KNOT USER GUIDE

Last updated: 03/09/2021

Contents

[Introduction 2](#_Toc66281818)

[Changelog 2](#_Toc66281819)

[Abbreviations: 2](#_Toc66281820)

[Part 0: Installing Python and relevant packages 3](#_Toc66281821)

[Setting up the data folders 4](#_Toc66281822)

[Part 1: Simulating data using KNOT 3](#_Toc66281823)

[Making a simulation 4](#_Toc66281824)

[Defining model-based simulations in 2-D 4](#_Toc66281825)

[Generating random-process based simulations in 3-D 4](#_Toc66281826)

[Part 2: Preparing real data for KNOT & KNOT for real data 3](#_Toc66281827)

[Cropping to a suitable ROI 4](#_Toc66281828)

[Matching simulated to experimental PSF 4](#_Toc66281829)

[Adjusting preprocessing strength 4](#_Toc66281830)

[Verifying agreement 4](#_Toc66281831)

[Tuning the linking radius 4](#_Toc66281832)

[Part 3: Running KNOT 3](#_Toc66281833)

[Part 4: The results 3](#_Toc66281834)

[Appendix A: Troubleshooting common errors 3](#_Toc66281835)

[Analysis questions 3](#_Toc66281836)

[Exceptions 4](#_Toc66281837)

[Appendix B: User parameters 4](#_Toc66281838)

[Instrument parameters 5](#_Toc66281839)

[Appendix C: Code overview 4](#_Toc66281840)

[\_\_ENUM.py 4](#_Toc66281841)

[\_\_FUNCTION.py 4](#_Toc66281842)

[\_\_OPERATION.py 4](#_Toc66281843)

[\_\_VISUALS.py 5](#_Toc66281844)

[\_CREATE.py 5](#_Toc66281845)

[\_INITIALIZE.py 5](#_Toc66281846)

[\_PREPARE.py 5](#_Toc66281847)

[\_RECOVER.py 5](#_Toc66281848)

[\_SEGMENT.py 5](#_Toc66281849)

[\_TRACK.py 5](#_Toc66281850)

[crop.py 5](#_Toc66281851)

[main.py 5](#_Toc66281852)

[simulate.py 5](#_Toc66281853)

[USER.py 5](#_Toc66281854)

# Introduction

This user guide will guide you through basic operation and troubleshooting of Knowing Nothing Outside Tracking (KNOT). If something is unclear or you have a question unanswered, you can reach the code designer at (Jorge Zepeda O: [14jazo@gmail.com](mailto:14jazo@gmail.com)) or the corresponding author (Christy F. Landes: [cflandes@rice.edu](mailto:cflandes@rice.edu)).

## Changelog

v1.0.0: Release

## Abbreviations:

DH: Double Helix

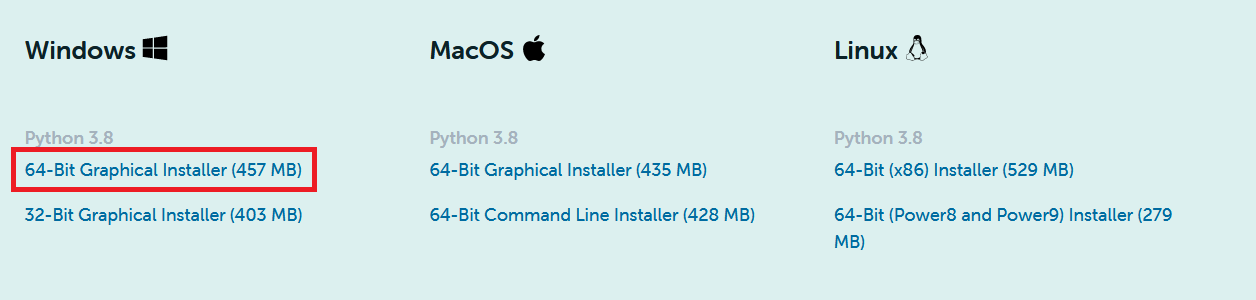
PSF: Point Spread Function

# Part 0: Installing Python and relevant packages

This part of the guide assumes the following:

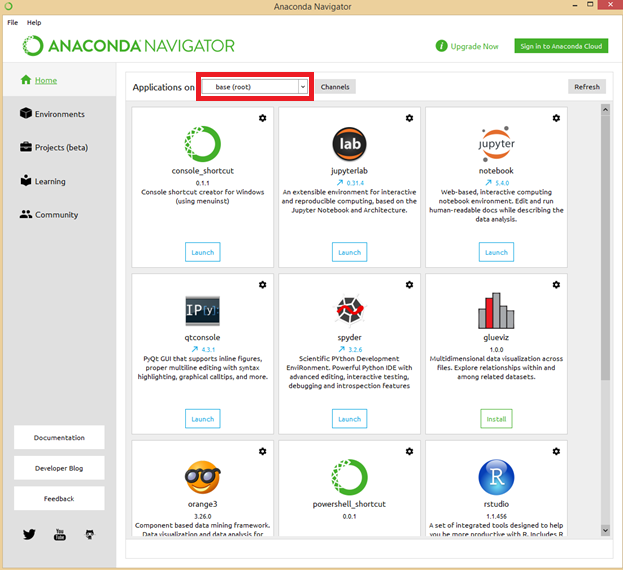
* You **do not** have python installed on your computer
* You **do not** have anaconda for python installed on your computer
* You have a stable internet connection

For its ease of use, we will be installing Anaconda for python (A recent release can be found here: <https://www.anaconda.com/products/individual> ). Download the appropriate installer to install Anaconda.



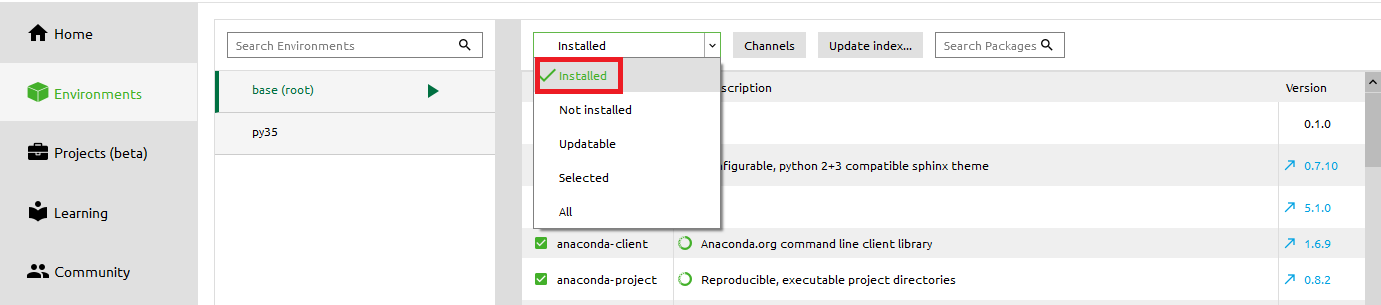
When installing, use all the default settings. Python can get finicky with the PATH variable and we do not want to cause an issue with python.

Once Anaconda is installed, open the **Anaconda Navigator** program shown below. You will see several different applications available to run or download that allow you to run and develop python code.



The base environment should already be selected. **IF YOU HAVE AN EXISTING PYTHON INSTALLATION, CREATE A NEW PYTHON ENVIRONMENT AT THIS POINT.**

Open the **Environments** tab and set the dropdown menu to Installed.

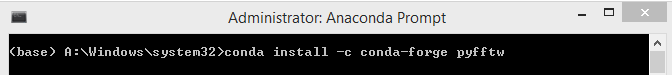


Proceed to use the search bar to ensure that the following packages and their dependencies are installed:

* numpy (*For array manipulations. Absolutely mandatory*)
* imageio (*For loading and saving images. Absolutely mandatory*)
* scipy (*For Delaunay triangulation. Mandatory*)
* matplotlib (*For visualization. Optional but helpful*)
* pyfftw (*For the FFTW. Optional but not installing requires workarounds*)

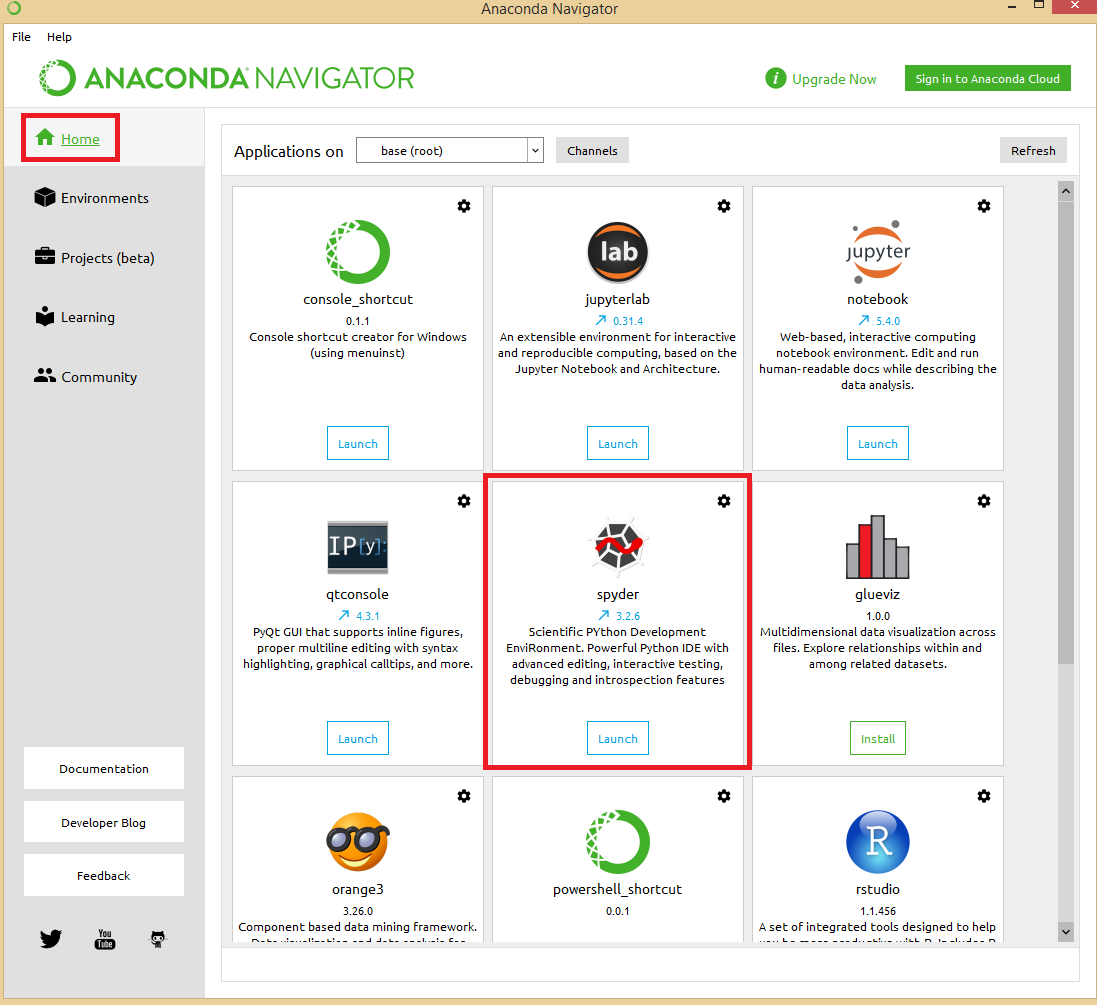
If one of these packages is missing, you can change the dropdown menu to Not Installed and use the search bar to locate these packages and install them. However, the one package you will be unable to install is the pyfftw package. For this package, you will need to open **Anaconda Prompt** likely using administrator privileges. In the command prompt, input the following command and install pyfftw:

conda install -c conda-forge pyfftw

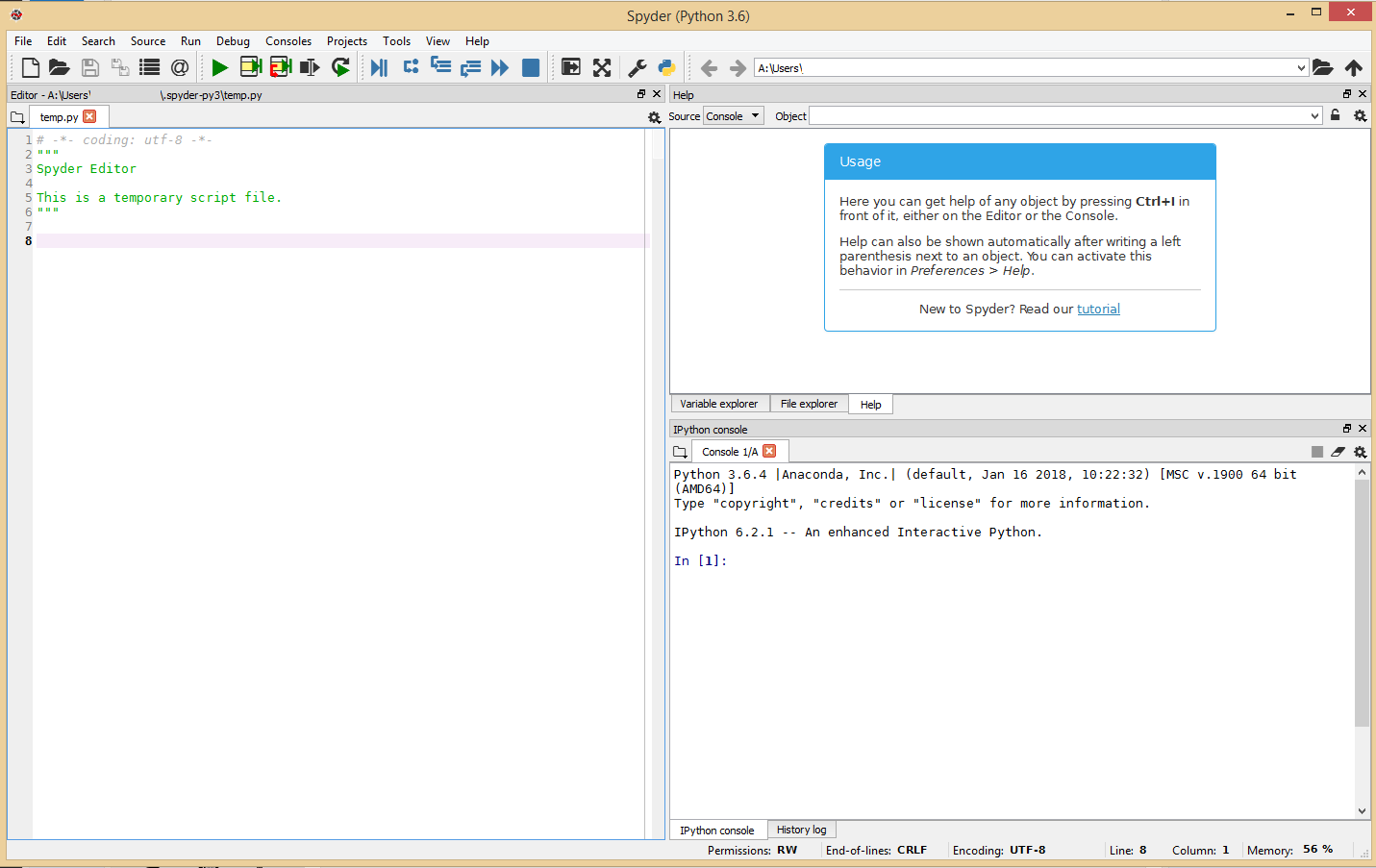


**IF YOU ARE USING A NEW PYTHON ENVIRONMENT, MAKE SURE TO CHANGE ENVIRONMENTS IN ANACONDA PROMPT FIRST.**

Once you are finished installing pyfftw, you may close **Anaconda Prompt** and return to the Home tab of **Anaconda Navigator**.



If it is not already installed, install **Spyder**. Once installed, open **Spyder** from **Anaconda Navigator** (or your IDE of choice – accommodate instructions for your IDE accordingly). You will be greeted with the three-panel interface shown below.



It may be wise to additionally read their tutorial when you have the time.

To get started with this guide, open crop.py, main.py, simulate.py, and USER.py in the Spyder editor and proceed to the next section.

## Setting up the data folders

KNOT relies on knowing where data is stored to keep organized. Before using KNOT, ensure that the following folders are created in your KNOT directory (where main.py is):

* **Apertures** (will autogenerate)
  + Contains .tif files for each aperture simulated
* **Experiment** (will **not** autogenerate)
  + Contains .tif files for cropping and the cropping result
* **Images** (will **not** autogenerate)
  + Contains .tif files used for analysis
* **Matlab** (will autogenerate)
  + Contains .mat files containing particle motion with Troika compatibility
* **Phase Masks** (will autogenerate)
  + Contains .tif files for each phase mask simulated
* **Simulation** (will autogenerate)
  + Contains .tif, .gif, and .xml files for each particle motion simulated
* **Temp** (will autogenerate)
  + Will contain a folder for each data processed, which contains .tif and .json files needed to pick up where analysis left off
* **Truth** (will autogenerate)
  + Contains .xml files; only needed to compare against ground-truth data

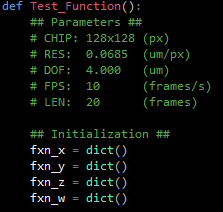
Once created, proceed to **Part 1** if you are simulating data, or proceed to **Part 2** if you are analyzing experimental data.

# Part 1: Simulating data using KNOT

Provided that we do not have any experimental data, KNOT is able to generate data on its own using its Function objects. Defined in \_\_FUNCTION.py and explained in the Code Overview, KNOT can work with semi-analytical functions like polynomials, sinusoids, and exponentials. In this section, we will describe the process to create our own 2-D particle motion that emulates a known model and one involving a mixture of Brownian and non-Brownian motion in 3-D.

## Making a simulation

To make a new simulation, open \_CREATE.py and define a new function in the **SIMULATION DEFINITIONS** section. Here we are creating Test\_function():



**It is important to write down the parameters used in the simulation in case you need to re-simulate it.** Adjust the appropriate variables in USER.py now. Suggested parameters are shown in the image above. Secondly, it is required to initialize four dictionaries, ideally titled fxn\_x, fxn\_y, fxn\_z, and fxn\_w for the x, y, z positions of particles and their weight or intensity. If not supplied for a particle, the value in the appropriate dimension will be zero (or 1 in the case of fxn\_w). It is also imperative that these functions be returned as a tuple at the end of the definition as follows:

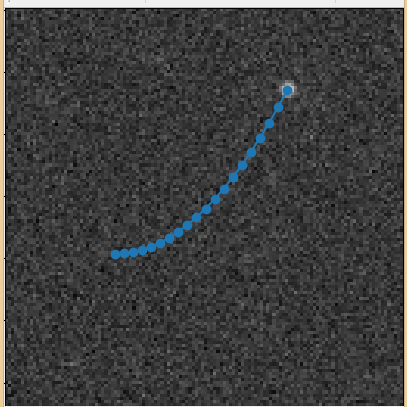


## Defining model-based simulations in 2-D

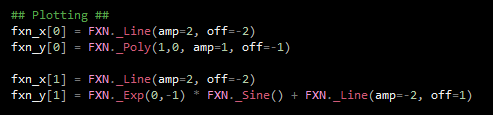
To test things out, we will first simulate a particle (particle 0 in blue) in a parabolic path. Ensure that your KER\_Z and KER\_T in USER.py are set to 1 for now.



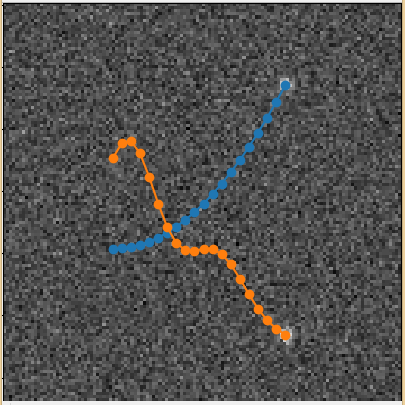
This will generate the following image which shows the trajectory of the particle and the last frame encountered:



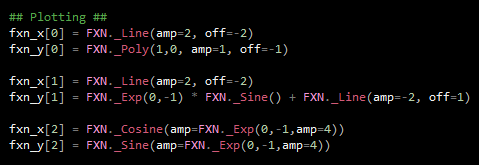
Congratulations! We have constructed our very first particle simulation! We can of course make things more interesting by adding more particles. Introduce another key (1) into fxn\_x and fxn\_y, this time with a decaying, moving sinusoid:



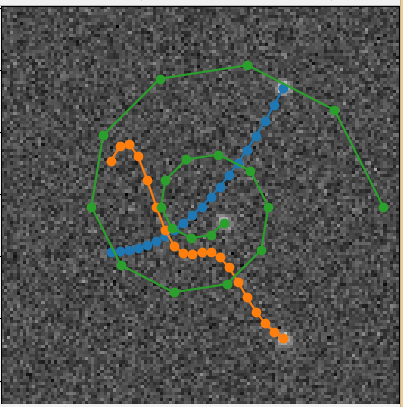
Which returns a more complicated image (particle 1 is orange):



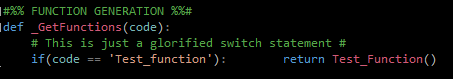
As is evident, Function objects can be added, subtracted, multiplied, and divided as one would expect. Additionally, function composition is also possible, as shown here for particle 2 (green):



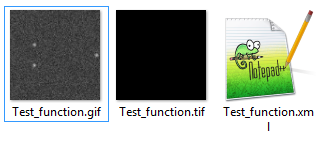
Though the result obviously needs to be slowed down to be able to track it effectively:



Now that we are content with our simulation, add the function code and function definition to the \_GetFunctions(code) function:

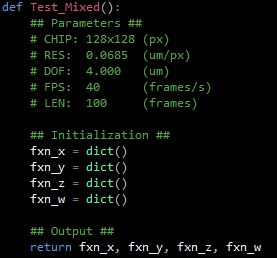


And we can run simulate.py once more to solidify our results. We will see that three files have appeared in the **Simulation** folder: a .gif of the result for viewing, a .tif of the simulation for analysis, and an .xml file containing the motion in a manner usable with Icy.

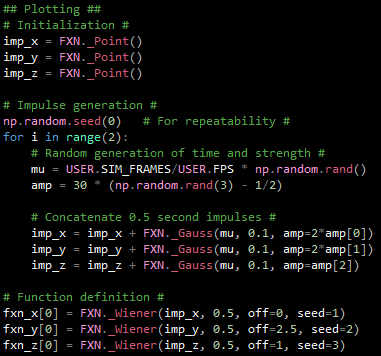


## Generating random-process based simulations in 3-D

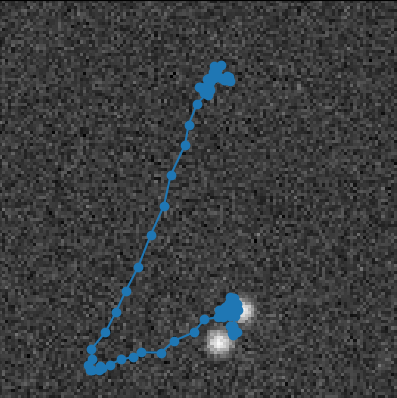
Now that we understand how to create simulations in 2-D, we can extend our abilities to 3-D and with non-differentiable motion. Start by creating a new function definition with slightly higher temporal resolution, Test\_Mixed():



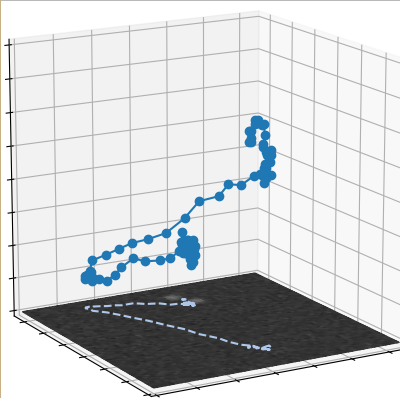
Here, we will introduce a couple of short, but randomly generated impulses:



Giving us the rather interesting result of:



in 2-D, with the 3-D result looking like:



As such, this simulation has phases of Brownian motion coupled with instances of random directed motion. More complex motion can be built using concatenation and composition of functions, but this is enough of a primer to get your feet wet with simulating data.

And as always, do not forget to add your function to the \_GetFunctions block!

A tip going forward:

* Always make sure to have different seeds for each random process and define the seeds explicitly for repeatability.

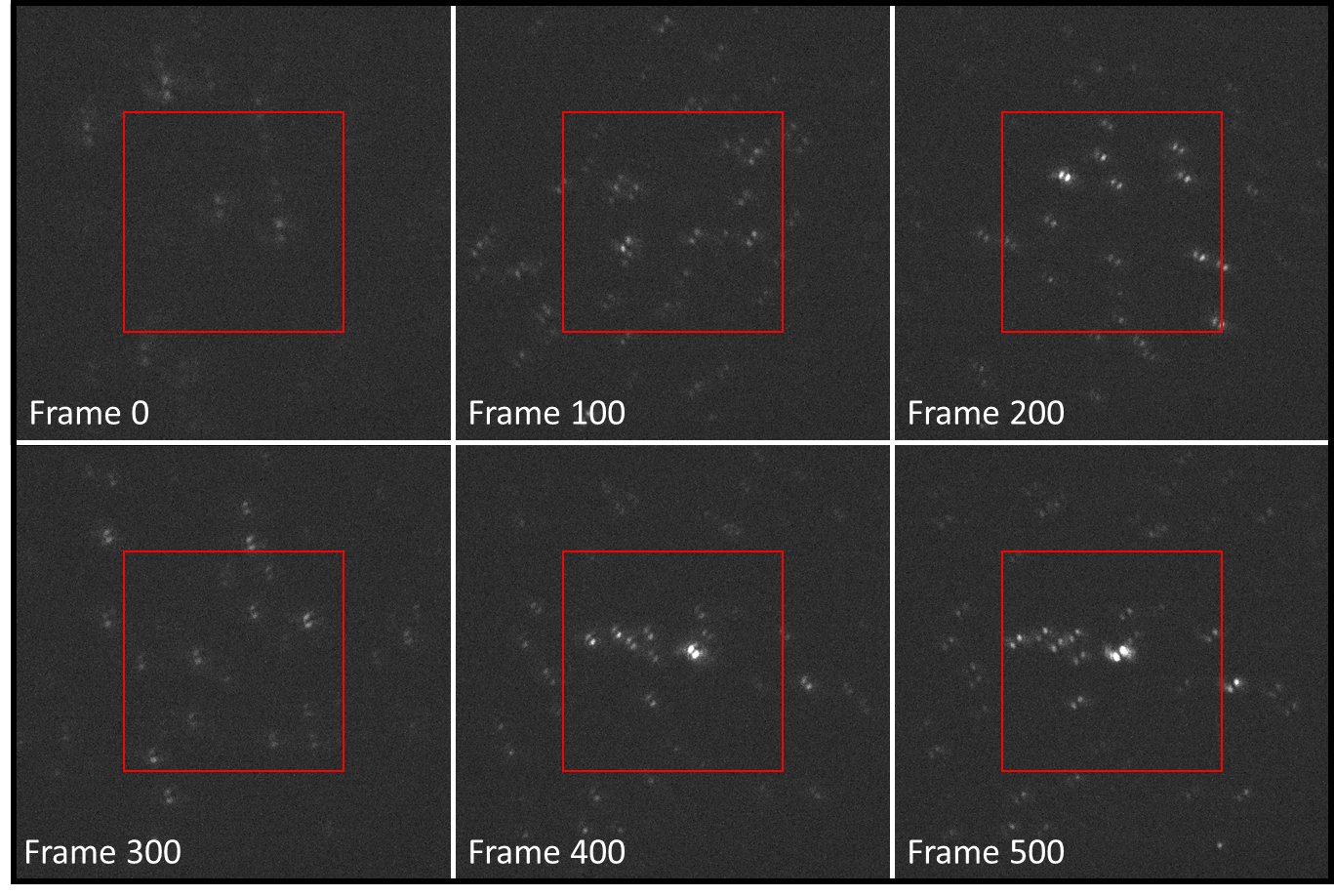
You may now proceed to **Part 3** of this guide if you do not wish to use experimental data. Else, continue to **Part 2**.

# Part 2: Preparing real data for KNOT & KNOT for real data

Simulating data is not an effective way to learn more about the real world, so KNOT needs to be able to access your experimental data to process. For best results, it is recommended to do the following:

* Crop to regions of interest (ROI) (speeds up processing time greatly)
* Ensure simulated PSF is the same as experimental PSF (needs to only be done once per instrument. N.B. if you load an experimental PSF, it may be best to always use the same dimensions as your image.)
* Ensure preprocessing variables are suitable to your data (this should only need to be done once.)
* Ensure the linking radius is appropriately set (this may be data dependent)

For this example, we will be using a 600x600 movie with 500 frames. This movie contains polystyrene beads on glass attached to a piezoelectric stage moving with a predefined motion. As such, we know *a priori* what the particle motion will be. Below is a screenshot of each hundredth frame for reference:



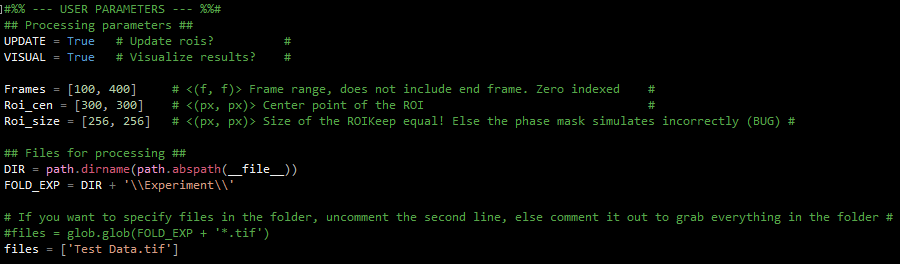
When calibrating, it is best to isolate one or two strongly fluorescing emitters to compare with. The increase in processing speed between a 256x256 and 64x64 image will allow for a quicker calibration process without sacrificing accuracy.

## Cropping to a suitable ROI

The first thing you may notice in the image above is the red box in the center. This box captures a 256x256 area which we will denote as our ROI. We select this region because the emitters in the image appear brightest here, as well as having strong intensities during the last hundred frames. Because the intensities in frame are relatively low, we will focus our analysis on frames 100 to 400.

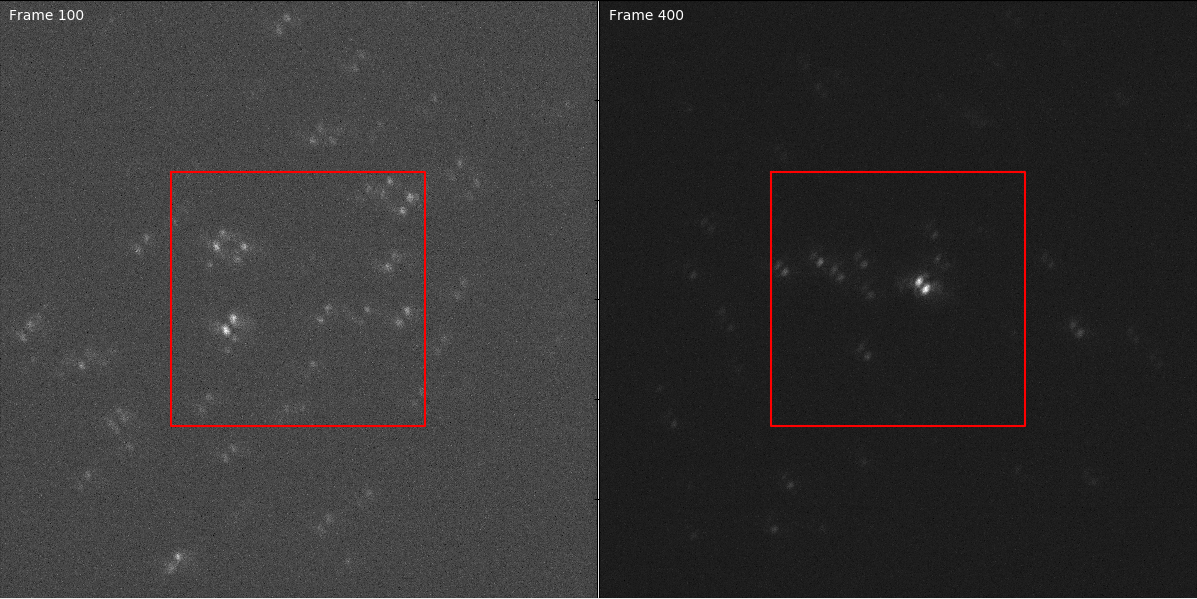
**Please note: it is best to crop out regions of equal dimensions and are a power of two. That is, 256x256 is preferable to 256x512, which is preferable to 273x291. Other highly composite numbers such as 120, 180, 360 are useful for speeding up the FFT.**

Opening crop.py, we can modify the USER PARAMETERS section to construct the appropriate ROI by modifying the Frames, Roi\_cen, and Roi\_size parameters as such:

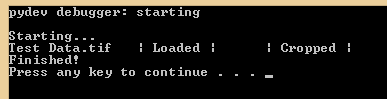


Also note that we have assigned files to ‘Test Data.tif’, which is the filename of the movie under the Experiment folder. If multiple files are to be cropped identically, you can add more filenames to the files list, or select all files in the Experiment folder by commenting out the list definition.

Running crop.py will present the user with two sample images from the movie at the first and last frame of selection, including the cropping region as shown below. If you agree with the ROI taken, then close the figure to resume the program. Else, halt debugging and adjust the ROI before restarting.

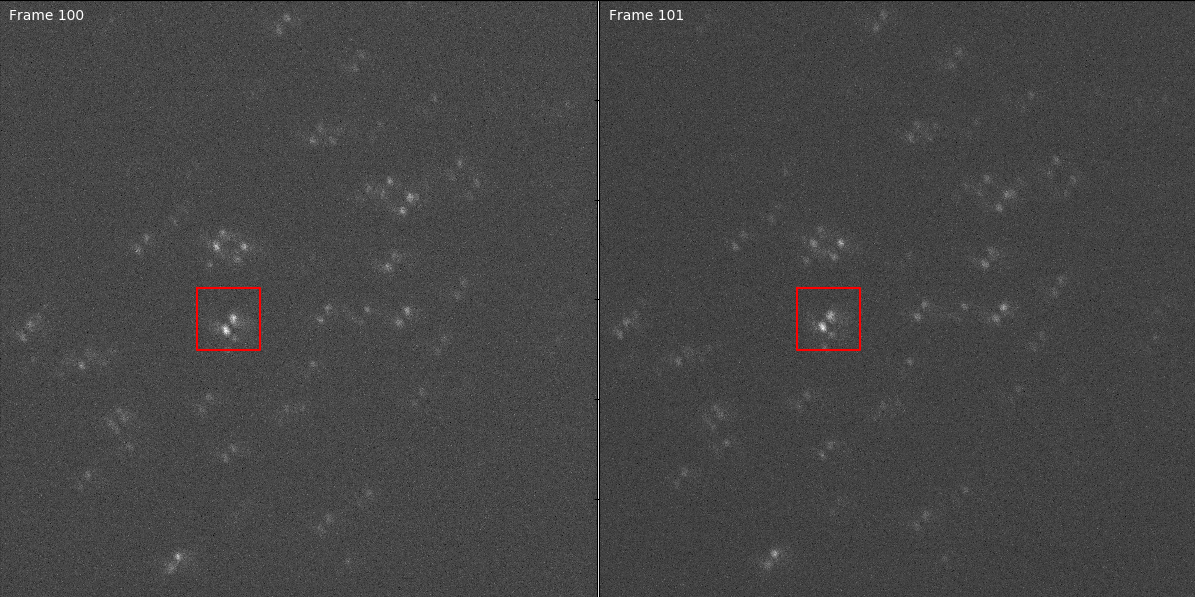


During the cropping process, KNOT will display the current progress in the console, and let you know when cropping has completed.



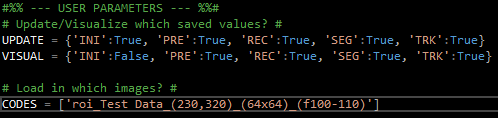
After cropping, move the roi\_xx.tif files to the **Images** folder, perhaps with an additional renaming for ease of use. (For clarity, we will not be doing this here.)

We will do a similar crop on a smaller region (64x64, frame 100) for calibration in the next section:

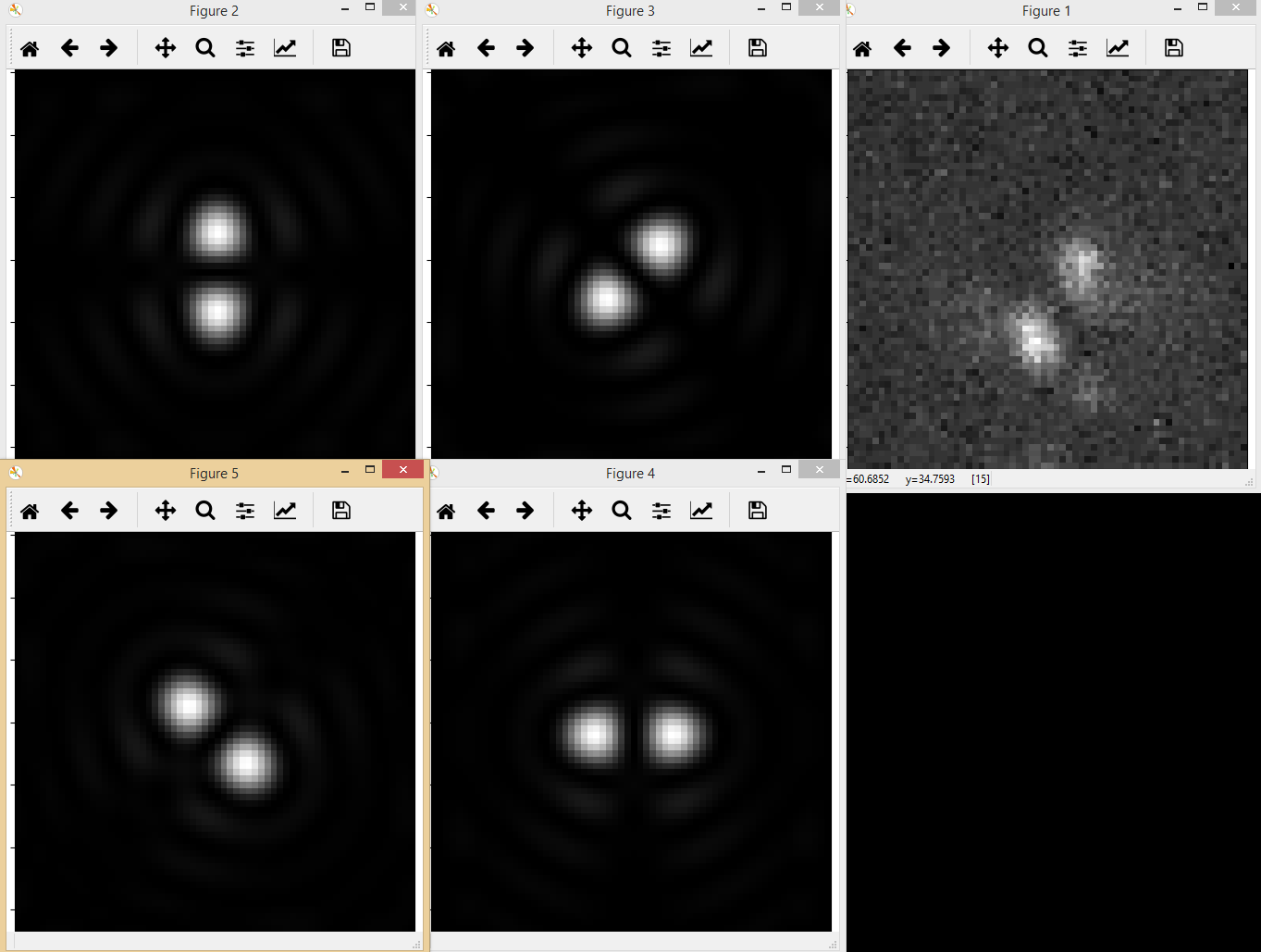


## Matching simulated to experimental PSF

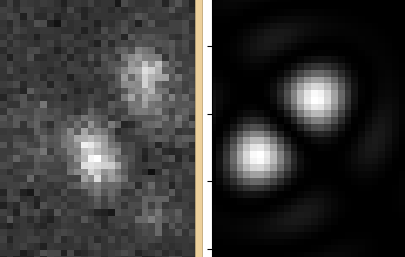
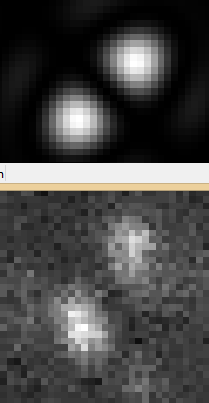
To ensure that ADMM will process the image properly, we must ensure that KNOT’s simulated PSF equals the experimental PSF. To do this, open main.py and ensure that all fields in the VISUAL dictionary are set to True. Additionally, **write the filename without the .tif extension** as the first entry in our CODES list.



Once debugging begins, the first figures you will encounter include four samples of the simulated PSF and the first frame from the image:



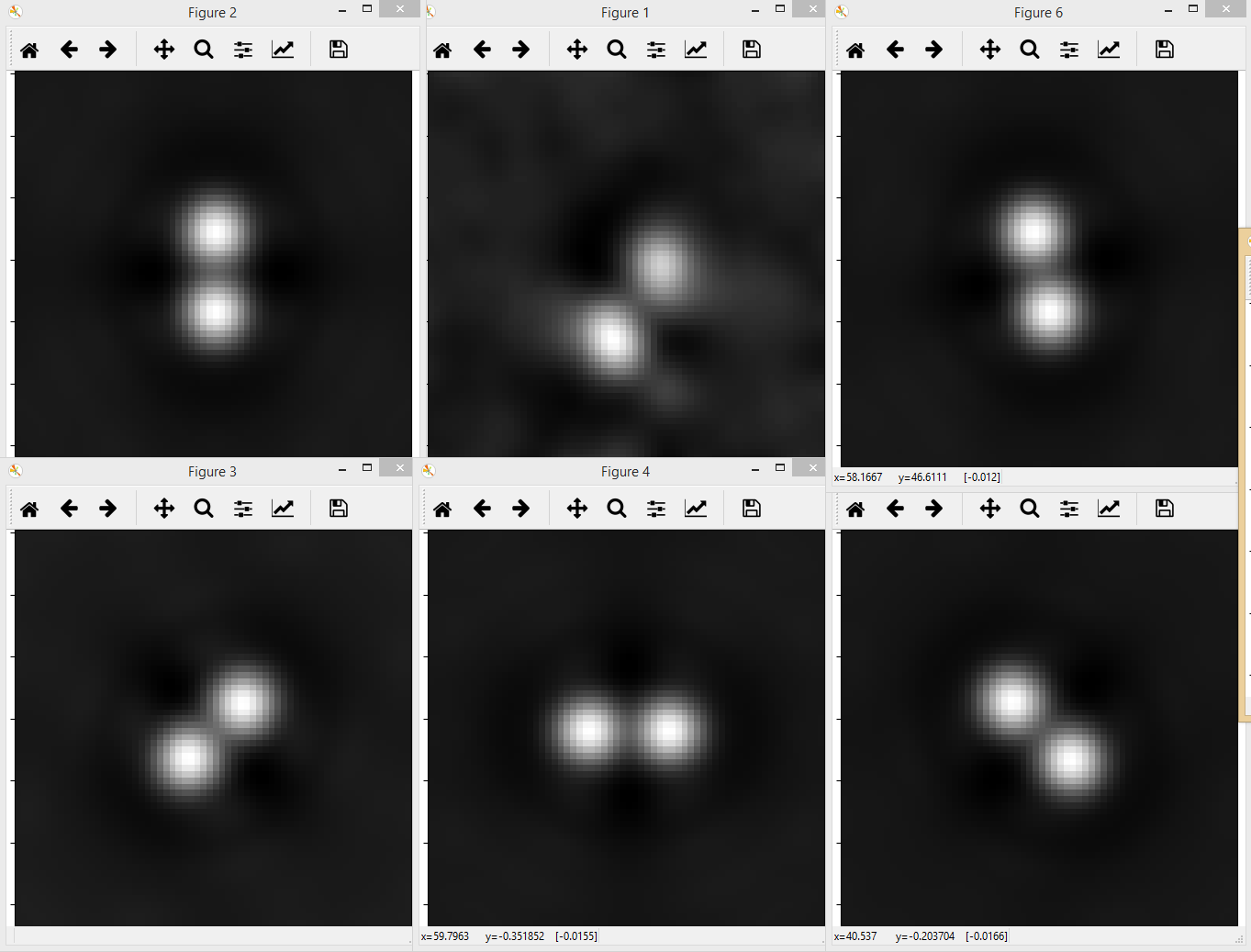
To ensure a good match, find the PSF sample that most closely corresponds to a bright particle in the image and match the two figures together as close as possible:

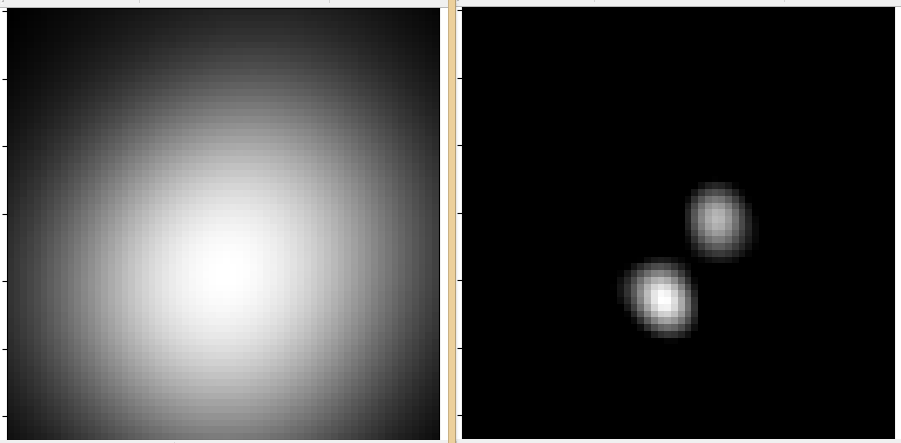
If the lobe size and separation is appropriate, then proceed to the next step. Else, halt debugging and adjust the aperture radius (APR\_RAD) and lobe separation (KER\_SEP) in USER.py as desired. We can further check the similarities when performing ADMM, as good matches will return resolved particles while poor matches will return nothing. **Make sure that your instrument parameters are correct before manipulating these values!**

## Adjusting preprocessing strength

Once the PSFs match, close the figures spawned from the initialization step (or set the ‘INI’ key in VISUAL to False.) KNOT will pre-process the movie and return a similar set of figures to double check your work, this time with the first frame in the upper middle and samples of the PSF surrounding:



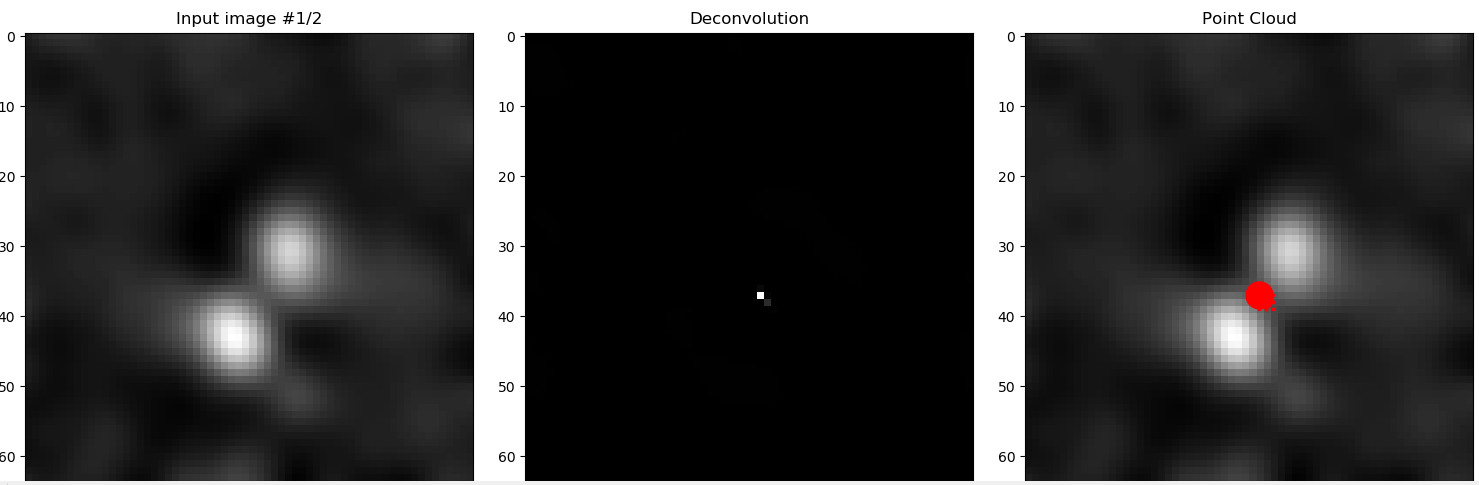
Adjust the sizes of the background subtraction (PRE\_BG), noise suppression (PRE\_NS), local threshold (PRE\_LT) filters as desired. It should be noted that PRE\_BG should be greater than APR\_RAD to prevent the filter from removing the PSF itself. Similarly, PRE\_NS should be greater than the camera resolution RES but smaller than APR\_RAD, else the convolved PSF will be considerably larger. This may come as an advantage in low SNR settings, but large PSFs may also reduce localization precision. The value of PRE\_LT should be greater than the background subtraction, with no bound on how large it can be. Please note that large values PRE\_LT remove the locality of the threshold and in effect act as a global threshold. The figure below shows the local threshold, which would be considered “too large” for a small image such as our 64x64 calibration but is standard for a larger 256x256 image. Additionally, the right panel shows the preprocessed image as filtered by the local threshold.



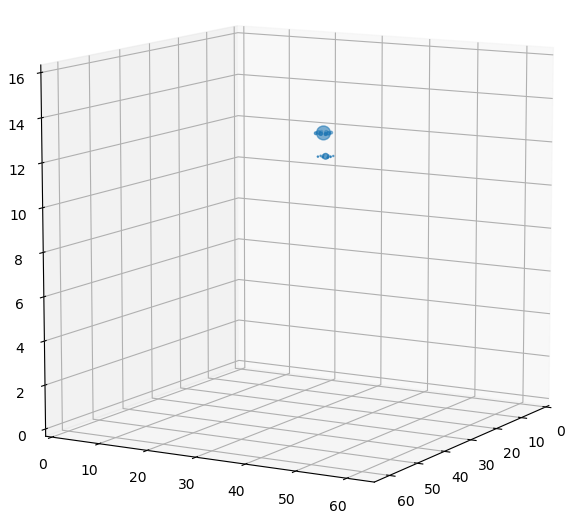
For highly mobile particles in relatively static backgrounds, introducing the temporal smoothing filter (PRE\_TS) may benefit you for appreciably large amounts of frames (50+).

## Verifying agreement

Once satisfied with the preprocessing parameters, close the figures spawned from the preprocessing step (or set the ‘PRE’ key in VISUAL to False.) KNOT will then begin recovering emitter positions using ADMM. **Usually, this step takes the longest, so it is important to have a small movie for calibration and testing.** Once ADMM has completed, two figures will appear. The first contains the input image for the current frame, the deconvolved result in 2-D, and the corresponding 2-D point cloud overlaid on the input image:



The second figure displays the point cloud in 3-D to show its structure:



In this case, KNOT was able to localize the emitter to a small (here, approx. 4x4x2 voxels) region, indicating that we have sufficiently tuned KNOT to our experimental PSF.

## Tuning the linking radius

Once KNOT knows what PSF to look for, it can be run on experimental data from here on out. It may also be best to re-simulate any motion previously simulated using these parameters for best results.

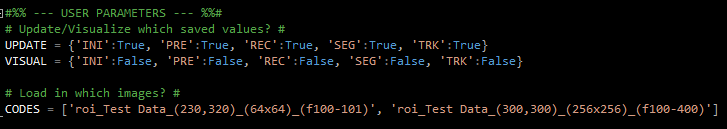
However, we do not know *a priori* what linking radius (TRK\_RAD) to use. Unfortunately, this needs to be found by trial and error. Some tips to use however:

* **Do not set TRK\_RAD too high!** KNOT will attempt to find *something* to link for each frame, so setting TRK\_RAD to anything beyond 1 micron should be done only for very sparse, fast moving particles. If you find your resultant trajectories strongly jittering about, consider lowering the linking radius.
* **Do not set TRK\_RAD** **too low!** Similarly, setting TRK\_RAD to anything below 1 or 2 pixels will likely only capture confined motion, if any at all. The best range to search is between ~300 nm and ~800 nm.
* **Consider your frame rate!** Not that the linking radius acts *per frame* not per second. For a frame rate like 30 ms, large jumps in position are unlikely. If you are analyzing frame rates closer to 100 ms or greater, then a larger TRK\_RAD is acceptable.

After testing values for the linking radius (which may depend on the data analyzed) you are ready to run KNOT on bigger datasets. **Continue to Part 3.**

# Part 3: Running KNOT

The interface provided in main.py remains the same if you are analyzing simulated data or experimental data. In particular, the only section that needs interacting with once KNOT is set up is the **USER PARAMETERS** section:



**UPDATE** keeps track of which sections (Initialize, Pre-procesing, Recovery, Segmentation, or Tracking, respectively) of the analysis need to be updated. If set to False, KNOT will check if there is a file already saved for that section of analysis and will opt to load it if present instead.

**VISUAL** keeps track of which sections of analysis should be displayed. Keep in mind that visualization pauses execution, so it is best to keep these all False when doing batches of analyses.

**CODES** determines which files to analyze. It is required to be a list of extension-less filenames, and KNOT will search for the relevant files (with the appropriate extensions). **Do not place file paths here**, as KNOT will not know how to interpret the backslash.

Some additional tips and tricks:

* Once ADMM has been run on a dataset, there is little need to re-run it. As such, if you need to adjust something in the segmentation or tracking steps, turning the UPDATE keys ‘INI’, ‘PRE’, ‘REC’ to False can save you a lot of time. However, if you do decide to re-run the recovery step, it is wise to set all three flags to True in case of any issues encountered when saving the pre-processed images.
* It is possible to analyze data in parallel or in series. The number of parallel processes to start (PAR\_CORES in USER.py) should not exceed 6, else the overhead of parallel processing consumes more time than performing each analysis in turn. 3 processes is usually a good amount.

# Part 4: The results

Once KNOT has processed or is in the middle of processing a movie, it generates several files:

* In the Temp folder:
  + **CODE\_eps.tif** – The local threshold used each frame
  + **CODE\_ker.tif** – The 4-D phase mask kernel used for deconvolution
  + **CODE\_prep.tif** – The pre-processed version of the movie
  + **CODE\_pts.json** – A JSON file containing the location and weight of each point resolved before clustering into clouds.
  + **CODE\_clouds.json** – A JSON file containing each point cloud identified by KNOT to re-create PointCloud objects
  + **CODE\_tracks.json** – A JSON file containing the tracking results to re-create Particle objects.
* In the Matlab folder:
  + **CODE.mat** – A single variable Matlab file containing a 3-D matrix formatted as [time, dimension, particle]. This is equivalent to Troika’s representation of trajectories and is therefore compatible with Troika-related analyses.
* In the Evaluation folder:
  + **CODE.xml** – An XML file organized like the ISBI 2012 particle tracking challenge results. Further analysis, such as comparison to ground truth trajectories, can be carried out using [Icy](http://icy.bioimageanalysis.org/).
* In the Truth folder:
  + **CODE mot.json** – A JSON file created when simulating motion that describes the actual simulated motion.

These files are used by KNOT to save parts of the analysis for returning to later, or for outside analysis.

# Appendix A: Troubleshooting common errors

If you find an issue that is not presented here in some form, please contact us!

## Analysis questions

**KNOT didn’t return anything! / KNOT returned a bunch of garbage! What happened?**

In instances of low SNR or improperly aligned PSFs, KNOT can return either nothing or essentially noise from deconvolution. For instances where KNOT returns nothing, try re-calibrating the simulated PSF to suit the experimental PSF and re-running the analysis. Returning nothing could also be a sign that the threshold was set too high, requiring modifications to \_RECOVER.py or \_PREPARE.py.

For instances where KNOT returns what amounts to a huge mess of a point cloud, the reason is usually that the SNR was too low (and so KNOT attempted to grab everything it could). There is no easy fix for this. The following contingencies may help:

* Modify \_RECOVER.py or \_PREPARE.py to be less sensitive (For \_RECOVER.py, try setting the variable rhs to something higher. For \_PREPARE.py you may need to adjust variables such as eps\_global.)
* Adjusting the preprocessing filters to make the PSFs bigger, such as by making PRE\_NS larger.
* Batching frames together (outside KNOT) to increase SNR at the cost of temporal resolution.

**Why are my trajectories jittery?**

If two particles are too close together (within TRK\_RAD) and KNOT fails to detect one for a frame, the first particle will seem to “jump” to the location of the second particle until the opposite happens. This can, and hopefully will, be remedied by asking KNOT to look ahead a few more frames (rather than just the next frame) to determine if the original particle re-appears within the blinking tolerance. Otherwise, the next best thing to do is to lower TRK\_RAD.

**Why won’t KNOT work on non-square images?**

This is currently a minor oversight in the calculation of the simulated PSF. When a fix is found, the repository and this user guide will be updated. The result is a malformed PSF that effectively cannot be used, even in simulation.

## Exceptions

**FileNotFoundError ‘No such file or directory’**

This error can occur for several reasons when either creating a simulation or running data. Check to make sure that all folders (Apertures, Evaluation, Experiment, Images, Matlab, Phase Masks, Simulation, Temp, Truth) are present in the KNOT directory. These folders generate empty automatically when running any

Additionally, when analyzing a data code, make sure that **the** **extension is not included**. That is, write ‘Test’ rather than ‘Test.tif’.

# Appendix B: Code overview

It should be noted that the code base is split up into three categories based on the name prefix:

* \_\_NAME.py indicates a python module that contains global constructs that both span and are independent of the KNOT process. It is advised to leave these files untouched.
* \_NAME.py indicates a python module that contains functions and classes necessary for KNOT to process data. The filenames are selected to illustrate the order of execution.
* NAME.py indicates a python module that acts as a controller for the user.
* name.py indicates a runnable python module to perform a specific task.

If you are looking to extend or modify any part of the KNOT code, please make sure to read this section and any associated comments in these files.

## \_\_ENUM.py

This module contains enumerators used for various purposes such as determining the file format used, function codes for \_\_FUNCTION.py, and the phase mask or localization scheme being used.

## \_\_FUNCTION.py

This module contains the Function class, which allows for semi-analytic construction of motion using functions such as polynomials, sinusoids, or Gaussians as well as non-differentiable motion such as Wiener processes (also known as Brownian motion). The actual usage of the Function class is relatively limited and may be expanded in the future to include additional functionality. A brief tutorial is given in Part 1 and secondary information is given below. Importing this module is usually denoted as follows:

import \_\_FUNCTION as FXN

Function objects are constructed using the following template:

\_Handle(argument 1, … argument n, amp=amplitude, off=offset, seed=seed)

Which will construct a function fitting the handle provided with arguments 1 through n. The function will be scaled by the value provided in amp and offset from 0 by the value provided in off. For random processes, the seed to use is provided in seed. All arguments have default values (function arguments vary with the function being used, amp=1, off=0, seed=0) and the number of function arguments provided depends on the function used. When writing functions in equation form, amp will be written as and off as . Below is a list of functions, their arguments, and descriptions:

* Point() – A point in space; no arguments, amp has no effect, off shifts the point and is the only argument considered.
* Line() – A line in space; no arguments, amp determines the slope, off shifts the starting point.
* Poly(cn, cn-1, …, c2, c1) – Constructs an nth degree polynomial with coefficients cn­ through c1. amp determines the scale of the polynomial (and is recommended to be left at 1) while off affects c0. The number of arguments input determines the degree of the polynomial.
* Sine(f=1, phi=0) – Constructs a sine wave with frequency f and phase phi. Note that f is determined to be in oscillations per unit, so the default parameter will cycle once every unit. phi is evaluated in radians.
* Cosine(f=1, phi=0) – Constructs a cosine wave with frequency f and phase phi. Behaves like Sine(f, phi) but with the phase shift.
* Exp(mu=0, k=1) – Constructs an exponential with shift mu and rate constant k. Note that negative values of k correspond to a decaying exponential.
* Gauss(mu=0, sigma=1) – Constructs a non-normalized 1-D Gaussian distribution with mean mu and standard deviation sigma.
* Lorentz(mu=0, gamma=1) – Constructs a non-normalized 1-D Lorentzian distribution with mean mu and FWHM gamma.
* Wiener(mu=0, sigma=1) – Constructs a Wiener process with drift mu and standard deviation sigma. Note that these parameters define the underlying Gaussian distribution which the Wiener process accumulates and is dependent on the time steps provided in the domain. See [the Wikipedia page](https://en.wikipedia.org/wiki/Wiener_process) for more information about the Wiener process.

It should be noted that currently Function objects can be added, subtracted, multiplied, and divided using the normal python operators. (e.g. h = f + g) Additional function composition is also possible. For example, if you wanted to obtain a decaying sinusoid, you may construct it as follows:

Decay = FXN.\_Exp(0, -1)

Decay\_Sine = FXN.\_Sine(amp=Decay)

The type of stop-and-go motion observed in cells may be simulated using several Gaussian impulses in a Wiener process:

dom = np.linspace(0, 10, 1001) # The domain of evaluation #

imp = FXN.\_Point() # Initialization #

for i in range(10):

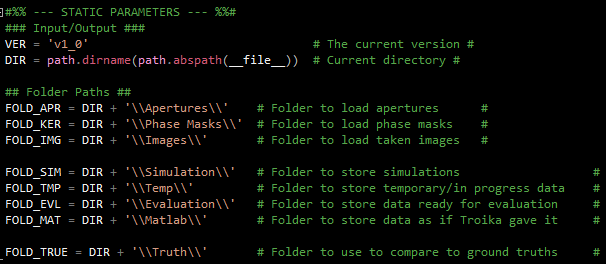
mu = 10 \* np.random.rand() # Generate a number between 0 and 10 #

imp = imp + FXN.\_Gauss(mu, 0.1) # Impulses are ~0.5 seconds long #

motion = FXN.\_Wiener(imp) # Brownian motion with impulses #

## \_\_OPERATION.py

This module handles various input/output methods for loading and saving data. The only parameters that may be of interest are the folder paths dictated at the top of the file:



Modification of DIR or any of the folder paths will cause KNOT to create new directories in those locations if not already present. As such, it is recommended to copy (preferably cropped) images into KNOT’s Images folder before running.

## \_\_VISUALS.py

This module contains helpful building blocks for visualization of data analysis or processing progress.

The \_ProgressBar function is used commonly for long operations to show the current progress in the console.

The methods listed under **STATIC METHODS** are useful for visualizing SFD distributions (\_DispSFDDist), displaying 2-D images on 3-D axes (\_Disp3Dimg), showing the time dependence of trajectories using a saturation and value gradient (\_DispLineGrad), and a helpful tool to draw images in figures at certain locations on the screen (\_VisImg).

For examples, the code that constructs various figures in the manuscript is shown in the section labeled **MANUSCRIPT FIGURES**.

If you wish to modularly create new visualizations, it may be best to place them as new functions in this module.

## \_CREATE.py

The module that contains information for particle simulation. All data in here is intended to be static, and function definitions are intended to be accessible through \_GetFunctions(code). See **Part 1** for an example of how to modify this file.

## \_INITIALIZE.py

The instrument simulation module. Contains several static methods for generating consistent meshes for lateral (\_MeshLat), Fourier (\_MeshFou), and meta-dimensions (\_MeshMeta). Additionally defines how the DH PSF rotates (\_FxnRot), separates (\_FxnSep) and stretches (\_FxnStr).

\_INITIALIZE.py contains the Microscope class, which can simulate apertures (SimAperture), phase mask PSFs (SimKernel), or whole images (SimImage). Available parameters for use outside of the class include: apr (aperture), ker (kernel) img (image), and mot (particle motion).

This class should be modified for development of new simulated PSF models such as the astigmatism or the tetrapod.

## \_PREPARE.py

The preprocessing module. Contains static methods for applying and getting preprocessing filters (\_ApplyFilter and \_GetFilters, respectively). This module also contains two preprocessing methods (\_Preprocess and \_Preprocess2), the latter of which includes time smoothing and should be used.

## \_RECOVER.py

The deconvolution module. Contains static functions to generate the point cloud thresholding filters (\_IDFilters), to perform recovery via ADMM (\_Recover), to chunk an image into smaller areas (\_Chunk). Additionally, this module defines the ADMM class which can recover emitter locations using the FFTW and ADMM.

## \_SEGMENT.py

The particle identification module. Contains static functions for breadth first search (\_BFS) and Delaunay clustering (\_DelaunayCluster). This module also contains mini-methods for thresholding and separating point clouds (\_CloudThr and \_Separate, respectively) along with the main method of particle identification (\_Identify).

This module also defines the PointCloud class, which acts as a structure to hold related points. The PointCloud class keeps the emitter location information in several forms for easy access.

## \_TRACK.py

The tracking module. Contains many static methods for creating single frame displacement distributions (\_SFD, \_SFDwgt, \_subSFD, \_subSFDwgt), Lorentzian kernel density estimations (\_LKDE), and performing random sample consensus (\_RANSAC). Depending on if subframe is desired, \_Track or \_SubTrack may be used to track particles through time. After tracking, \_Stitch is used to apply RANSAC to the trajectories, ensuring links between similar trajectories where possible.

Additionally, this module contains the Particle class, which holds a list of PointClouds and historical SFD information. There are two separate methods for on-frame (Link) and sub-frame linking (Sublink).

## crop.py

The module that crops images to form a specific region of interest in space and time. **Press F5 to crop the images present in the Experiment folder.**

## main.py

The main module used to run KNOT. **Press F5 when ready to run KNOT.**

## simulate.py

The module that simulates particle motion using functions defined in \_CREATE.py. **Press F5 when ready to simulate particle motion.**

## USER.py

The module that contains all the user parameters needed to run KNOT. See Appendix C for descriptions of each parameter.

# Appendix C: User parameters

This section contains a detailed list of everything KNOT uses and what it means

Apart from the list of files to run KNOT on and the dictionaries for updating or visualizing segments of the code, all user-defined parameters are present in USER.py, and are sorted by their appearance in the workflow. Detailed descriptions of each parameter follow, organized as follows:

**Parameter**: <units> [Available Enum Values] Description | Default Value

Please note that the value of units also describes the organization of the variable. For example, (#, …, #) describes a tuple, [#, …, #] describes a list, [[#, …, #]] describes a 2D (or higher if there are more brackets) array, and a simple # denotes a singular value.

Additionally, the following abbreviations are used for convenience:

px: pixel

um: micron

s: second

f: frame

Enum: Enumerator (acceptable values are provided in the description)

## Instrument parameters

**CHIP**: <(px, px)> The number of pixels along each dimension (x, y) of the camera. Only applicable when simulating new images, as KNOT uses the dimensions of the input image otherwise. | (64, 64)

**RES**: <(um/px, um/px)> The lateral resolution of the camera along each dimension (x, y). | (0.0068.5, 0.00685)

**DOF**: <(um, um)> The Depth of Field used when performing 3D tracking. Currently, the first parameter is used when doing XYZ tracking using the DH-PSF, while the second parameter is used when doing XYZT tracking using the stretching lobe phase mask. Other phase mask types (such as Astigmatism and Tetrapod) may be included in the future, and will be included in this variable. | (4.000, 1.600)

**FRATE**: <s/f> The frame rate of the camera in seconds per frame. Used more for visualization than actual computation. | 0.030

**KER\_TYPE**: <Enum> [NONE, HELIX] The phase mask to simulate for deconvolution or simulation. The HELIX value also supports the Airy disk PSF provided that KER\_Z and KER\_T are both 1. | PM.HELIX