PS-Poly Python Package

Version 0.2

Instruction Manual

Package Description

The PS-Poly algorithm is a particle detection program designed to separate individual features based on shape using images taken using atomic force microscopy. The program sorts particles into the following groups: linear, looped, branched without looping, branched with looping, overlapped, and noise. For linear molecules, persistence length is calculated using the worm-like chain model. Due to interpolation that is performed to increase the pixel density of the image, the persistence length result is achieved with subpixel precision.

Code is open source and available in both Igor Pro (WaveMetrics, Inc.) and Python. We present here an introduction to the Python package, including instructions for basic use and an introduction to more advanced methods. The Python package, when used as standalone software, is primarily operated through the terminal. As such, we assume a basic familiarity with terminal commands in your chosen operating system, but no Python experience is required.

We hope you find success using our provided software! Please do not hesitate to contact us with questions, suggestions, or bugs.

Simple Data Analysis

To begin using the PS-Poly Python package, you will need an AFM image, along with its scale. This package accepts most image types (.png, .jpg, etc.), as well as NumPy binary files (.npy). Image scales should be given as the width of a single pixel in nanometers, where pixels are assumed to be square.

If you wish to use Jupyter Notebook, open the provided examples.ipynb file. Otherwise, open a terminal window or similar environment, navigate to the location of the provided AFM data, and start Python. Import the PS-Poly Python package with `IMPORT PSPOLY`. To process and present the data in one step, use `PSPOLY.RUN`.

```
Python 3.8.19 (default, Mar 20 2024, 19:55:45) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> import pspoly
>>> pspoly.run('newCL_0000.png',2)
```

In the above example, newCL_0000.png is AFM data with a pixel scale of 2 nm. The file name, as with any text string in Python, must be enclosed within single or double quotes. The program should finish in about a minute. The output will include statistics for each of several feature types and the calculated persistence length, arranged in a table format.

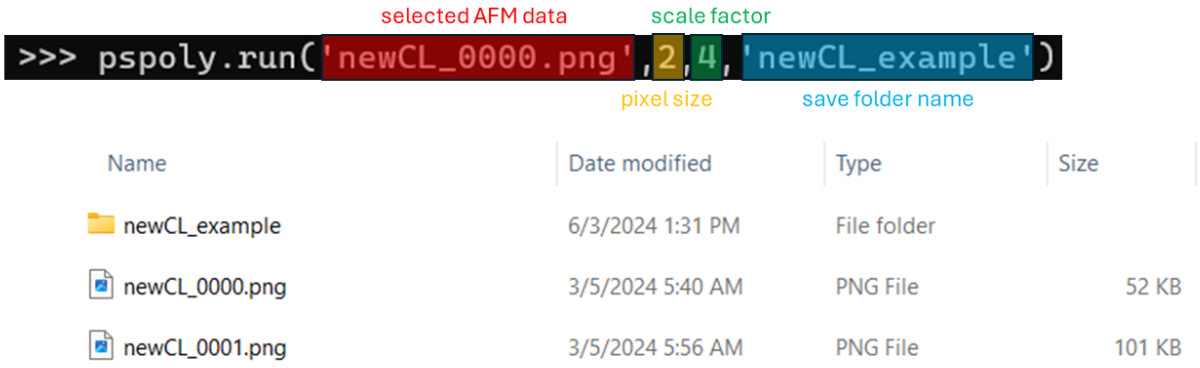
Implementing subpixel interpolation is as simple as adding a scale factor to the run command. This argument indicates the factor by which the program will scale up the original image. The entered pixel size need not be adjusted.

```
>>> pspoly.run('newCL_0000.png',2,4)
Particle Type          Number of Features          Average length (nm)          Percentage of Total Polymerization Length
Linear                 294                         13.6                         38.9%
Looped                 1                           52.7                         0.5%
Branched (no looping)  108                         46.4                         48.9%
Branched (with looping) 7                          115.1                        7.9%
Overlapped             4                           96.5                         3.8%

Number of High Points: 19
Number of Noise Particles: 0

Persistence Length (nm): 13.6
```

A scale factor of four is used here, with the same AFM data and pixel scale as before. Different scale factors may produce slightly different persistence lengths, so we recommend using one factor for all your data analysis. Note that larger scale factors can give better results, but the algorithm will slow down as the size of the image increases.

In order to calculate the above statistics, PS-Poly runs a number of filters on the input data, including thresholding, skeletonization, boundary pixel removal, and several others. It would be computationally intensive to evaluate these filters every time they are needed, so the program generates a temporary folder for the intermediate data: pspoly_temp. The folder is removed from the system's directory at the end of the algorithm, so it is important that the folder not be used by any external scripts unless you are very careful.

If you intend to reuse the intermediate data, such as by doing further analysis or referring to metadata, a folder other than pspoly_temp should be used. To do this, include the name of the new folder as a fourth argument to PSPOLY.RUN. The final command should be of the form PSPOLY.RUN(IMG,PX_SIZE,SCALE_FACTOR,NAME). Just like IMG, the NAME argument should be enclosed in quotes. Once the command has run, statistics will display as usual, and there will be a new folder in the opened directory containing all intermediate data.



sample file structure with AFM data and new save folder

There is a fifth, rarely used argument to PSPOLY.RUN called DISPLAY, which defaults to TRUE. If the argument is set to FALSE, the output statistics will be returned as a tuple of strings instead of being printed. This is mostly useful for developing your own display format.

Data Management Methods

Once we have saved an intermediate data folder, we need a way to interface with the folder inside our Python environment. This is accomplished through the PSPOLY.POLYDAT object.

A polydat object may be created from a preexisting intermediate data folder using the PSPOLY.LOAD method. A typical intermediate data folder contains info.txt, data.npy, and an additional folder for every filter applied to the data.npy file. The object created by PSPOLY.LOAD will contain the metadata from info.txt and an array representing the AFM data / filtered AFM data stored in data.npy. Loaded polydat objects' data are accessible via several getter methods.

The POLYDAT.GET_IMG method is an important getter. It is responsible for returning a copy of the stored array containing the AFM image. The returned object is a NumPy array. There are several libraries available for saving NumPy arrays to disk, but this is probably not necessary here because the returned array is already stored in the saved data folder.

If you would prefer to get only a subsection of the AFM image corresponding to an individual particle, POLYDAT.GET_PARTICLE is the best choice. This method takes three arguments: PARTICLE, IMTYPE, and PAD. The PARTICLE argument is an integer specifying which particle will be returned, IMTYPE is a string indicating the type of data to be returned ("default", "mask", or "skeleton"), and PAD is an integer giving the amount of padding around the edges of the image. The figure below demonstrates how to retrieve data using this method, how to display it using the external library Matplotlib, and how to abridge the code using the PSPOLY.SHOW command.

```
>>> particle_16 = newCL_example.get_particle(16,imtype='default',pad=5)
>>> import matplotlib.pyplot as plt
>>> plt.imshow(particle_16)
<matplotlib.image.AxesImage object at 0x000001D1AA533110>
>>> plt.show()
>>> # The above code is equivalent to the code below
>>> pspoly.show(newCL_example,16,imtype='default',pad=5)
```

The POLYDAT.GET_PARTICLE method returns an image of a particle within the context of its surroundings. It is preferable to use POLYDAT.ISOLATE_SKELETON when skeletal particle data is needed for shape classification or analysis. This method returns a polydat object containing only the selected skeleton and only takes the PARTICLE argument.

It is important to know that a polydat object can be created via the PSPOLY.POLYDAT constructor, rather than by loading an existing data folder. This is done with much the same syntax as the PSPOLY.RUN command, except that the name argument comes first. A typical command structure would look something like OBJECT = PSPOLY.POLYDAT(NAME,IMG,PX_SIZE,SCALE_FACTOR). An object created in this way will not automatically be analyzed, so you will have to do some work before you can analyze it. Such methods are detailed in the next section, along with two additional arguments to the constructor: ROOT and SAVE.

Before continuing with advanced examples, it will be advantageous to specify a new save directory for PS-Poly to use. This will ensure that we do not duplicate any files, and it will also allow us to save all PS-Poly objects to the same location, regardless of the folder from which Python was launched. The command to use is PSPOLY.INITIALIZE(CHOSEN_DIRECTORY), where CHOSEN_DIRECTORY is some location on your machine that you can easily access later. Remember to surround the path with quotation marks, as it is a string. You may also elect to pass a second argument to the initialization method, which will determine what happens if a data folder is duplicated. The choices are "ask", "overwrite", and "load". The default is "load".

Advanced Techniques

Some analysis methods beyond the run command are helpful to know. First, we will discuss how to implement the equivalent of PSPOLY.RUN using other methods in the package. Afterwards, we will discuss customizing the filtering process for AFM data, and we will conclude with brief explanations of several other methods included in the package.

Whenever the run command is utilized, it creates a new polydat object with the same arguments as were provided. If no name is given, the name is taken to be “pspoly_temp”, as explained in a previous section. The SAVE argument is set to TRUE, indicating that a folder will be saved to disk containing the contents of the object.

The run method creates annotated skeleton data from the AFM data. This is done via the PSPOLY.PREPARE method, which returns a new, filtered polydat object nested within the original. Annotated, in this context, means that the skeleton data folder contains subdata corresponding to each individual particle of the image, and the important points of each particle are identified. Using PSPOLY.PREPARE creates a nested object with POLYDAT.SUBDAT – a masked call to the polydat constructor where ROOT is set to the name of the object in which the object is nested. It is usually not necessary to refer to the root of an object directly because the various methods in the polydat class take care of nesting automatically.

The last step in PSPOLY.RUN is analyzing the data. This is where particles will be classified and the persistence length will be calculated. Analysis may be achieved through the PSPOLY.ANALYZE command. The analysis command first creates a list of all individual particles in the AFM image, and it proceeds to loop through and apply PSPOLY.CLASSIFY to each one. Classification returns one of the following strings: “linear”, “looped”, “branched”, “branched and looped”, “overlapped”, or “noise particle”. If there are any high points in the particle, the number of such points is also indicated in the string.

Code to mimic PSPOLY.RUN, using the same parameters as were used previously, is below.

```
>>> newCL_example = pspoly.polydat('newCL_example','newCL_0000.png',2,4)
>>> annotated_skeleton = pspoly.prepare(newCL_example)
>>> pspoly.analyze(annotated_skeleton)
```

The mimicking example does not hold any advantage over PSPOLY.RUN. Breaking the code up like this does, however, enable customization. To make use of our new freedom, we will write a custom preparation program to replace PSPOLY.PREPARE. The main functions of the preparation program are to create a binary threshold mask of the original data, skeletonize the mask, locate the individual particles within the skeleton image, and identify the important points in the particles.

Thresholding is accomplished within PS-Poly Python by use of PSPOLY.THRESHOLD. This method takes a polydat object, a threshold function, and possibly a numerical parameter as arguments. A threshold function is a method by which an image is converted into a binary mask. The default option is PSPOLY.FILTERS.OTSU_CENTRAL, which removes boundary particles and applies the Otsu thresholding method, imported from the external package scikit-image.

For the modified preparation program, we will define a threshold function that takes a user-input threshold and turns all values above that number into one, all other numbers into zero.

```
>>> def custom_threshold(img,threshold_value):
...     boolean_array = (img > threshold_value)
...     return boolean_array.astype('int')
```

Skeletonizing data is achieved with PSPOLY.SKELETONIZE. This method is also customizable, taking a polydat object and a skeleton function. It is possible to define your own skeleton function, in much the same way as the threshold function was defined, but we will use the default PSPOLY.FILTERS.SKELETONIZE, which is also imported from scikit-image.

We are ready to define the custom preparation program. The following code is taken from the definition of PSPOLY.PREPARE in the source code, with the thresholding algorithm modified.

```
>>> def custom_prepare(afm_data,threshold_value):
...     binary_mask = pspoly.threshold(afm_data,threshold_function=custom_threshold,t=threshold_value)
...     skeleton = pspoly.skeletonize(binary_mask,skeleton_function=pspoly.filters.skeletonize)
...     particles = pspoly.separate_particles(skeleton)
...     for particle in particles: pspoly.identify_points(particle)
...     return skeleton
```

The particle array is not returned by the function, but it is critical to the function of the algorithm! If branch points, endpoints, and the others are not identified for every particle in the image, the analysis algorithm will not work. The information for each particle is automatically stored in the in a subdirectory of the skeleton data folder, making it accessible through the skeleton object.

Rewriting the PSPOLY.RUN code with a custom threshold value is now possible. Let's use a value of 0.25 and see what happens. Delete the old data folder for newCL_example before running the code to avoid complications; we have not built error handling into the custom algorithm.

```
>>> newCL_example = pspoly.polydat('newCL_example','newCL_0000.png',2,4)
>>> annotated_skeleton = custom_prepare(newCL_example,threshold_value=0.25)
>>> pspoly.analyze(annotated_skeleton)
Particle Type           Number of Features              Average length (nm)             Percentage of Total particleization Length
Linear                          343                             11.8                            41.2%
Looped                          0                               N/A                             0.0%
Branched (no looping)           95                              42.9                            41.3%
Branched (with looping)         17                              74.8                            12.9%
Overlapped                      5                               91.3                            4.6%

Number of High Points: 15
Number of Noise Particles: 1

Persistence Length (nm): 7.3
```

It seems that the distribution of particle types is similar, but not identical, to the distribution found in the simple data analysis section. This is to be expected, given that we presumably used a different threshold value from what was determined by Otsu thresholding. In general, it is best to use PSPOLY.SHOW to examine manually thresholded data before committing to a threshold.

You have now learned all major methods in the software package. Please also note the following:

PSPOLY.LIST_PARTICLES takes a polydat object as an argument and returns a list of all particles in the data, given that PSPOLY.SEPARATE_PARTICLES has already been run.

PSPOLY.GET_HOME returns the save directory of the program, and PSPOLY.GET_SAVE_OPTION returns the option for what to do if a data folder is duplicated.

PSPOLY.GRAPHIFY takes an annotated and skeletonized polydat object as an argument and returns a simple graph representing the connectivity of the identified points in the data.

PSPOLY.LENGTH is a subpackage containing methods that aid in calculating end-to-end length, contour length, and persistence length of skeletonized data.