Pyxtal Reference

1 User Manual

This section provides basic information for using pyxtal. (Not modifying it; just using it.)

1.1 Getting Started

To run the pyxtal program, type "python3 pyxtal.py". It should open a window allowing you to pick files, select features, and hit "Go" to start processing files.

The Github repository includes three test images:

- double.tif: an *image* originally from an AFM.
- hex1short.gsd: a gsd file of individual *particles* forming hexagonal lattices.
- test_diblock1.gsd: A gsd file in which there are spherical *assemblies* of type 'B' particles.

To process any of these test files, hit Clear Files (to erase current selection) and "Add Files" to select the filename you want. You will then need to manually select the appropriate input file type, below. Then just hit the "Go" button, and you should at least see something. Go ahead and play around! :-)

1.2 Selecting Images

Select images to process using the Add Files and Clear Files buttons. The initial path is set to the directory from which pyxtal was executed. Clicking Add Files opens a file dialog box, allowing you to choose a directory and a file or files within it. Selecting files within a directory automatically changes the path as well. Pyxtal currently supports reading files from a single directory at a time. To start over, click the Clear Files button.

1.3 Input File Options

Pyxtal only supports processing a single type of file at a time. The type of file is selected by the user:

- **Image file:** reads a gray scale image (tested with .tif files, but should work with other types). Available user options: Dark Spheres, Sphere Size (must enter a number, in pixels), and Periodic Boundaries (not yet implemented).
- **gsd particles:** Reads a gsd trajectory file, and analyzes the positions of all individual particles. Available user options: Frames ("snapshots") to process, and Periodic Boundaries (not yet implemented).
- **gsd assemblies:** Reads a gsd trajectory file, and looks for assemblies of a particular type (typically 'A' or 'B') of particle. Looks for groups of particles of approximate size given by the Sphere Size option. Other user options: Frames to process.

1.4 Output File Options

This section of the user interface controls what image files are written as output. This part is only partially implemented so far.

It's not implemented yet, but the output image size will be specified by the user. The user gives a single number corresponding to the width of the output image(s) in pixels. If a negative number is entered, the output image size will be that multiple of the input image size. (That is, an 800 x 600 input image, with output image of -2 will give a 1600 x 1200 output image.)

Eventually it may also write some kind of log file (maybe a concatenation of info from the stats window?). And hopefully it will make mpeg movies too.

1.5 Analysis

The analysis frame of the user interface controls what analysis is done on the images. Presumably this section will expand as necessary to include whatever additional analysis is desired. Not yet implemented; currently no analysis is done, and the application only shows pretty pictures.

1.6 Window Control

This section of the user interface controls the behavior of the user interface with respect to multiple windows. Hit the "Go" button to start processing files; each file opens up it's own viewer window.

- Retain Windows: whether to retain or destroy each image window after the files are processed.
- Lock views: When this option is selected, the "views" that have been selected in one viewer window (which image and annotations to display) are transferred to other windows when they gain focus; useful for comparing two images.
- Lock zoom: the magnification and translation of an image in the active window are transferred to other windows when they gain focus. Again, useful for comparing two images.

1.7 Using the Viewer Window

When a viewer window opens, displaying a file, various images and annotations can be selected. These are mostly self-explanatory; go ahead and mess with them to see. Show stats displays basic information on spheres, bonds, and crystal defects. Trajectories will display on each image a particle's path over all of the frames, but is not yet implemented.

- Zoom the image using the mouse wheel.
- Move the image by clicking and dragging it.
- Double click on the image to recenter it in its original position at its original magnification.
- Use right and left arrow keys to flip through multiple viewer windows in order.

2 Programming Reference

This section is designed to be a guide to the structure of the Pyxtal program. Anyone attempting to make changes to the program should read this part first.

Pyxtal is coded in Python, and has only been tested in version 3.6.6. It also uses various bits of the libraries numpy, scipy, matplotlib, tkinter, trackpy, and gsd, in addition to the standard python libraries; presumably these will need to be installed in order for pyxtal to work.

2.1 Images, plots, and coordinates

Although Pyxtal was designed with the idea of using input data from either image files or gsd trajectory files, the program inherited some of its design philosophy from its predecessor ("collidl", written in IDL), which was designed to handle input only from microscope images.

When reading input from image files, the image itself provides the natural coordinate system, ranging from 0,0 to the x and y size of the image in pixels. Spheres are guaranteed to be at least several pixels wide. Pyxtal uses matplotlib to display the original image in the viewer (using imshow()), and adds various other scatter, line, and circle plots for annotations. The orientation field is also an image, created using the same pixel range as the original image.

Pyxtal maintains much of this philosophy even when analyzing gsd images. Although hoomd typically (always?) puts the center of its simulation box at the origin, Pyxtal performs a shift to put the lower left corner of the box at the origin instead. And while the default particle size in hoomd is a diameter of 1 (which tends to roughly equal the lattice constant too) , Pyxtal multiplies this and all other dimensions by an arbitrary scale factor (currently hard-coded at 10). This means that the size of the image created for the orientation field is 10 times the original hoomd box size.

For analyzing assemblies of hoomd particles from gsd files (for block copolymer spherical domains), using the original hoomd box size seems to make sense again.

2.2 PAGE

The user interfaces of Pyxtal were all developed using PAGE (Python Automatic GUI Generator, Version 4.17, available at http://page.sourceforge.net/.) Although this greatly simplified the process of creating and organizing the GUI layout, it also led to an overall organization of the code that is less than optimal in some ways. There are two fundamental issues. The first issue is that for each window, PAGE creates two files, one file for actually setting up the user interface (e.g. pyxtalviewer.py) and a second file that contains all of the supporting code (e.g. pyxtalviewer_support.py). The idea is that if small changes are made in PAGE to the look of the user interface, it should generate new code for the GUI, but leave the supporting code alone. Unfortunately, the separation between these two is done in a way that results in somewhat untidy code, in my opinion —though perhaps there's no better solution to the problem than what PAGE does. For example, all of the GUI objects and tkinter variables used within them are declared as global variables, which is clumsy at best, and actually fails in the case of two or more instances of the same GUI class, as with multiple viewer objects for different image files. In addition, this means that many of the attributes of the viewer that are created during image processing end up being declared outside of the class.

The second issue with PAGE is that it is designed to create an application with only a single window. In the case of Pyxtal, which has both a main control window and a window for each file or frame being viewed, Pyxtal created two separate groups of files, which then had to be stitched together by hand—again, more clumsily than if the whole thing had been hand coded from the start.

Given the structure imposed by PAGE, I had essentially two choices in writing Pyxtal. Either, I could try to preserve the structure of the files automatically generated by PAGE as much as possible (at least the GUI code) to facilitate easy regeneration of the GUI code, or I could change the entire structure to make it neater and better. In fact, I chose something in the middle. Here's the upshot of what I did:

- The supporting code should NOT be regenerated in PAGE, ever. It has changed too much from PAGE'S original version for the new code to be of any use.

- If the GUI code is regenerated, then various changes I made to it will have to be put back in, by hand. (Presumably, one should use diff or something to compare the new code to the old code, and select bits accordingly.

2.3 Brief description of Pyxtal Main Window (pmw) object

The pmw object is the tkinter window automatically created on running the application. Some of the object's attributes are set at creation, others are added after creation. Most of the attributes of object are self explanatory based on their names, but a few quirks deserve a little explanation here.

- **pmw.sphereSize, pmw.sphereSizeStr:** The attribute sphereSizeStr is a tinker string variable controlled directly by the associated user text entry box. The user must enter a valid integer; if not, the previous value is retained. That previous value is held in the attribute pmw.sphereSize[0], which is a list with a single integer element. (I did it as a list, so that I can pass the attribute to a function and have it be changed, using the same function for several different pairs of string/integer values. All of the text entry boxes that require an integer are handled this way. For image inputs, sphereSize is specified by the user. For gsd file inputs, the sphereSize will eventually be read from the gsd file, but is currently hardcoded to 1.
- **pmw.filelist** is a list of filenames to be processed. **pmw.viewers** is a list of the viewer objects generated for each file when the Go button is hit.
- **pmw.top:** the tkinter root window.

2.4 Brief description of Pyxtal Viewer object

For each file processed, a pyxtal viewer object is created. It is a tkinter toplevel window with lots of other widgets, and also includes as attributes many variables and bits of data that are created along the way.

Below is a complete (I think) list of attributes, where I've described the ones that warrant some explanation:

- **annotationsframe**:

- **ax:** The axes of the plot, created by pyplot.
- **canvWidget**:
- **circlesCheck**:
- **corners**: A 2D ndarray of the current axes limits: [[xmin, ymin], [xmax, ymax]].
- **defectsCheck**:
- **fig**: the figure containing all the plots, created by pyplot.
- **filename**: the string of the base filename (no path) being processed.
- **filteredButton:**
- **idx**: the index (0 to number of files) of the file being processed.
- **image**: the raw image as read from the file.
- **imageframe**: the frame that holds the controls for the image (raw, filtered, etc.)
- **imgCanvas**: the tkinter canvas that holds all the matplotlib plots.
- **imgshape**: the size [x, y] of the image.
- **inv_image**: the inverted image (black on white instead of white on black).
- **invertCheck**:
- **invertImage:**
- **locations**: a 2D ndarray of the x,y coordinates of all the particles (or spheres).
- **mousebuttondown**: whether the left mouse button is already currently down.
- **noneButton**:
- **orientationCheck**:
- **plt_angleimg:** plot of the orientation image.
- **plt_circles:** plot of circles drawn around all particles.
- **plt_disc:** plot of all disclinations.
- **plt_rawimg:** plot of the raw image, or circles of the particles if gsd particles.
- **plt_triang:** plot of blue lines for the triangulation
- **pmw:** the pyxtal main window object that created the viewer. Used to refer back to global attributes like pmw.sphereSize, etc.
- **prev_button_time:** The time at which the mouse button was last pressed, used to detect double clicks.
- **rawButton**:
- **rgbimg**: the image data (rgb) for the orientation field.
- **ShowCircles**:
- **showDefects**:
- **showOrientation**:
- **showStats**: Display a separate window with basic counts of particles, bonds, and defects.
- **showTraject**: Show the path of a particle over many frames.
- **showTriang**:
- **statsCheck**:
- **top**: the toplevel tkinter object of the viewer, used to set the title and destroy the window.
- **trajectCheck**:
- **tri**: A whole tree of stuff related to the triangulation, mostly created by the scipy.delaunay function, but with some new attributes added by me. I'll list only the stuff I added here, below.

- **tri.bondsangle:** the angle (0 to pi/3) of each bond.
- **tri.bondsl:** the length of each bond.
- **tri.bondsx:** x position of each bond.
- **tri.bondsy:** y position of each bond.
- **tri.cnum:** the coordination number (# of neighbors) of each vertex.
- **tri.disc:** the index of each vertex that has cnum not equal to six.
- **tri.outer_vertices:** the index of each vertex that is on the outside of the triangulation.
- **tri.points:** the original input points, identical to viewer.locations.
- **triangulationCheck**:
- **whichImage**:
- **zoom**: the ratio by which the current image is zoomed.

2.5 Description of sphere detection algorithms

For input files that are images, I found the locations using the trackpy.locate function of the trackpy package, written and maintained by the Trackpy soft matter group.

For detecting spheres of a particular type of particle in gsd files, I first created a density map of the number of particles of a particular type within each xy area in the gsd file. That density map is effectively just an image, and I used the trackpy.locate function to find spheres in it.

2.6 Description of dislocation algorithm

Two neighboring disclinations of opposite signs (say, one with 5 neighbors and one with 7 neighbors) form a dislocation. Pyxtal detects these neighboring pairs and denotes them with a connecting yellow bar between them. Disclinations that are not paired up this way are denoted with an additional red or green circle around them.

The simplest way to do this pairing would be to simply look for a “5” next to a “7”, and pair them together. But it gets a little more complicated when there’s a line of 5’s and 7’s, like this:

5 7 5 7

If you happen to start by pairing the middle two together (“5 7–5 7”), then you end up with two “unpaired” disclinations on the ends. A more sensible grouping would be “5–7 5–7”. Pyxtal accomplishes this using a recursive algorithm. It starts by simply trying to pair each disclination with a neighbor, and it may indeed start by pairing this group as “5 7–5 7”. But when it tries to find a “mate” for the 5 on the end, it has to look a little harder:

- First, it looks for a neighboring unpaired 7 (or greater).
- If unsuccessful, it tries to pair with the 7 that is already paired to the other 5. It can only do this if the 7 can break way from the other 5… and it can only do that if the other 5 can find another mate as well. In this case, it can be done: the other 5 pairs with the 7 on the end, and everybody ends up happy.

Things get even more complicated when you consider vertices on the edge, which often appear as disclinations since their real neighbors were clipped by the edges of the image. A disclination that is neighbors an “edge” disclination can always be paired with it, though Pyxtal will first attempt to pair it using only non-edge disclinations.