Visualizing Cubic Crystals in VPython

Connor Bailey
csbaile2@illinois.edu
Math 198, University of Illinois, Fall 2014

December 11, 2014

Abstract

One of the hardest aspects when first learning about crystallography is visualizing 3D crystals from 2D pictures. This program focuses on helping the user visualize cubic crystals in 3D using VPython. It allows the user to display planes and directions in a unit cell and control the crystal system of the unit cell (simple cubic, body-centered cubic, face-centered cubic, or blank). Using sliders, the program dynamically changes the scene while displaying the Miller Indices of the current direction and plane displayed. The scene displaying the unit cell can be rotated freely, allowing for easy understanding behind the structure.

1 Background

There are three basic cubic crystal structures- simple cubic (sc), body-centered cubic (bcc), and face-centered cubic (fcc). These are common structures for elemental metals to have at standard conditions and are the first structures a student of crystallography learns. It is also difficult to visualize these structures in 3D, which is where my program comes in.

The other lesson in crystallography needed to fully understand my program is the idea of *Miller indices*. This is the standard way of denoting planes and directions within a crystal. A Miller index is denoted as (hkl) (for planes) or [hkl] (for directions). It is customary to not put in any commas between indices and for negative numbers to be denoted with a bar instead of a negative sign. The meaning of the indices, however, are different between planes and directions.

For directions, each index is the ratio of the coordinate for the head of a vector with tail at the origin. Each Miller index is divided by the maximum index to get the coordinates for the head of the vector. For example, [123] denotes a vector starting at the origin and going 1/3 units in the x direction, 2/3 in the y, and 1 in the z (normally, a unit is the lattice parameter,a- the side length of the unit cell). Likewise, [$\bar{1}$12] goes -1/2 in the x, 1/2 in the y, and 1 in the z. To keep a direction displayed in a unit cell (like in my program), the origin has to be changed when there are negative Miller indices. With the last example, the origin would have to be moved from (0,0,0) to (1,0,0), because there is a negative change in the x direction.

For planes, each index is the reciprocal of the intercept of the plane with each respective axis. So (122) is the plane that intersects the x axis of the unit cell at 1, the y axis at 1/2, and the z axis at 1/2. Values of h, k, or l that are 0 mean that they never intercept that axis. (100) would not intercept the y or z axis- it is normal to the x axis.

From this last example, it might be obvious why Miller indices are useful- *for a given Miller index, the direction [hkl] is normal to the plane (hkl).* This relationship is easy to see in my program when the direction and plane indices are set equal to each other. Miller Indices give crystallographers an easy way to denote directions and planes (all integer values, very simple) for the most common situations that arise.

2 Program

With any necessary background discussion complete, my program can now be covered. The program allows the user to visualize planes and directions in a cubic crystal of their choosing, where Miller indices can be manipulated using sliders. It is split up into several parts.

2.1 Set-up

When first initialized, the program imports **visual** (allowing any VPython functions/objects to be used) as well as **wx** (which allows for sliders, buttons, etc. to be used for user interaction). It also sets values to the atomic radii and color (currently 0.2 and red, specifically, but could easily be changed in the future), which are used by the various crystal system functions when creating the unit cell.

The **setup** function is used to create the initial, blank cube where the unit cell will be. It creates 6 semi-transparent walls, which are VPython *box* objects, to form a cube. It also creates 3 axes (*curve* objects), and labels them as x, y, and z using *label* objects.

Two windows are created at initialization- the *scene* (main) window that displays the unit cell, and the “w” window, which is home to all the buttons, sliders, etc. that allow for the user to manipulate the scene. The specific buttons and sliders will be discussed with the appropriate objects that they act upon.

2.2 Crystal System

Four different unit cells are available in the program- sc, bcc, fcc, and none. None displays simply the blank cube created with **setup**, which can make it easier to see planes with high valued Miller indices. The crystal systems are each made using an appropriate function- **scunit**, **bccunit**, and **fccunit**. Each works by looping through 8 positions (corners of the cube) and creating a sphere object with size and color specified earlier in the program. The functions for bcc and fcc then add the remaining inner atoms. In this case, the cube is of side length 1.

Each unit cell is created over the cube made by **setup**. The user can control which crystal system is displayed using the radiobox (part of the **wx** library) called c1 in the w window, which allows one of the four options to be chosen. Each time a change is made to the radiobox, it calls the **refresh** function to update the scene.

2.3 Plane

The plane displayed within the scene is controlled by three sliders- one for each Miller index (h, k, and l). Each slider has a minimum of -7 and a maximum of 7- these choices are arbitrary, but were made to keep the plane relatively large in the scene and because Miller indices used in real life are rarely above 5. Note that negative signs are used instead of the standard bar notation because of the limitations in VPython, and to simplify the program.

To actually create the plane, the **planes** function is used. This takes three arguments- h, k, and l, and outputs a plane. In this case, the *face* VPython attribute is used to create a plane from 3 or 4 points (depending on the number of Miller indices that are zero). The **planes** function actually calls one of three different functions, depending on the number of Miller indices with value zero- **planes_nozero**, **planes_onezero**, or **planes_twozero**.

Each function operates slightly different. For example, when there are no zero-valued Miller indices, the plane displayed is a triangle, with one point on each axis. However, when at least one index is zero, the plane displayed is a rectangle (or square in the case where there are two zero). The **planes** function checks how many are zero and calls the appropriate function. Each function has to check which indices are negative and change the origin appropriately (as discussed earlier). Using this method, any plane can be displayed within a single unit cell. Each planes function operates the same when it actually has to make the plane- it uses VPython's *faces* to create a triangle (or two triangles for a rectangle) for the plane. It then makes the faces *two sided* and *normal* to make them visible from both sides, and makes the material *emissive* to make the planes bright enough to be viewed without further manipulation. When deciding where to put the origin and how to make the planes, the functions currently use a brute-force method with many if statements- this could be improved in future versions.

Each time a slider for any plane index is changed, it calls the **refresh** function. Note that when all three indices are zero, no plane is displayed (because there is no plane for a Miller index of (000)).

There is also a "Plane Reset" button within the window, w. It is a **wx** object that, when pushed, sets the h, k, and l values for the plane sliders all to zero, then calls the **refresh** function. This will result in no plane being displayed.

2.4 Direction

Displaying a direction in the unit cell is much easier than a plane, because it uses the same method regardless of any zero-valued indices. Like with the plane, there are three sliders that control the h, k, and l values for the displayed direction which also go from -7 to 7. Any time a slider value changes, it calls the **refresh** function, which in turn calls the **direc** function that draws the direction. As long as the Miller index is not [000], it draws the direction. It finds the maximum index, divides each index by this common denominator, and from there has a set of two points to make a direction. It has the origin, which is (0,0,0) (and adds 1 to any coordinate with a corresponding negative Miller index) and the point (h,k,l) divided by the maximum.

The **direc** function then uses the VPython *arrow* object to create an arrow (direction vector) from the calculated origin to the calculated point. The arrow object requires three parameters- starting location (tail), the axis it points along, and its length. The starting position is the calculated origin, the axis is the vector defined by the difference in the two points mentioned above, and the Pythagorean theorem calculates the length from the two points.

Like for planes, there is a “Direction Reset” button that sets all Miller index values for the direction equal to zero, then refreshes. Because they are all zero, no direction will be displayed.

2.5 Display

As mentioned previously, this program consists of two windows- the scene, where the unit cell is displayed, and the window, w, where all controls are housed. The window contains 10 components: the 6 sliders to control plane and direction, the radiobox that controls what type of unit cell is displayed, the two reset buttons for plane and direction, and finally the text box that displays the current Miller indices for both the displayed plane and direction. These are all objects from the **wx** library, which is included with VPython. Each time any of these objects' values is changed (sliding, clicking a button, etc.- an *event*), the **refresh** function is called.

refresh is what allows the program to dynamically change the scene in the program whenever a display parameter is changed. It is triggered by an event, and performs the following actions:

1. Clears the scene by making all objects in the scene invisible (using a loop).
2. Calls the **setup** function to create the cube and axes basis.
3. Obtains the value of the crystal radio box and calls the appropriate function to display the appropriate crystal system.
4. Obtains the values of the plane sliders using .GetValue, and calling the **planes** function on those values to draw a plane.
5. Obtains the values of the direction sliders using .GetValue, and calling the **direc** function on those values to draw a direction.
6. Uses the values of the plane and direction sliders to change the message in the text box to display the appropriate current Miller indices.

This is what allows the program to update in real time with any changes the user makes.

3 Conclusion and Future Updates

Making this program has been very enjoyable, with many obstacles to overcome. I am pleased with the final product and its ability to make crystal visualization simpler to see and understand. However, there are many more features that could be added to this program. For example, allowing the user to change the color and size of the atoms displayed would be relatively straight forward. Expanding the crystal options beyond the simple cubic systems is the next natural step in this RTICA- tetragonal, monoclinic, triclinic, or hexagonal close packed (HCP) are a few good examples. Future projects could build off this one to cover all 14 Bravais Lattices.