ParticleTracker

Release 2.0.0

Mike Smith, James Downs

Jun 30, 2022

CONTENTS

CHAPTER

ONE

STANDARD USE

1.1 Introduction to ParticleTracker

Particle tracking can be useful for many branches of science. There are a number of libraries available that can be used to build sophisticated tracking algorithms specific to a particular experiment or use case. However:

1. you need to be have a reasonable level of proficiency in coding.
2. finetuning the different stages takes quite a long time.
3. different projects have different tracking requirements: some you just need the positions as quickly as possible, some particles aren't circular, some change size with time. Hence you are likely to use different algorithms all with a different interface.
4. visualising the results requires writing more code
5. why reinvent the wheel every time you start a new project, its likely that the novel bits you want to add are a small fraction of the overall process.

Based on a desire to address many of these issues we came up with ParticleTracker. ParticleTracker is a fully gui based particle tracking software that requires minimal (or in some cases no) programming experience. There are faster particle tracking codes, better particle tracking codes but often you just want something that is easy to setup and produces good results in a sensible amount of time. A lot of that is about integrating different tools and packaging them so they can be used easily and intuitively.

ParticleTracker incorporates several different tracking algorithms with a standard interface to help make it quick and easy to setup different particle tracking projects. Depending on what you want to achieve this should be possible without any coding ability. On the other hand we've also designed the project so that you can easily add to and extend the code. Importantly however, you are just coding the bit that needs your novel input.

1.1.1 Reporting Issues

We aim to test this software against the testdata described in the tutorials however bugs do slip through. If you become aware of an issue please report it https://github.com/MikeSmithLabTeam/particletracker/issues

1.1.2 Citing ParticleTracker

ParticleTracker was created by Mike Smith and James Downs and is offered as open source software which is free for you to use. If you use this software for any academic publications please cite this work using the following paper:

"ParticleTracker: a gui based particle tracking software" M.I. Smith, J.G. Downs, J. Open Source Software 6, 3611 (2021)

ParticleTracker also relies on two other libraries. Trackpy is used not only for the "trackpy" tracking method but also for the linking algorithm. You should therefore also cite this project (https://zenodo.org/record/4682814#.YVcuc9rMLIU).

OpenCV is used for the contours and hough circles tracking methods and the annotation (https://opencv.org/).

1.2 Installation and Getting Started

1.2.1 Installation

Even if you know no python you should have no problems setting this up with the instructions below and this should run on any system. It can be run with 2 lines of code! If you encounter problems please report them via the issues tracker on the github page

https://github.com/MikeSmithLabTeam/particletracker/issues

An additional feature of this project, is that we have tried to make extending it as easy as possible. It might mean as little as adding a few lines of python code into a preconfigured template.

The software should work on all operating systems (Windows, Linux, Mac). It has only been thoroughly tested on Windows 10 and Ubuntu Linux.

To install here is a step by step recommended guide to setting things up. In case you are coming to this new to python or new to programming we provide the steps in a lot of detail. If you are comfortable in python skip through!

- Download and install miniconda (https://docs.conda.io/en/latest/miniconda.html)
- Open a conda terminal:

On Windows type Anaconda at the windows search and then select "Anaconda Prompt" On Linux and Mac open a terminal.

- Create a conda environment by typing "conda create -n particle" where particle here is the name of the environment.
- Type "conda activate particle"
- conda install git
- conda install pyqt
- conda install pytables
- pip install git+https://github.com/MikeSmithLabTeam/particletracker

On Windows we sometimes ran into an error at this point concerning the hdflib that can be resolved by installing the Microsoft Visual Studio Build tools. Once you've installed them restart computer, open anaconda terminal, activate environment (step 2 above) Rerun the final command above. The build tools can be installed from here:

https://visualstudio.microsoft.com/visual-cpp-build-tools/

(Optional - nice way to work with the final data in a jupyter notebook) - conda install jupyterlab - conda install openpyxl

to upgrade use:

- pip install –upgrade git+https://github.com/MikeSmithLabTeam/particletracker

Once installed you will need some way to write simple code and execute it. The bare bones approach is to use a notepad and write the few lines of code as detailed in "Getting started". Save the file as eg. testscript.py and then from the conda command prompt navigate to the correct folder and run this script using "python testscript.py". Alternatively and far better in the long run is to install a python IDE and learn how to run code in the conda environment you've just created. Good IDEs include among others:

- PyCharm (https://www.jetbrains.com/pycharm/download/),
- VsCode (https://code.visualstudio.com/download)

Instructions abound on Google.

1.2.2 Verifying the installation

To verify that the installation is working correctly you should read the getting started section which explains how to launch the software. Once you have done this there are 5 example videos which also act as tutorials which should enable you to verify and test the core functionality of the software.

1.3 How do you start?

Every time you want to run the software you should open the anaconda command prompt with your conda environment activated. On Windows type Anaconda at the windows search and then select "Anaconda Prompt". On Linux and Mac open a terminal. Then type "conda activate particle" (assuming you followed our installation steps). Finally, navigate to the folder where you will store your python scripts: "cd pathtofolder".

You could also do this within an IDE just make sure your python interpreter is running from the conda environment.

After this, the place to start is with the track_gui() function which is contained in the ParticleTracker. To start the tracking gui you need to write a simple python script into a file and save it with a .py extension. The simple script looks like this:

```
from particletracker import track_gui
track_gui()
```

When you run this script a dialogue will open asking you to select a video file to perform tracking on. To save typing this in each time and perhaps select a custom set of settings you can however modify the code above to include a video filename and a settings filename. You need to include the full path to each file:

```
from particletracker import track_gui
track_gui(movie_filename="FullPathToMovie.mp4", settings_filename="FullPathToSettings.
↪param")
```

1.3.1 Building a tracking project

The first thing to do is to read / watch the overview to give you some orientation and then follow through the different example cases where we explain how to use many of the different features. This will take you from some very simple use cases to building some more complicated projects. For most people's needs this will be sufficient. To follow these through you will need to download the testdata folder. This is available from the toplevel of the github page (https://github.com/MikeSmithLabTeam/particletracker). Inside this folder you will find several example videos and some .param settings files. Each example has its own page in these docs which will walk you through from start to finish.

- *Overview*
- Tutorial1 - Eye Tracking
- Tutorial2 - Diffusing Colloids
- Tutorial3 - Swelling Hydrogels
- Tutorial4 - Bacteria
- Tutorial5 - Birefringent Discs
- Tutorial7 - Tips and Tricks

1.3.2 Working with the output data

If you are doing a really small project (and absolutely have to avoid programming!) it is possible to export data as an excel file. Please bear in mind though that this is pretty clunky and that as soon as your projects increase in file size ie number of frames x number of objects tracked etc this becomes pretty much unworkable.

A better way to work is to use a Jupyter notebook to look at the data. The installation for python set this up. Simply open the anaconda command prompt and activate the "particle" environment. To do this you type "conda activate particle". At the command prompt you then need to navigate to the testdata folder. Type "cd pathtotestdata". Finally type "jupyter notebook". This will open the server from where you can open the data_example.ipynb file. This jupyter notebook helps explain the format of the output data and shows you how to manipulate it to extract the aggregated data you probably want.

- Tutorial6 - Working with the final data in a Jupyter Notebook

1.3.3 Other tips and Tricks

- Tutorial7 - Assorted tips and tricks to help you using ParticleTracker

1.4 Overview of tracking projects

In this overview we explain the basic principles.

1.4.1 Understanding tracking

To build a tracking project there are several steps

1. Crop and mask the image to remove any unwanted bits of images that might produce spurious results.
2. Preprocess the images for tracking. Different methods require different things. Some need binary black and white images, others need grayscale. You can also perform a lot of operations to improve how easy it is to track the objects you're interested in.
3. Track - which means to locate the position of objects within a frame. There are 3 main methods that are currently implemented in this software:
    1. Opencv Hough Circles
    2. Trackpy an existing particle tracking library for tracking "blobs"
    3. Opencv Contour finding
4. Link the tracks together so that you know which particle is which in consecutive frames

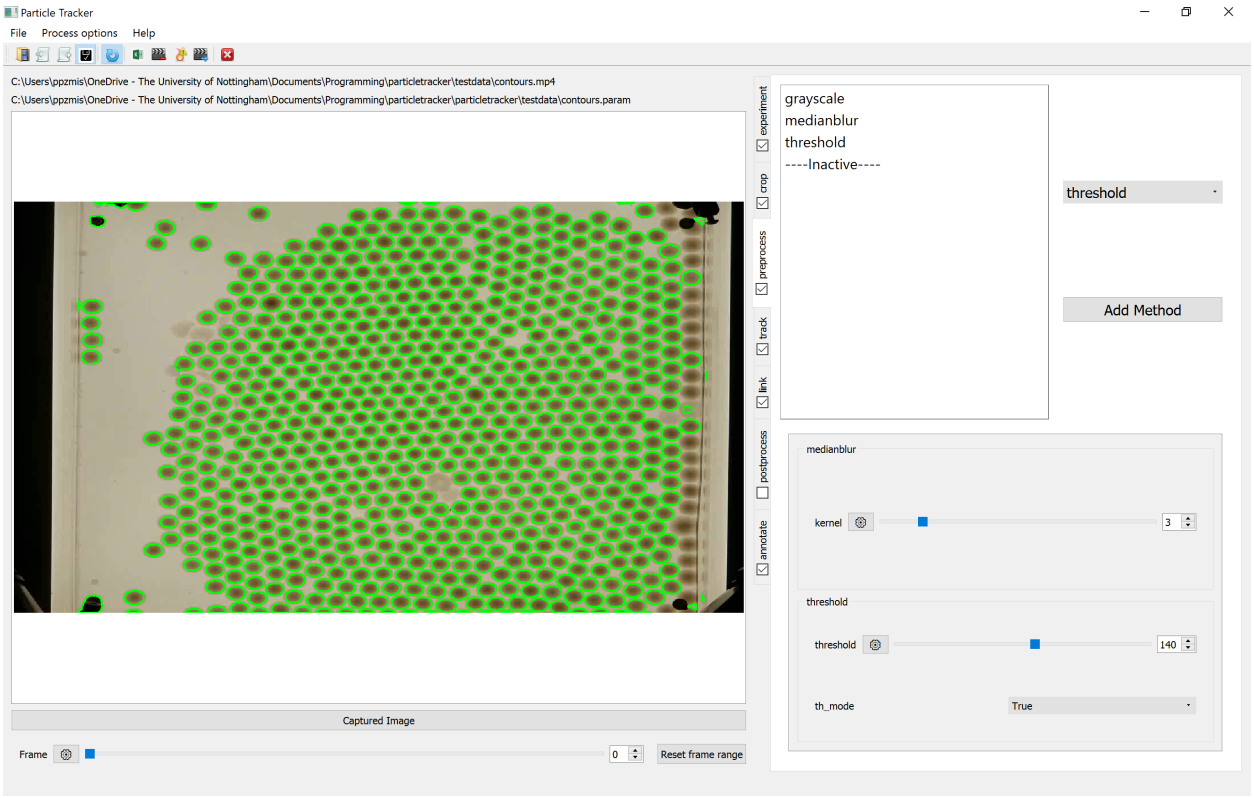
5. Postprocess. Calculate results based on the tracking. For example you might want to know which particles are neighbours or how fast particles are moving.
6. Annotate. This is useful for checking your tracking is working as expected but also to visualise values. eg you could colour code the value of an order parameter on each particle to see if they are clustered.

In general we use the track_gui to optimise all these different stages. However, once we have found the optimum set of conditions for tracking the settings can be saved to a .param file and used to setup all future tracking. Then if necessary you can automatically process batches of movies with the same settings.

1.4.2 Basic orientation for using the track_gui

Watch the video

The track_gui is divided into a few key areas: Toolbar / Menus, the Viewer, the Settings Panel



1.4.3 Toolbar / Menus

The icons and file menu enable you to: Open a new movie, load a settings .param file, save the current settings to a .param file, choose a file to act as the default settings, toggle the live updates, view data, export processed results to excel This option enables you to select if the results will also be published to an excel file.

1.4.4 The Viewer

The viewer displays a single frame from your movie with the annotations if you have added these. Since most tracking projects require you to preprocess the image you can also view the preprocessed image by toggling the button "Preprocessed Image". This is particularly useful in optimising the parameters before tracking. It is also useful to toggle between the preprocessed image and the tracked image with some form of annotation to assess whether the tracking could be improved by improving the preprocessing. There is also a slider with spinbox to allow you to scroll through the frames in the movie. The slider auto updates when released, the spinbox updates after you hit the enter key. You can also limit the range of frames being processed by selecting the settings wheel.

You can interact with the image:

- scroll wheel zooms on image
- right mouse button hold and drag zooms on selected region
- left mouse button hold and drag pans zoomed image
- double-click right mouse button resets zoom
- double-click left mouse button displays coordinates and image intensities at clicked point.

1.4.5 The Settings Panel

The settings panel consists of a series of tabs. Each tab connects to a different stage of the tracking process outlined above. Each tab has a checkbox which indicates whether the actions on this tab are active or not. Within each tab there are two sections: "Method Selectors" and "Parameter Adjustors"

1.4.6 The Method Selectors

Within each tab, the top half of the Settings Panel displays the methods. A method can be added by selecting from the drop down menu and clicking "Add Method". Initially this will appear at the bottom of the list below "—-inactive—-" place holder. The methods can be activated by dragging and dropping them (left mouse button) into the list above the "—-inactive—-" place holder. The methods are run in the order, from top to bottom, that they are listed in this dialogue. To remove a method temporarily move it below "—-inactive—-". To remove it more permanently you can right click on the method and it will disappear. In some cases you may want to apply the same method more than once with different parameters. This is not allowed for tracking methods but can be done for other processes. This will create a "methodname*1", "methodname*2" etc which can then be setup.

1.4.7 Parameter Adjustors

Each method has a set of parameters that need to be adjusted in order for it to work. These differ from method to method. These appear dynamically for all active methods in the bottom of the settings panel. There are several types of adjustor:

- Sliders with a spinbox. The limits of the sliders can be adjusted using the settings icon. This requires some care as we don't check that the new limits you put in are acceptable and hence there is a risk of crashing.
- Drop down menus with a fixed list of choices.
- Text boxes. Here the input may be quite varied. If you are unsure you can consult the reference for each method.

The crop section has a slightly different interface. One can manually enter the coordinates for a crop or mask functon but this is not recommended. Click the check box and then on the image click and hold the left mouse button and drag the shape and release to select the appropriate area. Afterwards the areas can be adjusted using the handles. Once finished

uncheck the check box to apply the crop or mask. This can be readjusted at any future point by simply rechecking the check box. Finally one can remove the crop / masks by clicking the reset button.

1.5 Tutorials

In these examples we illustrate how to build a number of different tracking projects.

- Tutorial1 - Eye Tracking
- Tutorial2 - Diffusing Colloids
- Tutorial3 - Swelling Hydrogels
- Tutorial4 - Bacteria
- Tutorial5 - Birefringent Discs
- Tutorial6 - Working with the final data in a Jupyter Notebook
- Tutorial7 - Assorted tips and tricks

1.6 Batch Processing files

Once you have got a working .param settings file using the gui, you can then batch process movies using the same settings. We can specify the movies to process using a pattern matching moviefilter. It accepts wildcard characters. * replaces any continuous set of characters. ? replaces a single character.

So if you have a folder with files:

```
[movieAB001.mp4, movieGOBBLEDYGOOK001.mp4,movieAB002.mp4,movieAB003.mp4,movieAB101.mp4,␣
↪movieAB001.avi]
```

a moviefilter = 'movie*00?.mp4' would process:

movieAB001, movieGOBBLEDYGOOK001.mp4, movieAB002.mp4 but not movieAB101.mp4, movieAB001.avi

To make use of this if you have a settings file saved with name. "mysettings.param" we just write the following code:

```
from particletracker import batchprocess
moviefilter = '/A/path/selector/movie*00?.mp4'
settings = '/full/path/to/settings.param'
batchprocess(moviefilter, settings,annotate=True)
```

You can optionally turn off the different steps so if you just want the data and don't want to produce annotated videos you could feed in the keyword argument annotate=False.

CHAPTER

TWO

EXTENDING THE SOFTWARE

2.1 Extending the functionality

The software is structured to help extension of the code be really simple. To extend any part of the software you need to do two things:

1. Add a function to the user_methods.py file in the top level of the particletracker project.
2. Add an appropriate entry to a .param file

To illustrate how to extend the software we use an example which is pretty much the same regardless of which part of the software you wish to extend. Lets say we have a new postprocessing method which we want to implement.

2.1.1 1. Add a function to user_methods.py

In the top level of the particletracker module is the user_methods.py. This contains template functions for the different sections. There are a lot of comments explaining details contained in these templates, however we have stripped out all the comments here to save space. The Docstrings in these examples explain what inputs and outputs your function needs to work. You then write whatever code is required.

```
def postprocessor_method_name(data, f_index=None, parameters=None, call_num=None):
    try:
        method_key = get_method_key('postprocessor_method_name', call_num=call_num)
        params = parameters['postprocess'][method_key]
        """
        Write the body of your code
        """
        return df
    except Exception as e:
        raise PPMethodNameError(e)
```

There is also a matching exception which you need to also copy. Make this exception name unique and match the raised Exception above:

```
class PPMethodNameError(PostprocessorError):
     """Implement this custom exception."""
     def __init__(self,e):
         super().__init__(e)
         self.error_msg = 'specific error message to show user in status bar'
         self.e=e
```

2.1.2 2. Add an entry to the dictionary

Open the file particletracker.general.param_file_creator. Inside this file there is a multiply nested dictionary that controls the behaviour of the particletracker.

Expand the "postprocess" dictionary. Add a new key to this dictionary with the same name as given to the function above and a value that is also a dictionary containing all the parameters needed.

```
postprocess = {postprocess_method:(smooth,),
               'smooth':{'column_name':'y',
                         'output_name':'y_smooth',
                         'span':[5,1,50,1],
                         'method':'default'
                        },
               'postprocessor_method_name':{'param1' : [startval, minval, maxval, step],
                                            'param2' : [value, ('value', 'value2',
↪'value3)],
                                            'param3' : (0,255,0),
                                            'param4' : 'simple text'
                                           }
                   }
```

The parameters are automatically assessed to decide what gui element to create.

- Param 1 will result in a slider with initial value startval and min, max and increment ("step") values as shown. If step is an integer eg 1 the slider will return integers. If step is a decimal eg 0.01 then the slider will return decimals. As a result if your value happens to be something like 1.0 you should write 1.0 and not 1.
- Param 2 results in a dropdown options box with the values value, value2, value3 to choose between. Within the brackets all these should be strings. However, the current value at the first position should be of the correct datatype.
- Param3 and 4 produce a text box which can take any value. The software recognises 'None', 'True' and 'False' as None, True and False.

To regenerate the settings file you can now call create_param_file(filename.param) to create a new file which can be read into the gui. This function is also used to create default settings when no settings filename is supplied.

```
from particletracker.general.param_file_creator import create_param_file

settings_filename = 'path/to/new/settings_file.param'
create_param_file(settings_filename)
```

To access the variables in your new dictionary entry inside the new method you need to write

```
param1 = get_param_val(params['param1'])
param2 = get_param_val(params['param2'])
```

2.1.3 Important note about reinstallation / upgrading particletracker

It is worth backing up your user_methods.py and param_file_creator.py files. If you upgrade and reinstall these can be easily lost so keep separate copies and then copy them back into the correct locations. If you are using the miniconda environments suggested it can be a pain to find the correct locations. A useful tip for quickly finding the directory is to activate your conda environment and then type python at the command prompt. Then type "import particletracker" followed by "particletracker.__file__". This will print the root directory of your particletracker installation. user_methods.py is in this root folder and param_file_creator is contained in the "general" subfolder.

2.2 Understanding key files

Within the software we make use of several important files

- .param files
- .hdf5 files

2.2.1 .param files

.param files are a nested set of python dictionaries. They effectively describe all the settings for a particle tracking project. When run with no arguments the software creates a default.param on start up. Alternatively if you are using python you can create one:

```
from particletracker.general import param_file_creator
filename = 'path/to/file.param'
param_file_creator(filename)
```

The top level is a dictionary which has keys:

```
PARAMETERS = {  'experiment': experiment,
                'crop': crop,
                'preprocess':preprocess,
                'track':track,
                'link':link,
                'postprocess':postprocess,
                'annotate':annotate
             }
```

One key for each key step in the tracking process. The value for each key is another dictionary which specifies the settings for that stage.

```
preprocess = {'preprocess_method': ('grayscale','medianblur',),
              'grayscale':{},
              'threshold':{'threshold':[1,0,255,1],
              'th_mode':[1,0,1,1]},
              'adaptive_threshold':{'block_size': [29,1,300,2],
                                    'C': [-23, -30, 30, 1],
                                    'ad_mode': [0, 0, 1, 1]
                                   },
              }
```

Above is a slimmed down version of the preprocess dictionary but all dictionaries are structured in the same way. The top line is always "dictionaryname"_method:(method1, method2,). Only the methods named in this tuple are actually

active methods eg grayscale and medianblur. Note below this there are many methods that are not listed here. These methods are not active but they are setup with default params so you can add them in.

For each method there is yet another dictionary. These contain the individual parameters for each method. These can be of several types.

- There are sliders with initial value startval and min, max and increment ("step") values as shown. If step is an integer eg 1 the slider will return integers. If step is a decimal eg 0.01 then the slider As a result if your value happens to be something like 1.0 you should write 1.0 and not 1.
- Dropdown options box with the values value, value2, value3 to choose between. Within the brackets all these should be strings. However value at the first position should be of the correct datatype.
- Text box which can take any value. The software recognises 'None', 'True' and 'False' as None, True and False.

These files can be saved and loaded directly within the gui to save sets of parameters appropriate for a particular experiment. Once a suitable .param file is created you can use this directly to batch process many files without needing to run the gui. When a video is processed a copy of the param file is automatically saved to the same folder with videoname.param

2.2.2 .hdf5 files

hdf5 files are for storing the data outputted from the tracking. These come in two types:

1. vidname_temp.hdf5
2. vidname.hdf5

The first is the output from a single frame analysed on the fly in the gui. This is what one is usually accessing. The second is the result from analysing all the frames either with the "process_part" or "process". When you check the "use_part" the software switches from using the vidname_temp.hdf5 file to the vidname.hdf5 to perform postprocessing / annotation. This is sometimes necessary. For instance to calculate a trajectory you must work with data from other frames. The vidname.hdf5 also represents the data file to which your tracked video data is stored and is the one you should access in the Jupyter Notebook to continue with the analysis.

CHAPTER

THREE

REFERENCE

3.1 Launching Tracking Methods

particletracker.__init__.**batchprocess**(*moviefilter*, *settings*, *crop=True*, *preprocess=True*, *track=True*, *link=True*, *postprocess=True*, *annotate=True*, *excel=False*)

batchprocess enables you to process all files specified with a filefilter using a single settings.param file

Parameters

- **filefilter** (*This is a full filename including filepath which may include wildcard characters*) –
- **paramfile** (*This is a full filename including filepath for a .param config file*) –
- **not.** (*Keyword arguments can be False or True. Determines whether this step is applied or*) –
- **None** (*Returns*) –
- **-------** –

particletracker.__init__.**track_gui**(*movie_filename=None*, *settings_filename=None*)

track_gui is a simple function that launches the main gui tracking window.

Parameters

- **movie** (*optional path to movie to process if not specified a dialogue window prompts user to navigate to file.*) –
- **settings** (*optional path to .param settings config file, if not set a default config file is automatically generated using create_param_file in general.param_file_creator*) –
- **None** (*Returns*) –
- **-------** –

3.2 Preprocessing

particletracker.preprocess.preprocessing_methods.**absolute_diff**(*frame*, *parameters=None*, *call_num=None*)

Calculates the absolute difference of pixels from a reference value

Notes

This function returns the magnitude of the difference in intensity of a pixel relative to a specified value. This is often useful in brightfield microscopy if you have objects slightly above and below the focal plane as one set will look darker than the background and the other set will look brighted than the background.

value
The value to take the absolute difference relative to

normalise
Stretch the intensity values to the full range 0-255, True or False

Parameters

- **frame** – This is must be a grayscale / single colour channel image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
grayscale image

particletracker.preprocess.preprocessing_methods.**adaptive_threshold**(*frame*, *parameters=None*, *call_num=None*)

Perform an adaptive threshold on a grayscale image

Notes

This applies OpenCVs adaptive threshold. This differs from global threshold in that for each pixel the cutoff threshold is defined based on a block of local pixels around it. This enables you to cope with gradual changes in illumination across the image etc.

block_size
Size of local block of pixels to calculate threshold on

C
The mean-c value see here: http://homepages.inf.ed.ac.uk/rbf/HIPR2/adpthrsh.htm

ad_mode
Inverts behaviour (True or False)

Parameters

- **frame** – This is must be a grayscale / single colour channel image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py

- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
binary image with 255 above threshold else 0.

particletracker.preprocess.preprocessing_methods.**blur**(*frame*, *parameters=None*, *call_num=None*)

Performs a gaussian blur on the image

Notes

This applies OpenCVs gaussian blur to the image (https://en.wikipedia.org/wiki/Gaussian_blur) Usually useful to apply before subtracting 2 images.

blur_kernel
single integer n specifying the size of kernel (n,n)

Parameters

- **frame** – This must be a grayscale / single colour channel image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
single colour channel image.

particletracker.preprocess.preprocessing_methods.**colour_channel**(*frame*, *parameters=None*, *call_num=None*)

This selects the specified colour channel of a colour image

colour
options are 'red', 'green', 'blue', We assume frame has (blue, green, red) format which is OpenCVs default.

Parameters

- **frame** – This must be a colour / single colour channel image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
Single colour channel image

particletracker.preprocess.preprocessing_methods.**dilation**(*frame*, *parameters=None*, *call_num=None*)

Dilate a binary image

This performs a dilation operation on a binary image. Dilation adds pixels to the edge of white regions according to the kernel and is useful for closing small holes or gaps. See an explanation - https://en.wikipedia.org/wiki/Dilation_(morphology)

dilation_kernel
single integer n specifying dimension of kernel (n,n)

iterations
how many times to apply the operation

Parameters

- **frame** – This must be a binary image (8 bit)
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
binary image

particletracker.preprocess.preprocessing_methods.**distance**(*frame*, *parameters=None*, *call_num=None*)

Perform a distance transform on a binary image

Notes

Implements the opencv distance transform. This transform operates on a binary image. For each chosen white pixel it calculates the distance to the nearest black pixel. This distance is the value of the chosen pixel. Thus if operating on a white circle the distance transform is a maximum at the middle and 1 at the perimeter.

See here for explanation : https://en.wikipedia.org/wiki/Distance_transform

Parameters

- **frame** – This must be a binary image (8 bit)
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
Grayscale image

particletracker.preprocess.preprocessing_methods.**erosion**(*frame*, *parameters=None*, *call_num=None*)

Perform an erosion operation on a binary image

Notes

This performs an erosion operation on a binary image. This means pixels are set to zero based on their connectivity with neighbours Useful for separating objects and removing small pepper noise.

See an explanation - https://en.wikipedia.org/wiki/Erosion_(morphology)

Parameters:

erosion_kernel : single integer n specifying dimension of kernel (n,n) iterations : how many times to apply the operation

Parameters

- **frame** – This must be a binary image (8 bit)
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
binary image with 255 above threshold else 0.

particletracker.preprocess.preprocessing_methods.**fill_holes**(*frame*, *parameters=None*, *call_num=None*)

Fills holes in a binary image.

Notes

This function uses a combination of flood fills to fill in enclosed holes in objects in a binary image.

Parameters

- **frame** – This is must be a binary image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
binary image

particletracker.preprocess.preprocessing_methods.**gamma**(*image*, *parameters=None*, *call_num=None*)

Apply look up table to image with power gamma

Notes

This generates a lookup table which maps the values 0-255 to 0-255 however not in a linear way. The mapping follows a power law with exponent gamma/100.0.

gamma
single float can be positive or negative. The true value applied is the displayed value / 100.

Parameters

- **frame** – This is must be a grayscale / single colour channel image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
grayscale image

particletracker.preprocess.preprocessing_methods.**grayscale**(*frame*, *parameters=None*, *call_num=None*)

This converts a colour image to a grayscale image

Parameters

- **frame** – This should be a colour image though won't error if given grayscale
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
grayscale image

particletracker.preprocess.preprocessing_methods.**invert**(*frame*, *parameters=None*, *call_num=None*)

Invert image

Notes

This inverts the supplied image. It will work with any kind of image (colour, grayscale, binary). The result for an 8bit image at each pixel is just 255 - currentvalue.

Parameters

- **frame** – will receive any type of image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
same as input image

particletracker.preprocess.preprocessing_methods.**medianblur**(*frame*, *parameters=None*, *call_num=None*)

Performs a medianblur on the image.

Notes

Setting each pixel to median value in the area specified by the kernel.

kernel
An integer value n that specifies kernel shape (n,n)

Parameters

- **frame** – This is must be a grayscale / single colour channel image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
grayscale image

particletracker.preprocess.preprocessing_methods.**subtract_bkg**(*frame*, *parameters=None*, *call_num=None*)

Subtract a background

Notes

This function will subtract a background from the image. It has several options: mean will subtract the average value from the image. img will subtract a preprepared background img from the img. Before subtracting the background image it is blurred according to the settings.

N.B. You must apply either a grayscale or color_channel method before the subtract_bkg method. The software subtracts the mean image value, grayscale or color_channel version of the background image which you select from the current image.

subtract_bkg_type
Type of background substraction to be performed. Options are are 'mean' or 'grayscale','red','green','blue'.

subtract_bkg_filename
filename of background image. If None it will look for a file named moviefilename_bkgimg.png. Otherwise it looks for the filename specified. The filename is assumed to be in the same directory as the movie. Alternatively specify the full path to the file.

subtract_bkg_blur_kernel
An integer n specifying the kernel size (n,n) to be used in blurring bkg image

subtract_bkg_invert
Subtract bkg from image or image from background.

subtract_bkg_norm
Stretch range of outputted intensities on resultant image to fill 0-255 - True or False

Parameters

- **frame** – This must be a grayscale / single colour channel image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
grayscale image

particletracker.preprocess.preprocessing_methods.**threshold**(*frame*, *parameters=None*, *call_num=None*)

Apply a global threshold

This applies OpenCVs threshold. This sets pixels to 255 or 0 depending on whether they are above or below the given value.

threshold
Threshold value to determine whether pixels are black or white

th_mode
True or False to specify whether above threshold is white or black.

Parameters

- **frame** – This is must be a grayscale / single colour channel image
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
grayscale image

3.3 Tracking

particletracker.track.tracking_methods.**contours**(*pp_frame*, *frame*, *parameters=None*)

Implementation of OpenCVs contours.

Notes

To use contours you must have preprocessed the image to produce a black and white binary image with separated object. Contours stores: the centroid x, y, area enclosed by contour, the bounding rectangle (not rotated) which is used with contour to generate mask so that you can extract pixels from original image and perform some analysis.

area_min
Minimum contour area to store object

area_max
Maximum contour area to store object

aspect_min
Minimum contour aspect ratio to store object

aspect_max
Maximum contour aspect ratio to store object

get_intensities
If not False results in the software extracting a region around each particle. Pixels outside the contour are masked. The remaining particle image is processed using get_intensities method. Select the method by writing its name in the get_intensities box.

Parameters

- **ppframe** – The preprocessed frame upon which tracking is to be performed.
- **frame** – The unprocessed frame on which get_intensities is run.
- **parameters** – Nested dictionary specifying the tracking parameters

Return type
Dataframe containing data from a single frame

particletracker.track.tracking_methods.**hough**(*ppframe*, *frame*, *params=None*)

Performs the opencv hough circles transform to locate circles in an image.

Notes

This method uses the opencv hough circles algorithm to look for circles in an image. It works well provided you constrain the radii searched to reasonably tight range. It is particularly good for tightly bunched large particles. To estimate the appropriate range of radii double left click on the image will give you a coordinate or you can use the circular crop tool to start off with about the right values. Set min dist that the centre of two circles can approach (a bit less than diameter). You then need to use P1 and P2 which are different gradient terms associated with the image. P1 is usually bigger than P2. Annotation with circles will automatically pick up the radii from the tracking so can be used to help get the settings right.

min_dist
minimum distance in pixels between two particles

min_rad
minimum radius of particles in pixels

max_rad
maximum radius of particles in pixels

p1
Control parameter

p2
Control parameter

remove_masked
Some circles have centres under the masked region. Selecting true removes this

get_intensities
If not False results in the software extracting a circular region around each particle of radius set by tracking and running a method in intensity_methods. Select the method by writing its name in the get_intensities box.

Parameters

- **ppframe** – The preprocessed frame upon which tracking is to be performed.
- **frame** – The unprocessed frame on which get_intensities is run.
- **parameters** – Nested dictionary specifying the tracking parameters

Return type
Dataframe containing data from a single frame

particletracker.track.tracking_methods.**trackpy**(*ppframe*, *frame*, *params=None*)
Trackpy implementation

Notes

This method uses the trackpy python library which can be found here: http://soft-matter.github.io/trackpy/v0.5.0 If you use this method in a research publication be sure to cite according to the details given here: http://soft-matter.github.io/trackpy/v0.5.0/generated/trackpy.locate.html

using get_intensities will seriously slow down the processing so optimise everything else first.

Parameters

- **information** (*First five parameters expose trackpy options. For more*) –
- **http** (*see*) –

- **diameter** – An estimate of the objects to be tracked feature size in pixels
- **minmass** – The minimum integrated brightness.
- **percentile** – Features must have a peak brighter than pixels in this percentile. This helps eliminate spurious peaks.
- **invert** – Set True if looking for dark objects on bright background
- **max_iterations** – max number of loops to refine the center of mass, default 10
- **get_intensities** – If not False results in the software extracting a circular region around each particle of radius set by intensity radius and running a method in intensity_methods. Select the method by writing its name in the get_intensities box.
- **intensity_radius** – The radius of the extracted intensity around each particle centre, see get_intensities.
- **show_output'** – print tracked data to terminal window.

x
x location of particle

y
y location of particle

mass
total integrated brightness of the blob

size
radius of gyration of its Gaussian-like profile

ecc
eccentricity

signal
?!

raw_mass
total integrated brightness in raw_image

Parameters

- **ppframe** – The preprocessed frame upon which tracking is to be performed.
- **frame** – The unprocessed frame on which get_intensities is run.
- **parameters** – Nested dictionary specifying the tracking parameters

Return type
Dataframe containing data from a single frame

3.4 Postprocessing

particletracker.postprocess.postprocessing_methods.**absolute**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Returns new column with absolute value of input column

Parameters

- **column_name** (*name of column containing input values*) –
- **df** – The dataframe for all data
- **f_index** – Integer for the frame in twhich calculations need to be made
- **parameters** – Nested dict object
- **call_num** –

Returns

- *df with additional column containing absolute value of input_column.*
- *New column is named "column_name" + "_abs"*

particletracker.postprocess.postprocessing_methods.**add_frame_data**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Add frame data allows you to manually add a new column of df to the dfframe.

Notes

This is done by creating a .csv file and reading it in within the gui. The file should have one column with the data for each frame listed on the correct line.

Parameters

- **data_filename** – filename with extension for the df to be loaded. Assumes file is in same directory as video
- **new_column_name** – Name for column to which data is to be imported.
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**angle**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Angle calculates the angle specified by two components.

Notes

Usually angle is used following calculating the difference along x and y trajectories. It assumes you want to calculate from x_column as dx and y_column as dy it uses tan2 so that -dx and +dy give a different result to +dx and -dy Angles are output in radians or degrees given by parameters['angle']['units']

Parameters

- **x_column** – x component for calculating angle
- **y_column** – y component for calculating angle
- **output_name** – New column name to store angle df
- **units** – 'degrees' or 'radians'
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**audio_frequency**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Decodes the audio frequency in our videos. We use this to encode information about the acceleration being applied to a video directly into the audio channel. This enables us to get the info back out

Parameters

- **([type])** (*df*) –
- **([type]** (*call_num*) –
- **optional)** (*[description]. Defaults to None.*) –
- **([type]** –
- **optional)** –
- **([type]** –
- **optional)** –

Returns
[type]

Return type
[description]

particletracker.postprocess.postprocessing_methods.**classify**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Classifies particles based on values in a particular column

Notes

Takes a column of data and classifies whether its values are within the specified range. If it is a True is put next to that particle in that frame in a new classifier column. This can be used to select subsets of particles for later operations.

Parameters

- **column_name** – input data column
- **output_name** – column name for classification (True or False)
- **lower_threshold** – min value to belong to classifier
- **upper_threshold** – max value to belong to classifier
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**contour_boxes**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Contour boxes calculates the rotated minimum area bounding box

Notes

This method is designed to work with contours. It calculates the minimum rotated bounding rectangle that contains the contour. This is useful for calculating the orientation of shapes.

'box_cx' - Centre of mass x coord of calculated box 'box_cy' - Centre of mass y coord of calculated box 'box_angle' - the angle of the long axis of the box relative to the x axis 'box_length'- Long dimension of box 'box_width' - Short dimension of box 'box_area' - Area of box

All values in units of pixels.

Parameters

- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**difference**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Difference of a particles values on user selected column.

Notes

Returns the difference of a particle's values on a particular column at span separation in frames to a new column. Please be aware this is the difference between current frame and frame - span for each particle.

Parameters

- **column_name** – Input column name
- **output_name** – Column name for median data
- **span** – number of frames over which to calculate rolling median
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**hexatic_order**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Calculates the hexatic order parameter of each particle. Neighbours are calculated using the Delaunay network with a cutoff distance defined by "cutoff" parameter.

Parameters

- **cutoff** – Distance threshold for calculation of neighbors
- **df** – The dataframe for all data
- **f_index** – Integer for the frame in twhich calculations need to be made
- **parameters** – Nested dict object
- **call_num** –

Return type
df with additional column

particletracker.postprocess.postprocessing_methods.**logic_AND**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Applys a logical and operation to two columns of boolean values.

column_name
input data column

column_name2
input data column

output_name
column name for the result

Parameters

- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**logic_NOT**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Apply a logical not operation to a column of boolean values.

Parameters

- **column_name** – input data column
- **column_name2** – input data column
- **output_name** – column name for the result
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**logic_OR**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Apply a logical or operation to two columns of boolean values.

Parameters

- **column_name** – input data column
- **column_name2** – input data column
- **output_name** – column name for the result
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**magnitude**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Calculates the magnitude of 2 input columns (x^2 + y^2)^0.5 = r

Parameters

- **column_name** (*Second column*) –
- **column_name** –
- **output_name** (*Column name for magnitude df*) –
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**mean**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Rolling mean of a particles values.

Notes

Returns the rolling mean of a particle's values to a new column. Useful to reduce fluctuations or tracking inaccuracies. The value of the mean is placed at the centre of the rolling window. i.e [2,4,6,8,4] with window 3 would result in [NaN, 4, 6, 6, Nan].

Parameters

- **column_name** – Input column name
- **output_name** – Column name for mean data
- **span** – number of frames over which to calculate rolling mean
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**median**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Median of a particles values.

Notes

Returns the median of a particle's values to a new column. Useful before classification to answer to which group a particle's properties usually belong. The value of the median is placed at the centre of the rolling window. i.e [2,4,4,8,4] with window 3 would result in [NaN, 4, 4, 4, Nan].

Parameters

- **column_name** – Input column name
- **output_name** – Column name for median data
- **span** – number of frames over which to calculate rolling median
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**neighbours**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Find the nearest neighbours of a particle

Notes

Neighbours uses two different methods to find the nearest neighbours: a kdtree (https://en.wikipedia.org/wiki/K-d_tree) or a delaunay method (https://en.wikipedia.org/wiki/Delaunay_triangulation) to locate the neighbours of particles in a particular frame. It returns the indices of the particles found to be neighbours in a list. You can also select a cutoff distance above which two particles are no longer considered to be neighbours. To visualise the result you can use "networks" in the annotation section.

Parameters

- **method** – 'delaunay' or 'kdtree'
- **neighbours** – max number of neighbours to find. This is only relevant for the kdtree.
- **cutoff** – distance in pixels beyond which particles are no longer considered neighbours

'neighbours' - A list of particle indices which are neighbours

Parameters

- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**rate**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Rate of change of a particle property with frame

Notes

Rate function takes an input column and calculates the rate of change of the quantity. Nans are inserted at end and beginning of particle trajectories where calc is not possible. The rate is calculated from diff between current frame and frame - span.

Parameters

- **column_name** – Input column names
- **output_name** – Output column name
- **fps** – numerical value indicating the number of frames per second
- **span** – number of frames over which to calculate rolling difference
- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

particletracker.postprocess.postprocessing_methods.**real_imag**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Extracts the real, imaginary, complex magnitude and complex angle from a complex number and puts them in new columns. Mainly useful for subsequent annotation with dynamic colour map.

Parameters

- **column_name** (*name of column containing complex values*) –
- **df** – The dataframe for all data
- **f_index** – Integer for the frame in twhich calculations need to be made
- **parameters** – Nested dict object
- **call_num** –

Returns

- *df with 3 additional columns containing real, imaginary and complex angle*
- *New columns are called "column_name" + "_Re" or "_Im" or "_Ang"*

particletracker.postprocess.postprocessing_methods.**voronoi**(*df*, *f_index=None*, *parameters=None*, *call_num=None*)

Calculate the voronoi network of particle.

Notes

The voronoi network is explained here: https://en.wikipedia.org/wiki/Voronoi_diagram This function also calculates the associated area of the voronoi cells.To visualise the result you can use "voronoi" in the annotation section.

'voronoi' - The voronoi coordinates that surround a particle 'voronoi_area' - The area of the voronoi cell associated with a particle

Parameters

- **df** – The dataframe in which all data is stored
- **f_index** – Integer specifying the frame for which calculations need to be made.
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Return type
updated dataframe including new column

3.5 Annotation Methods

particletracker.annotate.annotation_methods.**boxes**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Boxes places a rotated rectangle on the image that encloses the contours of specified particles.

Notes

This method requires you to have used contours for the tracking and run boxes in postprocessing.

Parameters

- **cmap_type** – Options are 'static' or 'dynamic'
- **cmap_column** – Name of column containing data to specify colour in dynamic mode,
- **cmap_max** – Specifies max data value for colour map in dynamic mode
- **cmap_scale** – Scale factor for colour map
- **colour** – Colour to be used for static cmap_type (B,G,R) values from 0-255
- **classifier_column** – None selects all particles, column name of classifier values to specify subset of particles
- **classifier** – The value in the classifier column which applies to subset (True or False)
- **thickness** – Thickness of box. -1 fills the box in
- **frame** – This is the unmodified frame of the input movie
- **data** – This is the dataframe that stores all the tracked data
- **f** – frame index
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)

- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Returns
annotated frame

Return type
np.ndarray

particletracker.annotate.annotation_methods.**circles**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Circles places a ring on every specified particle

Parameters

- **xdata_column** – Name of column to use for x coordinates
- **ydata_column** – Name of column to use for y coordinates
- **rad_from_data** – Specify radius manually: False or use measured rad: True. Only works for Hough transform.
- **radius** – If rad_from_data = False this specifies the radius of circle
- **cmap_type** – Options are static or dynamic
- **cmap_column** – Name of column containing data to specify colour in dynamic mode,#for dynamic
- **cmap_max** – Specifies max data value for colour map in dynamic mode
- **cmap_scale** – Scale factor for colour map
- **colour** – Colour to be used for static cmap_type (B,G,R) values from 0-255
- **classifier_column** – None - selects all particles, column name of classifier values to apply to subset of particles
- **classifier** – The value in the classifier column to apply colour map to (True or False)
- **thickness** – Thickness of circle. -1 fills the circle in solidly.
- **frame** (*np.ndarray*) – This is the unmodified frame of the input movie
- **data** (*pandas dataframe*) – This is the dataframe that stores all the tracked data
- **f** (*int*) – frame index
- **parameters** (*dict*) – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** (*int or None*) – Usually None but if multiple calls are made modifies method name with get_method_key

Returns
annotated frame

Return type
np.ndarray

particletracker.annotate.annotation_methods.**contours**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Contours draws the tracked contour returned from Contours tracking method onto the image.

Notes

Requires the contours tracking method.

Parameters

- **cmap_type** – Options are static or dynamic
- **cmap_column** – Name of column containing data to specify colour in dynamic mode
- **cmap_max** – Specifies max data value for colour map in dynamic mode
- **cmap_scale** – Scale factor for colour map
- **colour** – Colour to be used for static cmap_type (B,G,R) values from 0-255
- **classifier_column** – None - selects all particles, column name of classifier values to apply to subset of particles
- **classifier** – The value in the classifier column to apply colour map to (True or False).
- **thickness** – Thickness of contour. -1 will fill in contour
- **frame** – This is the unmodified frame of the input movie
- **data** – This is the dataframe that stores all the tracked data
- **f** – frame index
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Returns
annotated frame

Return type
np.ndarray

`particletracker.annotate.annotation_methods.`**networks**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Networks draws a network of lines between particles

Notes

The network must previously have been calculated in postprocessing. See "neighbours" in postprocessing.

Parameters

- **cmap_type** – Options are static or dynamic
- **cmap_column** – Name of column containing data to specify colour in dynamic mode,#for dynamic
- **cmap_max** – Specifies max data value for colour map in dynamic mode
- **cmap_scale** – Scale factor for colour map
- **colour** – Colour to be used for static cmap_type (B,G,R) values from 0-255
- **classifier_column** – None - selects all particles, column name of classifier values to apply to subset of particles

- **classifier** – The value in the classifier column to apply colour map to.
- **thickness** – Thickness of network lines
- **frame** (*np.ndarray*) – This is the unmodified frame of the input movie
- **data** (*pandas dataframe*) – This is the dataframe that stores all the tracked data
- **f** (*int*) – frame index
- **parameters** (*dict*) – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** (*int or None*) – Usually None but if multiple calls are made modifies method name with get_method_key

Returns
annotated frame

Return type
np.ndarray

particletracker.annotate.annotation_methods.**particle_labels**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Annotates image with particle info from one column. The most common use is to indicate the particle index but any column of data could be used.

Notes

For particle ids to be meaningful, you must have already run 'processed part' with linking selected. This is particularly useful if you want to extract information about specific particles. Annotate their ids to identify the reference id of the one you are interested in and then you can pull the subset of processed data out. See examples in Jupyter notebook. Any particle level data can however be displayed.

Parameters

- **values_column** – Name of column containing particle info to be displayed.
- **position** – Coordinates of upper left corner of text
- **font_colour** – Colour of font specified in (B,G,R) format where values are integers from 0-255
- **font_size** – Size of font
- **font_thickness** – Thickness of font
- **frame** – This is the unmodified frame of the input movie
- **data** – This is the dataframe that stores all the tracked data
- **f** – frame index
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Returns
annotated frame

Return type
np.ndarray

particletracker.annotate.annotation_methods.**text_label**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Text labels place a static label on an image at specific location.

Notes

This function is for adding titles or info that doesn't change

Parameters

- **text** – Text to be displayed
- **position** – Coordinates of upper left corner of text
- **font_colour** – Colour of font specified in (B,G,R) format where values are integers from 0-255
- **font_size** – Size of font
- **font_thickness** – Thickness of font
- **frame** – This is the unmodified frame of the input movie
- **data** – This is the dataframe that stores all the tracked data
- **f** – frame index
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Returns
annotated frame

Return type
np.ndarray

particletracker.annotate.annotation_methods.**trajectories**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Trajectories draws the historical track of each particle onto an image.

Notes

Requires data from other frames hence you must have previously processed the video and then toggled use_part_processed button.

Parameters

- **x_column** – column name of x coordinates of particle,
- **y_column** – column name of y coordinates of particle,
- **traj_length** – number of historical frames to include in each trajectory.
- **classifier_column** – None - selects all particles, column name of classifier values to apply to subset of particles
- **classifier** – The value in the classifier column to apply colour map to (True or False).
- **cmap_type** – Options are static or dynamic

- **cmap_column** – Name of column containing data to specify colour in dynamic mode,
- **cmap_max** – Specifies max data value for colour map in dynamic mode
- **cmap_min** – Specifies min data value for colour map in dynamic mode
- **colour** – Colour to be used for static cmap_type (B,G,R) values from 0-255
- **thickness** – Thickness of line.
- **frame** (*np.ndarray*) – This is the unmodified frame of the input movie
- **data** (*pandas dataframe*) – This is the dataframe that stores all the tracked data
- **f** (*int*) – frame index
- **parameters** (*dict*) – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** (*int or None*) – Usually None but if multiple calls are made modifies method name with get_method_key

Returns
annotated frame

Return type
np.ndarray

particletracker.annotate.annotation_methods.**var_label**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Var labels puts text on an image at specific location for each frame. The value displayed in that frame is mapped to a column in the dataframe. The values next to each frame should all be the same for that column. Use for example to specify the temperature.

Notes

This function is for adding data specific to a single frame. For example you could indicate the temperature of the sample or time. The data for a given frame should be stored in a particular column specified in the 'var_column' section of the dictionary.

Parameters

- **var_column** – Column name containing the info to be displayed on each frame
- **position** – Coordinates of upper left corner of text
- **font_colour** – Colour of font specified in (B,G,R) format where values are integers from 0-255
- **font_size** – Size of font
- **font_thickness** – Thickness of font
- **frame** – This is the unmodified frame of the input movie
- **data** – This is the dataframe that stores all the tracked data
- **f** – frame index
- **parameters** – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** – Usually None but if multiple calls are made modifies method name with get_method_key

Returns
annotated frame

Return type
np.ndarray

particletracker.annotate.annotation_methods.**vectors**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Vectors draw info onto images in the form of arrows.

Notes

Vectors draws an arrow starting at each particle with a length and direction specified by 2 components. The magnitude of the vector can be scaled to be appropriate.

Parameters

- **dx_column** – Column name of x component of vector, defaults to 'x'
- **dy_column** – Column name of y component of vector, defaults to 'y'
- **vector_scale** – scaling between vector data and length of displayed line
- **classifier_column** – None - selects all particles, column name of classifier values to apply to subset of particles
- **classifier** – The value in the classifier column to apply colour map to.
- **cmap_type** – Options are static or dynamic
- **cmap_column** – Name of column containing data to specify colour in dynamic mode,
- **cmap_max** – Specifies max data value for colour map in dynamic mode
- **cmap_min** – Specifies min data value for colour map in dynamic mode
- **colour** – Colour to be used for static cmap_type (B,G,R) values from 0-255
- **line_type** – OpenCV parameter can be -1, 4, 8, 16
- **thickness** – Thickness of line. Defaults to 2
- **tip_length** – Controls length of arrow head
- **frame** (*np.ndarray*) – This is the unmodified frame of the input movie
- **data** (*pandas dataframe*) – This is the dataframe that stores all the tracked data
- **f** (*int*) – frame index
- **parameters** (*dict*) – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** (*int or None*) – Usually None but if multiple calls are made modifies method name with get_method_key

Returns
annotated frame

Return type
np.ndarray

particletracker.annotate.annotation_methods.**voronoi**(*frame*, *data*, *f*, *parameters=None*, *call_num=None*)

Voronoi draws the voronoi network that surrounds each particle

Notes

The voronoi cells must previously have been calculated in postprocessing. See "voronoi" in postprocessing.

Parameters

- **cmap_type** – Options are static or dynamic
- **cmap_column** – Name of column containing data to specify colour in dynamic mode,#for dynamic
- **cmap_max** – Specifies max data value for colour map in dynamic mode
- **cmap_min** – Specifies min data value for colour map in dynamic mode
- **cmap_scale** – Scale factor for colour map
- **colour** – Colour to be used for static cmap_type (B,G,R) values from 0-255
- **classifier_column** – None - selects all particles, column name of classifier values to apply to subset of particles
- **classifier** – The value in the classifier column to apply colour map to.
- **thickness** – Thickness of network lines
- **frame** (*np.ndarray*) – This is the unmodified frame of the input movie
- **data** (*pandas dataframe*) – This is the dataframe that stores all the tracked data
- **f** (*int*) – frame index
- **parameters** (*dict*) – Nested dictionary like object (same as .param files or output from general.param_file_creator.py)
- **call_num** (*int or None*) – Usually None but if multiple calls are made modifies method name with get_method_key

Returns

annotated frame

Return type

np.ndarray

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

I

L

M

N

P

R

S

T

V