Feel free to navigate around and explore or just go directly to the workbench by clicking the arrow in the top right corner labeled “Workbench.”

To configure your new Eclipse install to access the PlasticSim source code, you will need to install EGit. See the “Installing EGit” section for how to install this. Return to this point once you have installed EGit. Moving forward, I am assuming that you have EGit installed. If you haven’t yet done that, please go do it. It makes editing the source code and keeping it in sync with my changes extremely easy.

To add PlasticSim to your Eclipse IDE, click on **File**, then **Import**, then expand “Git,” select “Projects from Git and click “Next”. Then click “Clone URI” and click “Next.” Type “**https://github.com/ericdill/PlasticSim.git**” into the URI: text box. Alternatively you can navigate to https://github.com/JamesDMartin/PlasticSim and click the ‘copy to clipboard’ icon on the right hand side of the web page and the little popup should change from “copy to clipboard” to “copied!” Once that has been copied, return to Eclipse and right click in the URI box and click paste (or Ctrl + V). After the URI box has is populated, click Next, then Next again, one more time, make sure that “Import existing projects” is selected and click next, then click Finish.

You will now need to clone another project, this time located at “**https://github.com/ericdill/GlobalPackages.git**”. Do this the exact same way that you just did added PlasticSim to the workspace.

At this point you will return to the Eclipse IDE with two new folders in the “Package Explorer” window on the left hand side of the screen named something like “PlasticSimPrototype1 [PlasticSim master]” and “GlobalPackages [GlobalPackages master]”. Expand the PlasticSimPrototype1 project by double clicking on the root folder or clicking the arrow to the left of the name. There is a default runnable file called “PlasticSim.jar” that will run a version that I compiled. There will be a folder called “src” which contains the source code. There is also a folder named “doc” which contains this file among a few others. In the src folder there are nine package folders containing various aspects of the program source files. To run PlasticSim in the Eclipse IDE navigate to the “gui” folder and double-click “**SimulationTypes/RunSimulationThread.java**”. Then click the green circle with the play arrow in it (arrow in Figure 1), click “Run As” and then click “Java Application.”

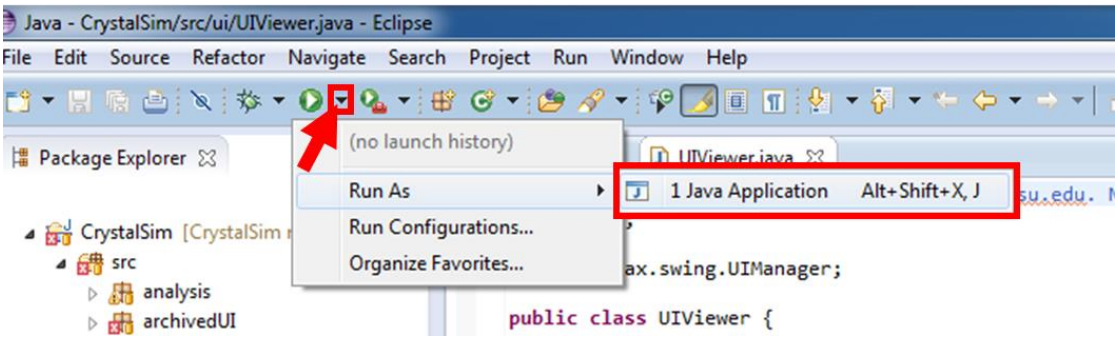

Figure 1. Running a program from the Eclipse IDE.

Because I've included various files that were useful at some point in the past but are now broken due to changes in other files, a popup will appear that is telling you that there are "Errors in the required project(s): Global Packages and PlasticSim. Proceed with launch?" The errors that are in these projects do not affect the runtime behavior of PlasticSim, so go ahead and "Proceed." If you so desire, you can check the box "Always launch without asking" to never see this message again. I would advise against this, but it's up to you. At this point, you should see a GUI that resembles **Error! Reference source not found.**.

ECLIPSE PROTIP: Add line numbers by right clicking just to the left of the text in the editor and click "Show Line Numbers."

ECLIPSE PROTIP: Change the font size by going to Window -> Preferences. A popup will appear, go to General -> Colors and Fonts. Now in the right side of that window click on "Java" and then "Java Editor Text Font" and then "Edit" on the right hand side of the window. I like 14 point font, but I would not change the font style. Looking at source code that is not monotonically spaced is just weird.

Figure 2. Click where the arrow is to install EGit from the Eclipse Marketplace. Yes, I realize that it says "Uninstall" in this image, but it will say "Install" if you have not yet installed it.

2.3. Installing EGit

Installing EGit is very easy from within the Eclipse IDE. Navigate to "Help" on the menu bar and then "Eclipse Marketplace." Once this loads, search for "EGit" and click the "Install" button. (I realize that my image shows an "Uninstall" button; that's because I already have it installed. If you don't have it installed your button will say "Install.") Click through the installation, accepting the various agreements. Once finished, Eclipse will require a restart. After restarting you will have EGit installed.

3. Program Design

PlasticSim is designed to provide a single entry point (RunSimulationThread.java) for a set of simulation codes that allow the user to simulate the plastic crystalline lattice of CBr_4 and calculate the corresponding

2D diffraction patterns along generic reciprocal lattice directions. The general program flow is illustrated in Figure 3 with the three main components of PlasticSim being the construction of the lattice, the minimization of the lattice and the calculation of the corresponding diffraction patterns. Visualization of the calculated diffraction patterns is handled by Ramdog (http://www.github.com/JamesDMartin/Ramdog).

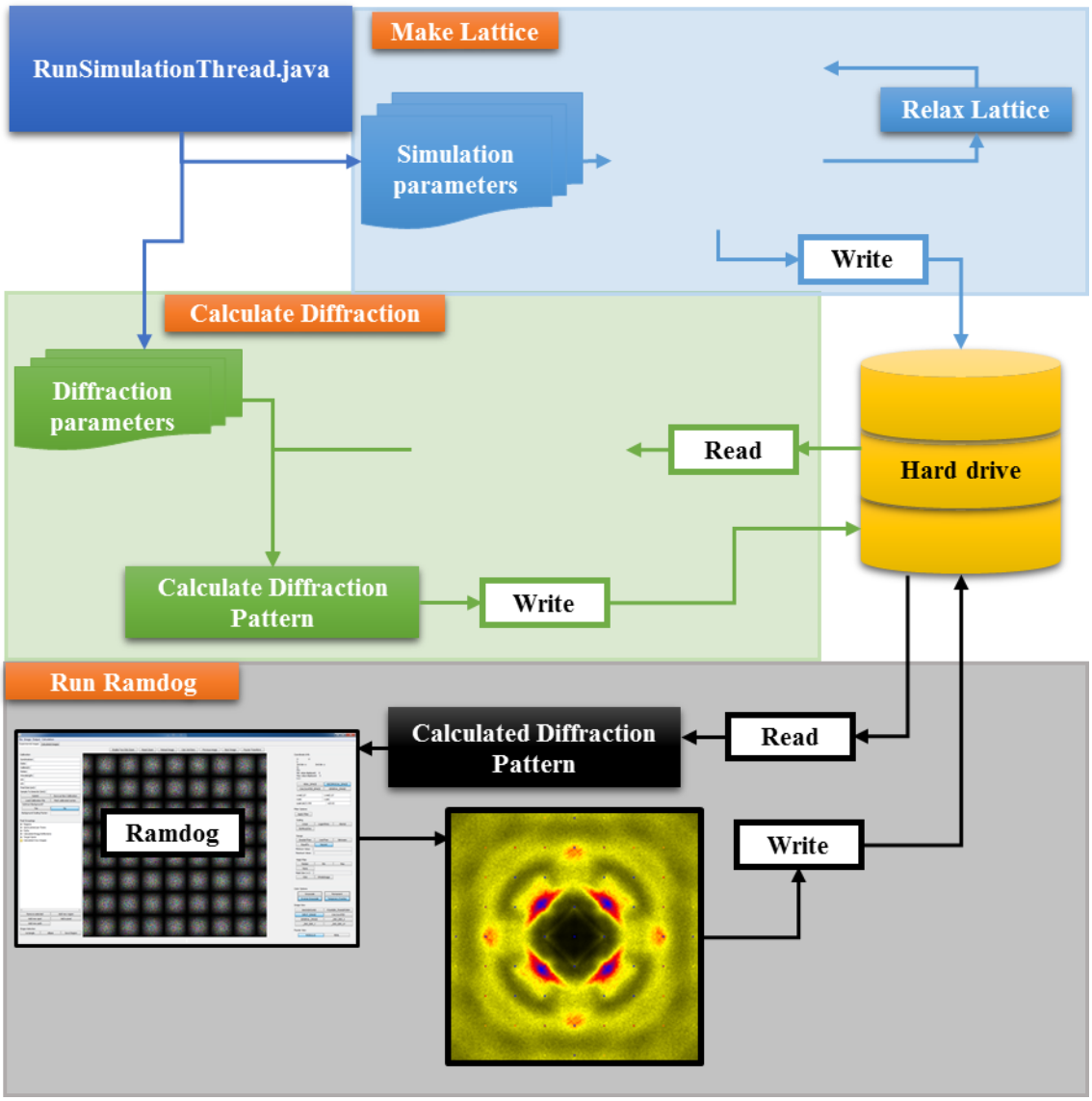

Figure 3. General program flow of PlasticSim.

3.1. Lattice Construction

Simulating the plastic crystalline lattice begins by constructing a (10×10×10 to 30×30×30) supercell where each tetrahedral molecule is placed on an FCC lattice site with an orientation based on one of a few ordering motifs, more fully described in Chapter 5 of my dissertation.{Dill, 2014 #27000} Five ordering motifs are currently implemented as part of the `InitialLatticeTypes` enum:

- Random orientation

- Six D_{2d} orientations
- 1st shell Monoclinic
- 2nd shell Monoclinic
- 3rd shell Monoclinic "star"

Random Orientation. The random orientation lattice motif distributes tetrahedra onto all FCC lattice sites in the supercell and then independently rotating each tetrahedron by three randomly generated angles around each of the Cartesian axes.

Six D_{2d} Orientations. There are six orientations in which the $\bar{4}$ axis of a tetrahedron can be aligned with the Cartesian axes. These six orientations are randomly distributed onto all FCC lattice sites in the supercell.

1st shell Monoclinic. The low temperature solid phase of carbon tetrabromide adopts a non-plastic monoclinic ($C2/c$) superstructure (Z=32), with cell parameters a=21.441(10) Å, b=12.116(6) Å, c=21.012(8) Å and β=110.91(3)°, and four molecules in the asymmetric unit.{{Tse, 1985 #26700;Coulon, 1980 #153;Koga, 1984 #26709;Xiao, 1991 #26707}} Molecules in the first coordination shell of each of the four crystallographically unique molecules (13 total molecules) were extracted from this low temperature crystal structure and rotated such that the [110] direction was brought into coincidence with the other 11 members of the $\langle 110 \rangle$ family, resulting in 4 × 12 = 48 total first shell molecular arrangements. These first shells were randomly distributed onto all FCC lattice sites in the supercell, with new molecules overwriting existing molecules when a collision occurred. See **src/basisSets/FirstShell.java** for the source code which extracts the first shell monoclinic constructs from the monoclinic crystal structure and prints to a file called "sorted_eight.lattice".

2nd shell Monoclinic. The exact same principle as the 1st shell monoclinic ordering motif except the 2nd molecular coordination shell was used for a total of 1 + 12 + 42 = 55 total molecules.

3rd shell Monoclinic "Star". The monoclinic "star" ordering motifs were constructed by starting from the four crystallographically unique molecules in the monoclinic unit cell and finding the neighboring molecules whose centers of mass were well-aligned with the intermolecular $\langle 110 \rangle$ directions. The black molecules in the illustration given in Figure 4a show a slice through the center of a 3rd shell star construct. There are six additional $\langle 110 \rangle$ directions that are above and below the plane shown in Figure 4a that are not shown. The intermolecular $\langle 110 \rangle$ directions are defined by the vectors connecting the central molecule with its first shell molecules. The criteria for determining if higher-shell molecules were "well-aligned" with these intermolecular directions is illustrated in Figure 4b. In this illustration, consider molecule 1 to be the central molecule, molecule 2 to be in the first shell and molecule 3 to be in the second shell. If the angle between the vectors $\overrightarrow{12}$ and $\overrightarrow{23}$ was less than 15°, then the second shell molecule was considered to be well-aligned with the first shell molecule and thus part of the shell structure. Interestingly, this algorithm produced shell structures that were not equivalent in all of the intermolecular $\langle 110 \rangle$ directions but instead were more planar, as illustrated in Figure 5.

Similar to the 1st and 2nd shell monoclinic structural motifs, the four 3rd shell star aggregates were rotated such that the [110] direction was aligned with the remaining 11, producing 4 × 12 = 48 unique molecular aggregates which were then randomly placed on the FCC lattice sites with new molecules overwriting existing molecules when a collision occurred.

See **src/basisSets/Star_110_Lattices.java** for the source code which computes this star pattern to an arbitrary shell distance and prints to a file called "110_stars-[numShells]shells.lattice" where numShells is the coordination shell to compute the star constructs. See output/3 shell star/ for the atoms files and the corresponding rendered .png images.

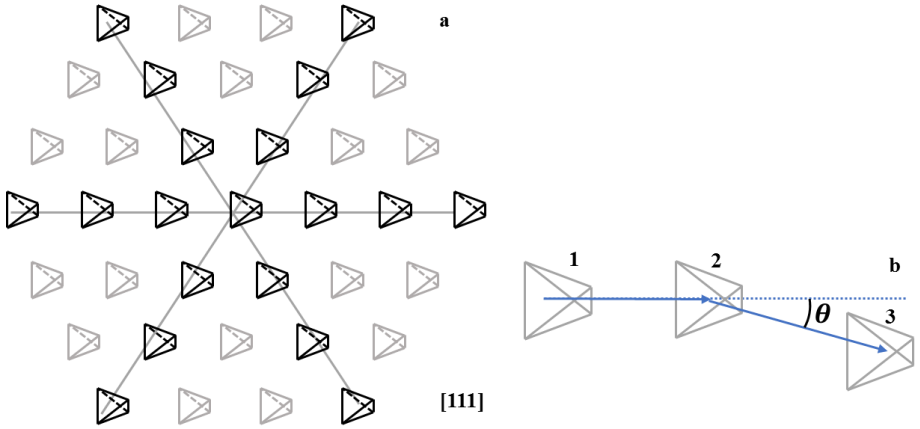

Figure 4. ***a.*** *Illustration of the [111] plane of one of the 3rd shell monoclinic star ordering motif. The remaining six members of the ⟨110⟩ family are out of the plane and not shown.* ***b.*** *Illustration of the criteria to decide if a higher-shell molecule is "sufficiently aligned" with the intermolecular ⟨110⟩ directions as defined by the central molecule and its first shell.*

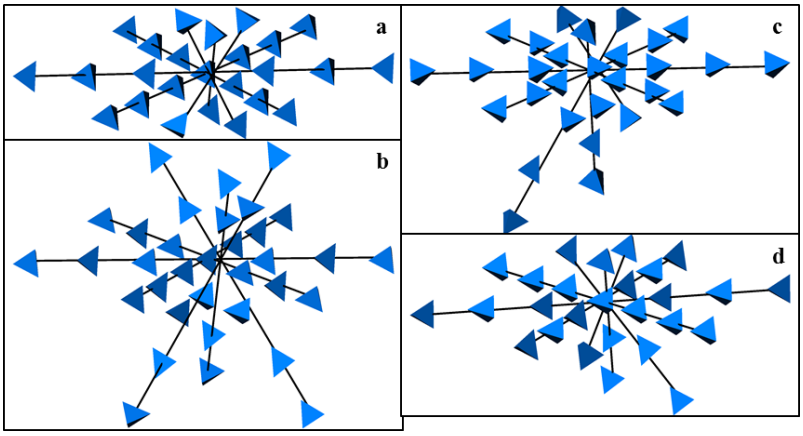

Figure 5. 3rd shell monoclinic star constructs based on the four crystallographically unique molecules.

3.2. Lattice Relaxation

Once the simulation lattice is constructed according to one of the ordering motifs, it is in a high-energy state due to the numerous physically impossible contacts that result from the randomness inherent to the construction mechanism. The potential used was the 2-body Lennard-Jones potential (`src/defaultPackage/LennardJonesPotential.java`), given in Equations 3.1 and 3.2, manipulated to be slightly less computationally intensive than the original analytical form.

$$F(r) = \frac{24\varepsilon\sigma^6}{r^7}\left[\frac{2\sigma^6}{r^6} - 1\right] \qquad 3.1$$

$$U(r) = \frac{4\varepsilon\sigma^6}{r^6}\left[\frac{\sigma^6}{r^6} - 1\right] \qquad 3.2$$

CrystalSim also has the Hooke potential available for use (`src/defaultPackage/HookePotential.java`) and an interface for implementing other isotropic potentials (`src/defaultPackage/Potential.java`). These potentials are pre-calculated to significantly reduce computational time and stored in a HashMap with a distance precision of 0.001 Å (`src/defaultPackage/PotentialLookup.java`).

The plastic crystalline lattices as constructed initially resulted in many impossibly short intermolecular Br-Br contacts. There are many approaches to reducing the energy of a simulated system including Molecular Dynamics (MD) and Monte Carlo (MC). MD simulations allow atoms to move according to the laws of classical mechanics whereby the forces on all atoms are calculated at every time step and atoms and molecules usually retain their velocity from the previous time step. While MD calculations may provide additional insight, the goal of this work is to understand the extent to which the structure of α-CBr_4 can be described from a static perspective. Thus, MD simulations do not provide the desired insight into α-CBr_4. Traditional MC attempts to move one atom at a time by a random amount in a random direction using the energy before and after as criteria for deciding whether the move is to be kept, though this strategy is not commensurate with maintaining rigid tetrahedra. Another MC approach is to randomly distribute the six D_{2d} orientations onto the FCC lattice sites and then randomly change the molecular orientations until no impossibly short distances remain.{{Folmer, 2008 #1}} Using MC to change molecular orientations was shown to fit the most intense structured diffuse scattering but was unable to account for the new structured diffuse features.{Dill, 2014 #27000} Reverse MC (RMC) is similar to traditional MC with the exception that the simulation cell is converged to "fit" some experimental measurement, though the RMC results are frequently poorly interpreted as it can be quite challenging to relate RMC results to fundamental structural principles.

For these reasons, a new energy minimization strategy was desired that focused on enhancing intermolecular orientational correlations while allowing the freedom of movement (rotation + translation) that the MD approach provides which is missing from MC methods. This approach is described as a pseudo-random walk, with its algorithm described in pseudocode in Figure 3.6. The random walk algorithm begins by selecting a molecule (`testMolecule`) from the simulation cell at random. The pairwise interaction energy with the surrounding twelve nearest neighbors along the ⟨220⟩ directions is determined and the molecule with the most unfavorable interaction energy is selected (`worstMolecule`). The pairwise intermolecular forces are calculated and the molecules are moved as a result of these forces. The `testMolecule` was then set to the `worstMolecule` and the algorithm was repeated some number of times, called `numWalkSteps`. After each molecule in the simulation cell had been the starting point for a random walk, one "walk" was complete. The lattice which resulted from this approach is dependent upon the number of walk cycles (`numWalkCycles`), the number of steps per random walk cycle (`numWalkSteps`) and the interaction energies selected. If too many walk cycles and walk steps were set, the resultant lattices did not produce diffraction patterns that closely resembled the experimental images, as shown in some images presented in §4.3.2. The ideal number of walk cycles was generally between 1 and 5 and the ideal number of walk steps was generally between 256 and 512, depending upon ε, the depth of the LJ well as a deeper well will converge faster than a slower one. Additionally, because this algorithm must resolve physically unreasonable intermolecular contacts, the force calculation often results in extremely large forces. To ensure that the simulation cell did not "explode", the maximum translational and rotational motions for any given movement attempt were set to 0.25 Å and 1°, respectively.

See `src/defaultPackage/Simulate.java` for for all methods related to the energy minimization strategies.

```
FOR walkIdx = 0 to numWalkCycles
     FOR moleculeIdx = 0 to numberOfMoleculesInLattice
          SET testMolecule equal to the molecule with moleculeIdx
          FOR testIndex = 0 to numWalkSteps
                    SET worstMolecule equal to the molecule interacting
                              the least favorably with the testMolecule
                    CALL reduceEnergy of worstMolecule and testMolecule
                    SET testMolecule equal to the previous worstMolecule
          END FOR
     END FOR
END FOR
```

Figure 3.6. Pseudocode random walk algorithm.

Once the lattice has been sufficiently relaxed, as determined by the criteria that the user sets in the primary control file, , the lattice is output as a .xyz file. The .xyz file is structured such that the first line contains a single integer representing the number of atoms contained in the file followed by a line for comments which is ignored by most .xyz file readers. The remainder of the file consists of rows representing a single atom as four numbers: Z, x, y, z, where Z is the atomic number and (x, y, z) are the coordinates of that atom. Note that Z, x, y, z must be tab-delimited to adhere to the .xyz file standard. Also note that Z can be the string (C, Br, etc) or the atomic number (6, 35, etc.). The .xyz files output by PlasticSim gives atoms in terms of the crystal coordinates and not Cartesian coordinates. The crystal coordinates, in the case of a cubic crystal are the Cartesian coordinates divided by the edge length of the unit cell edge.

3.3. Calculating the diffraction patterns

A 2D diffraction pattern calculated from a single simulated lattice has a very high signal-to-noise ratio. However, because no two simulated lattices are identical due to the randomness inherent to the lattice construction and relaxation, 2D diffraction patterns can be calculated for multiple simulated lattices with the same input parameters and then summed together to increase the signal-to-noise ratio, as shown in Figure 7 for a series of D_{2d} simulations with 10×10×10 supercells after a 1×256 random walk.

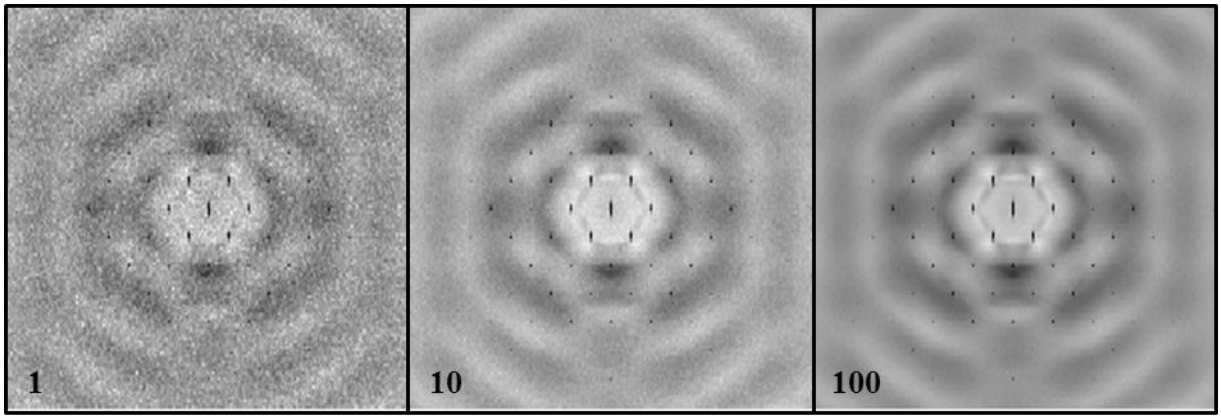

Figure 7. Visually demonstrating the increase in signal-to-noise ratio as the diffraction patterns from an increasing number of simulations are summed together.

CrystalSim can compute diffraction patterns along any reciprocal lattice direction with the constraint that three orthogonal vectors must be given (see `runDiffractionCalc(…,…)` in `src/simulationTypes/RunSimulationThread.java`). The [001], [011], [111], [211], [210], [010] and [100] directions are built in.

CrystalSim computes diffraction patterns by determining the number of pixels to be calculated based on the size of the .xyz input lattices, the desired ΔQ and the maximum Q to calculate, computing the Q vectors in 3-space for those pixels and pre-calculating the scattering factors for each element in the input lattices for each pixel at the given wavelength. The scattering factors are pre-computed for each of the pixels and each element in the input lattices and then CUDA-capable GPUs are leveraged to compute the scattering equation, as given in 3.3. (See `c&cuda/cuDiffraction.cu` and associated files for the CUDA implementation)

$$F(\vec{q}) = \sum_{a=1}^{all\ atoms} f(\vec{q}, Z_a) \times \exp(2\pi i \times \vec{q} \cdot \vec{r_a}) . \qquad 3.3$$

The elemental scattering factors, $f(\vec{q}, Z_a)$, are a combination of the scattering power, $f^0(\vec{q}, Z_a)$, and anomalous dispersion effects, $f'(\vec{q}, Z_a) + i \cdot f''(\vec{q}, Z_a)$, which are large near elemental absorptions. Numerous models have been proposed to describe f^0 but the most popular was computed with Hartree-Fock wavefunctions and fit to the nine parameter expression in Equation 3.4 where the nine parameters are called the Cromer-Mann coefficients.{{Cromer, 1968 #26325}}

$$f^0(\sin\theta/\lambda) = \sum_{i=1}^{4} a_i \exp[-b_i(\sin\theta/\lambda)^2] + c. \qquad 3.4$$

The wavelength dependence of the anomalous dispersion effects are available for incident X-ray energies between 50 eV (λ=24.8 Å) and 30 keV (λ=0.4 Å) for most elements.{{Henke, 1993 #26580;Gullikson, 2010 #23938}} The experimental images presented in this chapter were collected at λ=0.13702 Å (90keV), significantly higher than the measured energies. The wavelength dependence of the anomalous dispersion effects for both elements at 90 keV are likely extremely similar to the reported values at 30 keV because neither element has an absorption edge near either energy and anomalous dispersion effects only provide a meaningful contribution to the scattering factor near

absorption edges. Therefore, for the diffraction pattern calculations, the contribution from anomalous dispersion effects, $f' + i \cdot f''$, were calculated for λ=0.4 Å as opposed to the experimental value of λ=0.13702 Å.

3.4. Visualizing the diffraction patterns

As PlasticSim is set up to automate much of the simulation and diffraction calculation process, tens to hundreds of diffraction patterns may be calculated for a single set of simulation parameters. As shown in Figure 7, the sum of those diffraction patterns provides significantly more information than a single diffraction pattern. To sum those diffraction patterns, have a look at the main method of src/input_output/SumXrayFilesInFolder.java. This java file allows you to simply provide a path to the calculated diffraction patterns and it will sum them into a single file. This file can then be loaded in to Ramdog (http://www.github.com/ericdill/Ramdog). Ramdog will automatically parse the header information which contains the diffraction axes and the ΔQ so that the coordinate info panel (in the top right of the Ramdog UI) will provide accurate Q-coordinate information as the mouse is moved across the image.

4. References