gbeflow Documentation

Release 0.0.0

Morgan Schwartz

Dec 13, 2018

Contents:

CHAPTER 1

README: Germband Extension

Efforts to track the rate of extension of the Drosophila germ band

1.1 Usage

1.2 Prerequisites

1.2.1 Anaconda

Anaconda is a distribution and package manager targeted for scientific computing. While Anaconda mostly focuses on python, it is capable with interfacing with other languages/packages as well. Several of the packages required for gbeflow have underlying C dependences that make independent installation difficult. Anaconda includes all of these packages and makes it simple to install any additional packages. gbeflow is written in Python 3, so we recommend installing the most up-to-date Anaconda installation for Python 3.

1.2.2 Matlab Setup

The optical flow algorithm which gbeflow relies on is written in Matlab (Vig et al. 2016). For more information on optical flow and the algorithm, checkout *Optical Flow*. For the purposes of installation and setup, all you need to know is that you need a local installation of Matlab on your computer to run steps involving the optical flow algorithm. gbeflow was developed using Matlab 2017b, but there are not any dependences known to this specific version. We will run Matlab scripts out of python so no Matlab knowledge is required. However, Matlab is currently only compatible with Python 3.6, so we will need to set up an environment to specifically run 3.6. Once we have the correct environment set up, we will install a matlab engine for python.

1.3 Setup

The code for gbeflow is hosted on Github. Users can clone the repository by running the following code in the command line

```
git clone https://github.com/msschwartz21/germband-extension.git
```

Alternatively, the current version of the repository can be downloaded as a zip file here.

1.3.1 Python Environment Setup

For simplicity, there is a script that enables you to setup a virtual environment in Anaconda with all the appropriate dependencies. This file should be downloaded during the installation steps below, but it is also available here. To setup the environment run the following commands in your terminal from the root of the germband-extension directory.

```
# Create python3.6 virtual environment
conda create -n python36 python=3.6 anaconda h5py bokeh tqdm numpydoc

# Activate new environment
conda activate python36

# Install remaining conda packages
conda install nodejs
conda install -c conda-forge av altair OpenPIV

# Install pip packages
pip install czifile tifffile sphinx-rtd-theme bebi103 nb_conda_kernels
```

This script will create an anaconda virtual environment named python36. You can activate the environment by running `conda activate python36`. When you are done with the environment run `conda deactivate` to return to the basic python environment.

Warning: `conda activate` doesn't appear to work in powershell, but it does work from the Anaconda Prompt and Command.

When we are working in Jupyter Notebooks, nb_conda_kernels will provide the option to launch the notebook from any available virtual environment including python36. Justin Bois has a great introduction to Jupyter notebooks available here.

1.3.2 Matlab Engine Installation

Matlab includes a python engine with its default installation. In order to install the engine as a python module, follow the instructions listed here. Make sure that the python 3.6 environment is active by running `conda activate python36`.

1.3.3 Troubleshooting installation problems

These notes are here to serve as a record for previously encountered problems, but undoubtably new problems will show up in the future. When I was working on the installation for the lab computer, Anaconda recommended that PATH variables not be set by Anaconda during the installation process. The result of this choice is that conda/python/etc. can be called from the Anaconda Prompt, but these commands are not available from any other terminal interface such

as Command or Powershell. I ended up reinstalling Anaconda and choosing to install with the PATH variables and it hasn't caused any problems to date.

Matlab is typically installed for all users, which means it requires administrator priveleges to make any changes to the directory. As a result, if we use a normal command prompt to try to run `python setup.py install` from within the Matlab directory, we are blocked from making changes since normal command interfaces do not have administrator priveleges. To get around this problem, Command or Anaconda Prompt can be launched with administrator priveleges by right clicking on the program to launch and selecting the "Run as Administrator" option. This administrator option should only be used when absolutely necessary, such as running the python installation for matlab.

1.3.4 gbeflow Installation

Now that we have a python 3.6 environment setup, we are ready to locally install gbeflow. From the terminal, run the following code to enter the python36 environment and install gbeflow. Begin by navigating to the root of the gbeflow directory and run the following from the command line.

```
$ conda activate python36
$ pip install -e .
$ conda deactivate
```

1.4 API

Documentation is available on Read the Docs.

1.5 License

gbeflow is licensed under the MIT License.

1.5.1 API

gbeflow

class gbeflow.**CziImport**(*fpath*, *summary=True*)

Bases: object

Defines a class to wrap the czifile object Identifies data contained in each dimension Helps user extract desired data from multidimensional array

__init__(*fpath*, *summary=True*)

Read in file using czifile

Parameters **fpath** (*str*) – Complete or relative file path to czi file

print_summary()

Prints a summary of data dimensions Assumes that the data is a standard brightfield timelapse collection, e.g. (?, roi, channel, time, z, x, y, ?)

squeeze_data()

Uses np.squeeze to reduce dimenions of data according to input preference

class gbeflow.**MaskEmbryo**(*points*)

Bases: object

Fit an ellipse to an embryo and calculate mask

__init__(*points*)

Calculate a first try ellipse using default parameters

Parameters **points** (*pd.DataFrame*) – Contains the columns x and y with 2 rows

Returns *self.ell, self.rell, self.fell*

calc_ellipse(*center_x*, *center_y*, *radius_x*, *radius_y*)

Calculate a parametrized ellipse based on input values

Parameters

- **center_x** (*float*) – Center point of the ellipse in x dimension
- **center_y** (*float*) – Center point of the ellipse in y dimension
- **radius_x** (*float*) – Radius of ellipse in x dimension
- **radius_y** (*float*) – Radius of ellipse in y dimension

Returns *Ellipse in a 400x2 array*

calc_rotation(*ell=None*, *df=None*)

Calculate angle of rotation and rotation matrix using -angle

Parameters

- **ell** (*np.array, optional*) – Ellipse array
- **df** (*pd.DataFrame, optional*) – Contains the columns x and y with 2 rows

Returns *self.rell*

calc_start_ell(*scale=1.5*, *yradius=300*, *df=None*)

Customize the fit of an ellipse to an embryo based on the selected endpoints

Parameters

- **scale** (*float, optional*) – Typically greater than 1 to extend the length between the two points beyond the ends of the embryo
- **yradius** (*int, optional*) – Y radius for initial ellipse, default=300
- **df** (*pd.DataFrame, optional*) – Contains the columns x and y with 2 rows

Returns **self.ell** (*array*) – Array of shape 400x2 that contains position of custom ellipse

contour_embryo(*img*, *init=None*, *sigma=3*)

Fit a contour to the embryo to separate the background

Parameters

- **img** (*2D np.array*) – 2D image from a single timepoint to mask
- **init** (*400x2 ellipse array, optional*) – Starting ellipse array that is bigger than the embryo
- **sigma** (*int, optional*) – Kernel size for the Gaussian smoothing step

Returns *Masked image where all background points = 0*

mask_image(*img*, *mask*)

Apply mask to image and return with background = 0

Parameters

- **img** (*2D np.array*) – 2D image from a single timepoint to mask
- **mask** (*2D np.array*) – 2D boolean array containing mask

shift_to_center(*rell=None*, *df=None*)
Shift ellipse that started at (0,0) to the center of the embryo

Parameters

- **rell** (*np.array, optional*) – Ellipse array
- **df** (*pd.DataFrame, optional*) – Contains the columns x and y with 2 rows

Returns *self.fell*

class gbeflow.**VectorField**(*name*)
Bases: object

Object to manage results and calculations from OpticalFlow.mat

See *__init__* for more information

__init__(*name*)
Initialize VectorField object by importing and transforming data

Parameters **name** (*str*) – String specifying the name passed to OpticalFlowOutput Can include complete or partial path to file

name
str – Based on the parameter name

df
pd.DataFrame – Dataframe of vectors produced by *tidy_vector_data*

tt
np.array – Meshgrid for t dimension

xx
np.array – Meshgrid for x dimension

yy
np.array – Meshgrid for y dimension

vx
np.array – X component of the velocity vector

vy
np.array – Y component of the velocity vector

xval
np.array – Sorted array of unique x values in xx meshgrid

yval
np.array – Sorted array of unique y values in yy meshgrid

tval
np.array – Sorted array of unique t values in tt meshgrid

starts
pd.DataFrame – Dataframe initialized to contain starter points

add_image_data(*impath*)
Imports a 3D (txy) dataset (czi or tiff) that matches the vector data

Parameters **impath** (*str*) – Complete or relative path to image file Accepts either tif or czi file types

img
np.array – 3D array of image data

calc_track (*x0*, *y0*, *dt*, *tmin=0*)
Calculate the trajectory of a single point through space and time

Parameters

- **x0** (*float*) – X position of the starting point
- **y0** (*float*) – Y position of the starting point
- **dt** (*float*) – Duration of time step
- **trange** (*list or np.array*) – Range of t values to iterate over for interpolation

Returns **track** (*np.array*) – Array of dimension number_t_steps x 2

calc_track_set (*starts*, *dt*, *name=''*, *timer=True*, *tmin=0*)
Calculate trajectories for a set of points using a constant dt

Parameters

- **starts** (*pd.DataFrame*) – Dataframe with columns x and y containing one point per row
- **dt** (*float*) – Duration of time step
- **name** (*str, optional*) – Default, '', encodes notes for a set of points
- **timer** (*boolean, optional*) – Default = True, activates tqdm progress timer

tracks
pd.DataFrame – Dataframe with columns x,y,t,name,track Contains trajectories based on points in *starts*

initialize_interpolation (*timer=True*)
Calculates interpolation of vx and vy for each timepoint Uses scipy.interpolate.RectBivariateSpline for optimal speed on meshgrid data

Parameters **timer** (*boolean, optional*) – Default = True, activates tqdm progress timer

Ldx
list – List of dx interpolation objects for each t

Ldy
list – List of dy interpolation objects for each t

interp_init
boolean – Set to True after completion of interpolation for loop

pick_start_points (*notebook_url='localhost:8888'*)
Launches interactive bokeh plot to record user clicks

Parameters **notebook_url** (*str, optional*) – Default 'localhost:8888', specifies jupyterlab url for interactive plotting

Returns *p* – Plotting object for bokeh plot

save_start_points (*p*)
Uses the *to_df* method of the plotting object generated by *pick_start_points* to generate dataframe of click points

Parameters **p** (*object*) – Generated by *pick_start_points* after clicks have been recorded

starts
pd.DataFrame – Appends new recorded clicks to *starts* dataframe

gbeflow.**calc_embryo_theta**(*line*)
Given a line fit to each embryo, calculate the angle of rotation to align the embryo in the horizontal axis

Parameters **line** (*pd.DataFrame*) – Dataframe returned by *calc_line()* containing columns dy and dx

Returns **line** (*pd.DataFrame*) – Input dataframe with an additional column theta in degrees

gbeflow.**calc_flow_path**(*xval*, *yval*, *vx*, *vy*, *x0*, *y0*, *dt*, *timer=True*)
Calculate the trajectory of a point through the vector field over time

Parameters

- **xval** (*np.array*) – A list of unique x values that define the meshgrid of xx
- **yval** (*np.array*) – A corresponding list of unique y values that define the meshgrid of yy
- **vx** (*np.array*) – Array of shape (time,len(xval),len(yval)) containing the x velocity component
- **vy** (*np.array*) – Array of shape (time,len(xval),len(yval)) containing y velocity component
- **dt** (*float*) – Duration of the time step between intervals
- **timer** (*boolean, optional*) – Default true uses the tqdm timer as an iterator

Returns *Array of shape (time,2) containing x and y position of trajectory over time*

gbeflow.**calc_line**(*points*)
Given two points calculate the line between the two points

Parameters **points** (*pd.DataFrame*) – Dataframe containing the columns x, y, and 'level_1' (from df.reset_index()) Each row should contain data for a single point

Returns **lines** (*pd.DataFrame*) – Dataframe with columns that specify an equation for a line

gbeflow.**imshow**(*img*, *figsize=(10, 8)*)
Show image using matplotlib and including colorbar

Parameters

- **img** (*np.array*) – 2D array to display
- **figsize** (*tuple, optional*) – Default = (10,8) Tuple passed to matplotlib to specify figure size

gbeflow.**load_avi_as_array**(*path*)
Use *av* module to load each frame from an avi movie into a numpy array

Parameters **path** (*str*) – Complete or relative path to avi movie file for import

Returns *np.array* – Array with dimensions frames,x,y

gbeflow.**make_track_movie**(*movie*, *df*, *c*, *name*)
Plots the trajectory of points over time on each frame of an existing movie or array

Parameters

- **movie** (*str*) – Complete or relative path to the movie file to plot on

- **df** (*pd.DataFrame*) – Dataframe of tracks minimally with columns x,y,t
- **c** (*str, color*) – Currently only supports single color assignments, but data specific assignments could be possible
- **name** (*str*) – Root of filename for output file, without filetype

Returns Saves a tif stack using path provided by *name*

gbeflow.**read_hyperstack**(*fpath*, *dataset='channel1'*, *tmax=166*)

Read in an hdf5 hyperstack where each timepoint is a group containing two channels which are saved as datapoints

> **Warning:** Doesn't work on files that were written with *write_hyperstack()*. Tiff files are probably better.

Parameters

- **fpath** (*str*) – Relative or absolute path to data file
- **dataset** (*str, optional*) – Default = 'channel1' Specifies the key to select a dataset from the h5 file
- **tmax** (*int, optional*) – Default = 166 Number of expected time frames

Returns *np.array* – Array with h5 data in dimensions txy

gbeflow.**reshape_vector_data**(*df*)

Convert dataframe structure into a set of meshgrid arrays

Parameters **df** (*pd.DataFrame*) – Dataframe with columns x,y,frame,vx,vy

Returns *tt,xx,yy,vx,vy* – Set of arrays of shape (len(T),len(X),len(Y))

gbeflow.**tidy_vector_data**(*name*)

Tidys csv files exported from matlab OpticalFlow

Parameters **name** (*str*) – String specifying the name passed to OpticalFlowOutput Can include complete or partial path to file

Returns *pd.DataFrame* – Dataframe containing the following columns: frame, x, y, vx, vy

gbeflow.**write_hyperstack**(*hst*, *fpath*, *dataset='channel1'*)

Write an h5 file in the same format as was read in

Parameters

- **hst** (*np.array*) – Array with dimensions txy
- **fpath** (*str*) – Complete path to output file with '.h5'
- **dataset** (*str, optional*) – Specifies name of the dataset in h5 file

1.5.2 Data Preprocessing

Import raw microscopy data

Data can be imported from czi and tiff files. The *gbeflow.CziImport* wraps the czifile module published by Christoph Gohlke to read the propriatory zeiss file format. The underlying datastructure of czi's varies based on the collection parameters, so the *gbeflow.CziImport* may not work reliabily.

```
import gbeflow

# Load czi file from relative/absolute path
czi = gbeflow.CziImport(filepath)

# Raw array of data is available as an attribute
raw = czi.raw_im

# Data squeezed to minimum dimensions also available
data = czi.data
```

Alternativly, data can be read from tiff files saved using Fiji. The module tifffile makes this by far the easiest approach with `tifffile.imread` and `tifffile.imsave`.

```
import tifffile

# Import tiff file from relative or absolute path
img = tifffile.imread(filepath)

# Save array to new tiff file
tifffile.imsave(outpath, data=img)
```

Embryo alignment

During imaging, embryos are not expected to be aligned in any particular orientation in the XY plane. While we will accept only embryos that have an approximate lateral mounting, we need to correct XY positioning in post-processing. In the notebook rotate_embryo, a workflow is proposed that accepts user inputs in order to guide the alignment process. The user input functions rely on bebi103 which is a package written by Justin Bois for the BE/Bi103 course that is still under active development. Below is an example of what processing a single sample might look like.

```
import os
import glob
import tqdm

import numpy as np
import pandas as pd
import scipy.ndimage

import tifffile
import bebi103

import bokeh.io
bokeh.io.output_notebook()

import gbeflow
```

Import first time point from each original tiff

```
files = glob.glob(os.path.abspath('../data/original/20181130-gbe_mutants_brt/*.tif'))
files = files
```

The `key` parameter for `tifffile.imread` lets us import just the first timepoint with both channels. This makes it possible so that we can look at a sample of all of the data at once. Since the total dataset is >20Gb we can't load it directly into memory.

```
%%time
raw = {}
for f in files:
    raw[f] = tifffile.imread(f,key=(0,1))
```

Select points to use for alignment

Try selecting two points on the dorsal surface that represent the linear plane of the dorsal side. This approach should hopefully be less sensitive to variability in how the user picks the points.

```
clicks = []
for f in files:
    clk = bebi103.viz.record_clicks(raw[f][1],flip=False)
    clicks.append(clk)
```

Extract the points selected for each image into a dataframe.

```
Ldf = []
for clk in clicks:
    Ldf.append(clk.to_df())

points = pd.concat(Ldf,keys=files)
points
```

Reshape `points` array to have one row per sample.

```
points = points.reset_index(level=1)
points.head()
```

Calculate a line for each embryo

$$y - y_1 = m(x - x_1)$$

$$m = \frac{y_2 - y_1}{x_2 - x_1}$$

```
line = gbeflow.calc_line(points)
```

```
line = line.reset_index().rename(columns={'index':'f'})
line.head()
```

Plot embryos with line on top

$$y = m(x - x_1) + y_1$$

```
# Create list to collect plot objects
Lp = []

# X values to compute line on
x = np.linspace(0,1024,100)

for f in files:
    p = bebi103.viz.imshow(raw[f][1,:,:],flip=False)

    x1 = line[line['f']==f]['x1'].values
    y1 = line[line['f']==f]['y1'].values
    m = line[line['f']==f]['m'].values
    y = m*(x-x1) + y1

    p.line(x,y,color='red',line_width=3)
#     p.scatter(line[line['f']==f]['x'], line[line['f']==f]['y'],color='white',
↪size=15)

    Lp.append(p)

bokeh.io.show(bokeh.layouts.gridplot(Lp,ncols=2))
```

Calculate rotation

The angle of rotation is calculated as follows

$$\theta = \arctan\left(\frac{y_2 - y_1}{x_2 - x_1}\right)$$

This calculation can be code-blockd using the `np.arctan2`, which has two arguments that correspond to Δy and Δx.

```
line = gbeflow.calc_embryo_theta(line)
line.head()
```

Apply rotation based on θ

With θ calculated, we are now ready to rotate each sample accordingly. Since we cannot load all of the data into memory at the same time, we will currently only rotate the first timepoint to check that it worked. After we have determined all necessary manipulations for each embryo, we will run the actual rotation.

```
# Dataframe to save first timepoint from each rotate embryo
rot = {}

# List to save bokeh plots
Lp = []

for f in tqdm.tqdm(files):
    # Extract the theta value for this sample
    theta = line[line['f']==f]['theta'].values[0]

    # Rotate single image
    rimg = scipy.ndimage.rotate(raw[f][1],theta)
```

(continues on next page)

(continued from previous page)

```
    # Save and plot first timepoint
    rot[f] = rimg
    p = bebi103.viz.imshow(rimg,title=f)
    Lp.append(p)
```

```
100%|| 4/4 [00:00<00:00,  6.63it/s]
```

```
bokeh.io.show(bokeh.layouts.gridplot(Lp,ncols=2))
```

```
# Import modules
import bebi103
import gbeflow
import tifffile
import scipy.ndimage

# Import image from tiff
raw = tifffile.imread(impath)

# Select first timepoint from `raw` to display
clk = bebi103.viz.record_clicks(raw[0],flip=False)

# Extract points from `clk` to a dataframe
points = clk.to_df().reset_index()

# Calculate line for each embryo
line = gbeflow.calc_line(points
                ).reset_index(
                ).rename(
                    columns={'index','f'}
                )

# Calculate angle of rotation
line = gbeflow.calc_embryo_theta(line)

# Rotate embryo
rimg = scipy.ndimage.rotate(raw,line['theta'])

# Save rotated stack to file
tifffile.imsave(outpath,data=rimg)
```

Manual curation of orientation

After the rotation based on θ has been applied, the embryos should be positioned such that the AP axis is horizontal. However, the rotation does not gurantee that the embryo will be positioned with anterior left and dorsal up. At this point, the user can individually specify any additional rotations to correct the embryo's orientation. The following examples are typically sufficient to correct most orientation errors:

```
# Rotate by 180 degrees around center point
img = scipy.ndimage.rotate(img,180)

# Flip horizontally by specifying a specific axis
img = np.flip(img,axis=-1)
```

(continues on next page)

(continued from previous page)

```
# Flip vertically
img = np.flip(img,axis=-1)
```

The final result of this workflow is that all samples are aligned dorsal up with the anterior end of the embryo to the left. This consistent alignment should facilitate future comparisons of the results of optical flow.

Embryo Masking

Rationale

We want to eliminate the background by segmenting the embryo from the background. Later in our analysis, our job will be easier if we do not need to worry about the background contributing noise.

Basic Approach

Segment the embryo from the background in order to clearly isolate changes in signal associated with germ band extension

```
import numpy as np
import matplotlib.pyplot as plt

from imp import reload
import sys
sys.path.insert(0, '..')
import utilities as ut

from skimage import img_as_float
from skimage.filters import gaussian
from skimage.segmentation import active_contour
from skimage.measure import grid_points_in_poly
```

Data Import

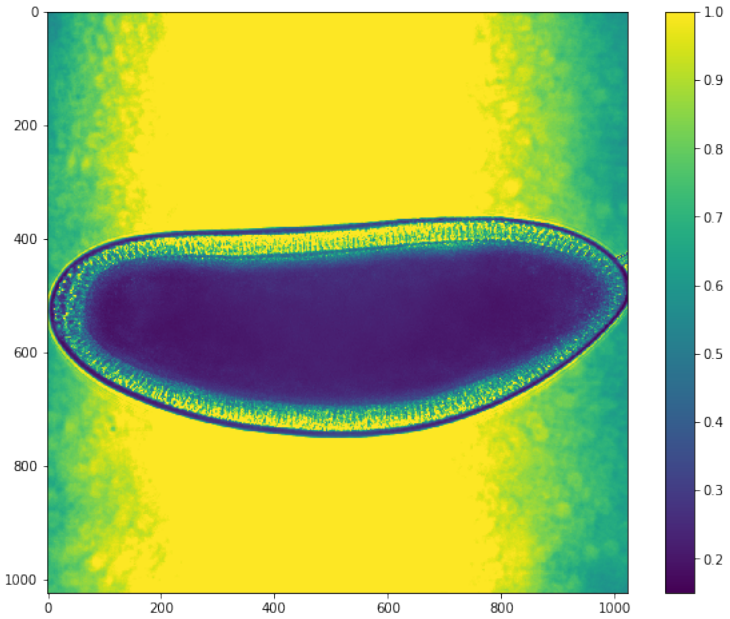
Read in hdf5 hyperstack as a 3d numpy array. Convert to float for ease of processing.

```
hst = img_as_float(ut.read_hyperstack('../data/wt_gbe_20180110.h5'))
```

Select a single image from the first timepoint to use as a test case for segmentation development

```
img = hst[0]
```

```
ut.imshow(img)
```



Background Segmentation

In order to analyze any signal variation that is associated with germ band extension, we need to isolate the signal of the embryo from any background. This task is complicated by the nature of brightfield images where the intensity of the background ~1 is similar to the intensity of cells within the image. As a result we cannot use a simple threshold which would remove signal from within the embryo as well as from the background.

However, the embryo does have a clear dark line that separates the embryo itself from the background. We will take advantage of this clear line to separate the embryo from the background using contour based methods.

To start, we will approximate an ellipse that is slightly bigger than the size of the embryo, which will provide a starting point for contour fitting.

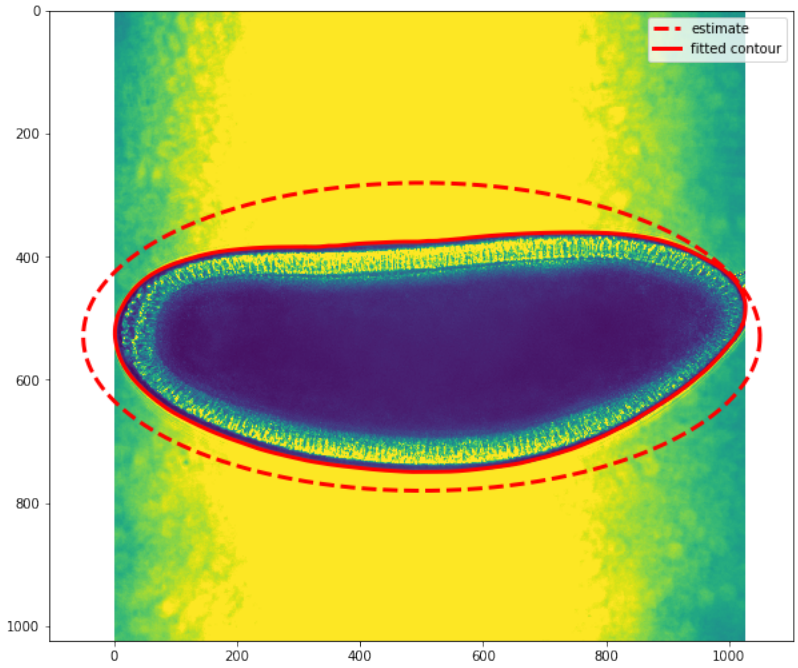
```
# Values to calculate the ellipse over using parametrized trigonometric fxns
s = np.linspace(0, 2*np.pi, 400)

# Define the approximate center of the embryo/ellipse
center_x = 500
center_y = 530

# Define the radius of the ellipse in x and y
radius_x = 550
radius_y = 250

# Calculate the position of the ellipse as a function of s
x = center_x + radius_x*np.cos(s)
y = center_y + radius_y*np.sin(s)
init = np.array([x, y]).T
```

Next we will use `skimage`'s `active_contour` method to fit our approximated ellipse to the contour of the embryo. The kwarg parameters for this function were copied based on the active countour tutorial.

```
snake = active_contour(gaussian(img, 3),
                       init, alpha=0.015, beta=10, gamma=0.001)
```

```
fig,ax = plt.subplots(figsize=(10,10))
ax.imshow(img)
ax.plot(init[:, 0], init[:, 1], '--r', lw=3,label='estimate')
ax.plot(snake[:, 0], snake[:, 1], '-r', lw=3,label='fitted contour')
ax.legend()
```

```
<matplotlib.legend.Legend at 0x1c19e6f048>
```



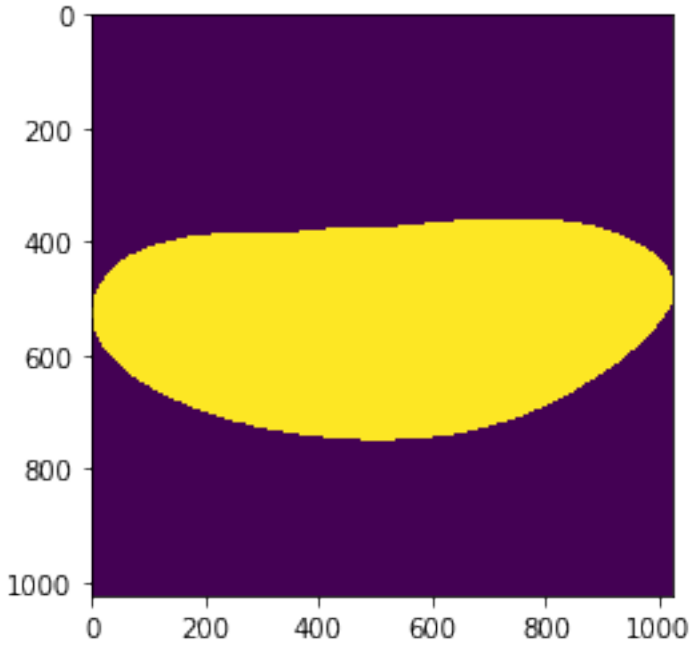
The plot above shows our image overlaid with the approximated ellipse (dashed line) and the fitted counter (red continuous line). This contour follows the boundary between the embryo and the background.

Create background mask

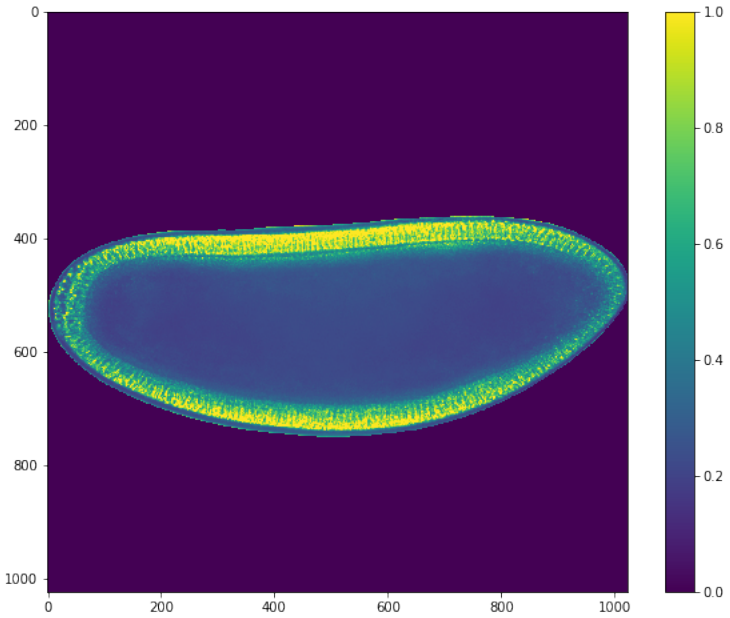
Now that we have a estimated function `snake` that defines the boundary of the embryo and the background, we need to define a mask in the shape of the image that defines which points belong in the image. Skimage's `grid_points_in_poly` function takes a set of points defining a shape (`snake`) and identifies which points over a given raster area fall within the input shape.

```
mask = grid_points_in_poly(img.shape, snake).T
plt.imshow(mask)
```

```
<matplotlib.image.AxesImage at 0x1c1aa11c88>
```




```
im_masked = img.copy()
im_masked[~mask] = 0
ut.imshow(im_masked)
```



Write a function to fit a contour to a new embryo

```
def calc_ellipse(center_x,center_y,radius_x,radius_y):
    '''
    Calculate a parametrized ellipse based on input values
    '''
```

(continues on next page)

(continued from previous page)

```
    # Values to calculate the ellipse over using parametrized trigonometric fxns
    s = np.linspace(0, 2*np.pi, 400)

    # Calculate the position of the ellipse as a function of s
    x = center_x + radius_x*np.cos(s)
    y = center_y + radius_y*np.sin(s)
    init = np.array([x, y]).T

    return(init)
```

```
def contour_embryo(img,init):
    '''
    Fit a contour to the embryo to separate the background
    Returns a masked image where all background points = 0
    '''

    # Fit contour based on starting ellipse
    snake = active_contour(gaussian(img, 3),
                       init, alpha=0.015, beta=10, gamma=0.001)

    # Create boolean mask based on contour
    mask = grid_points_in_poly(img.shape, snake).T

    # Apply mask to image and set background to 0
    img[~mask] = 0

    return(img)
```

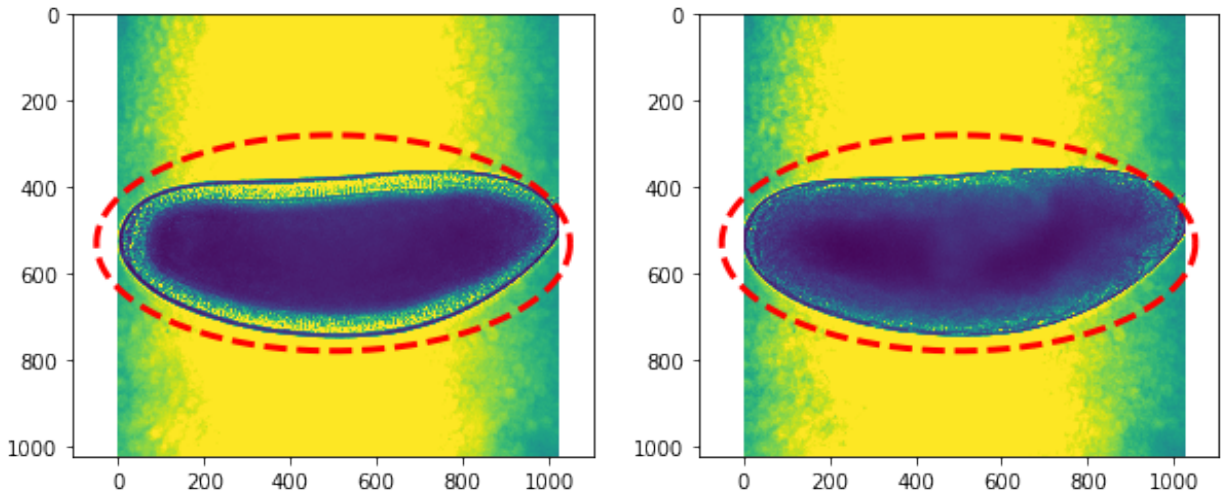
Apply mask to hyperstack

Check ellipse approximation on first and last timepoints

```
center_x,center_y = 500,530
radius_x,radius_y = 550,250
ellipse = calc_ellipse(center_x,center_y,radius_x,radius_y)
```

```
fig,ax = plt.subplots(1,2,figsize=(10,8))
ax[0].imshow(hst[0])
ax[0].plot(init[:,0],init[:,1],'--r',lw=3)
ax[1].imshow(hst[-1])
ax[1].plot(init[:,0],init[:,1],'--r',lw=3)
```

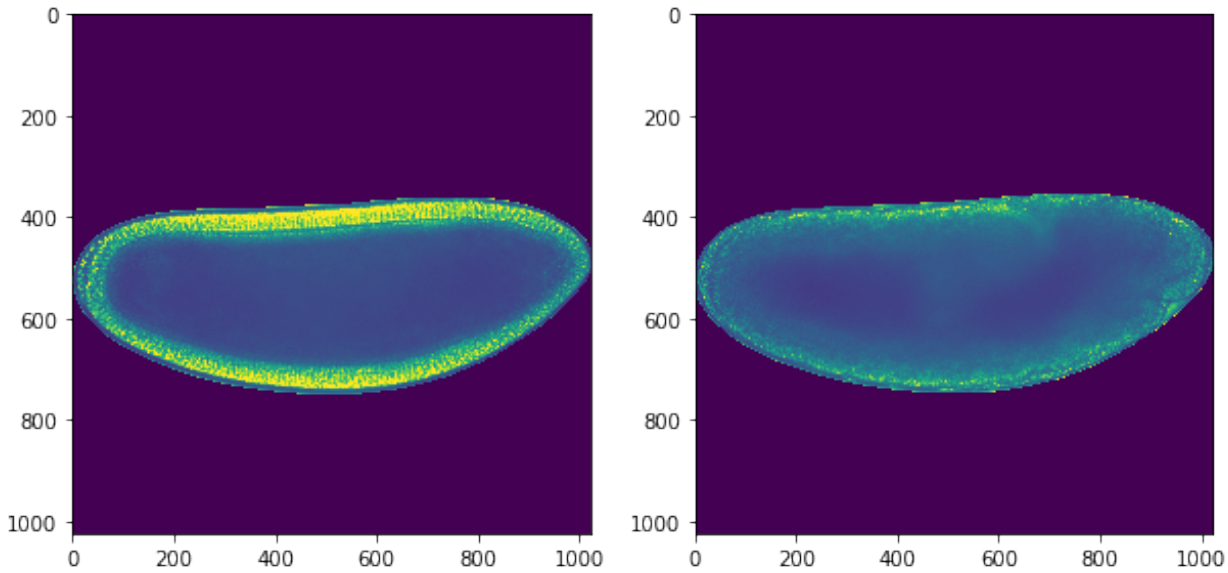
```
[<matplotlib.lines.Line2D at 0x1c1a65a978>]
```




```
# Loop through each timepoint in hyperstack
for t in range(hst.shape[0]):
    hst[t] = contour_embryo(hst[t],ellipse)
```

```
fig,ax = plt.subplots(1,2,figsize=(10,8))
ax[0].imshow(hst[0])
ax[1].imshow(hst[-1])
```

```
<matplotlib.image.AxesImage at 0x1c1a3349e8>
```



Following the development of the method above, an object was created within gbeflow to handle the tasks associated with contouring: `MaskEmbryo`. An example of using this new class is included below.

```
import gbeflow
import tifffile
import bebi103

# Import sample from tiff
```

(continues on next page)

(continued from previous page)

```
img = tifffile.imread(filepath)

# Select endpoints of embryo for ellipse calculation
clk = bebi103.viz.record_clicks(raw[0],flip=False)

# Save points to dataframe after selection
points = clk.to_df()

# Initialize object with selected points
me = gbeflow.MaskEmbryo(points)

# Contour embryo based on ellipse calculated during initialization
# Input accepts only 2d data
mask = me.contour_embryo(img[0])

# Apply binary mask to the raw image
mask_img = me.mask_image(img,mask)
```

After running the code above, this workflow can be applied to the remaining timepoints and saved as a tif.

Limitations

While this approach worked well on the test dataset, when applied to other embryos it would make mistakes on ~10% of frames. There does not appear to be a straightforward way to improve performance without manual curation. See 20181106-endpoint_ellipse.ipynb and 20181108-vector_calculation.ipynb for examples of masking mistakes.

Next Steps

The implementation of contour masking shown in 20181108-vector_calculation.ipynb fits an individual contour to create a mask for each timepoint in the timecourse. Simply by repeating the calculation many times, we introduce more opportunities for mistakes. In the future the contour should be fit just to the first timepoint to create a mask that the optical flow algorithm can use on all timepoints (See the paper's supplement for more information.) To accommodate any minute changes in embryo size, we could try to increase the size of the mask by 10%, but given the large scale of the movements that we are interested in it may not be necessary.

1.5.3 Optical Flow

Implementation in Matlab

The optical flow algorithm is implemented in a Matlab script published in the supplement to Vig et al. 2016. It can be run directly in Matlab according to the instructions in the supplementary PDF. In order for Matlab to use the script *OpticalFlow.m*, it needs to either be placed in the same folder as the data or the path to the script needs to be added to the Matlab environment before running the script.

```
addpath('path/to/OpticalFlow.m')
[X,Y,Vx,Vy,Mov] = OpticalFlow (MovieName, BinaryMask, scale, dt, BoxSize, BlurSTD,␣
↪ArrowSize)
```

The `OpticalFlow` function takes a set of parameters that are described in the supplementary materials of Vig et al. 2016. The parameters are summarized below:

Table 1: Optical Flow Parameters (Vig et al. 2016)

Parameter	Description	Value
MovieName	Avi or tiff file to be analyzed	
BinaryMask	Name of the region-of-interest image sequence files can be either AVI or TIFF format. If an ROI mask is not necessary then the input is [ ]	[ ]
scale	Converts pixels to microns scale is defined in microns per pixel	0.5 μm/pixel (must be a float)
dt	Time interval between frames	60.0 s (must be a float)
BoxSize	Sets the linear size of the subregions (in pixels) where the velocity is computed. Should be set to be large enough that each subregion contains at least one identifiable image feature.	30 pixels
BlurSTD	Sets the size of the standard deviation for the Gaussian blur. Should be set to half maximum velocity between two images in pixels.	1.0 (must be a float)
ArrowSize	Used to define a coarser output grid for the velocity vectors. Defines the spacing (in pixels) between output velocity vectors.	5
Optical Flow Method	To access an augmented mode of Optical Flow enter 'Rotation' to determine the local voriticity (ωo) or 'React' to measure the effects of an added source term (γ). To run the standard default version use 'none' or omit this input parameter.	

Selecting Parameter Values

BlurSTD and BoxSize are the two parameters that are most difficult to determine for a naive user. In order to determine appropriate values for brightfield data, I tried running combinations of BoxSize={10,20,30,40,50} and BlurSize={1,4,10,20,30} and reviewed the outputs to select parameter values that had the least noise without obscuring features in the data. This same approach could be applied to any new data types that are investigated in the future.

Running OpticalFlow from python

Matlab distributes its application with a python installation, which can be installed by the user. See this link for instructions on installing the matlab python package.

Managing the output of OpticalFlow.m

`OpticalFlow.m` returns a set of variables as output: `[X,Y,Vx,Vy,Mov]`. Transfering matlab data objects directly to python is difficult and limited to a relatively simple data structures. To avoid this problem, I wrote a function that wraps around `OpticalFlow.m` and saves the output to a set of csv files and an avi. The code shown below is available in OpticalFlowOutput.m.

```
function OpticalFlowOutput(OutName,MovieName,BinaryMask,BoxSize,BlurSTD,ArrowSize,
↪scale,dt)

    [ X,Y,Vx,Vy,mov ] = OpticalFlow( MovieName,BinaryMask,BoxSize,BlurSTD,ArrowSize,
↪scale,dt );

    csvwrite(strcat(OutName,'_X.csv'),X);
    disp(size(X));
    csvwrite(strcat(OutName,'_Y.csv'),Y);
    disp(size(Y));
```

(continues on next page)

(continued from previous page)

```
    csvwrite(strcat(OutName,'_Vx.csv'),Vx);
    disp(size(Vx));
    csvwrite(strcat(OutName,'_Vy.csv'),Vy);
    disp(size(Vy));

    vidobj = VideoWriter(strcat(OutName,'.avi'));
    vidobj.open();
    vidobj.writeVideo(mov);
    vidobj.close();

end
```

Interacting with matlab from python

Once the matlab package is installed in the `python36` environment, we can run `OpticalFlow.m` script directly from python using a Jupyter Notebook.

```
# First we need to import the matlab engine package
import matlab.engine

# Next start the engine
eng = matlab.engine.start_matlab()

# Add path for matlab script to namespace
eng.addpath(r'../matlab',nargout=0)

# Define opticalflow parameter values
BinaryMask = matlab.single([])
scale = 0.5
dt = 1.0
BoxSize = 30
BlurSTD = 1.0
ArrowSize= 5

# Define file paths and names
outpath = 'abs/path/name'
inpath = 'abs/path/name.tif'

# Run optical flow script
eng.OpticalFlowOutput(outpath, inpath, BinaryMask,
                      BoxSize, BlurSTD, ArrowSize,
                      scale, dt, nargout=0)
```

Some notes about running matlab from python

When Matlab functions are called from Python, we need to include an additional arguement `nargout`, which tells Matlab whether we expect a value returned by the function. For the functions `eng.addpath` and `eng.OpticalFlowOutput`, there are no return values so we can just append `nargout=0` to the function parameters as shown above.

There are two options for the parameter BinaryMask. If we want to use a previously calculated binary mask, we can pass the absolute path to an avi or tiff file that matches the primary file that we are analyzing. Alternatively, if we don't want to use a mask, we can set BinaryMask to `matlab.single([])`, which passes an empty array to Matlab.

Finally, the `OpticalFlow.m` script and Matlab seem happiest when working with absolute paths as opposed to relative paths. For convenience, we can import the package `os` and use the function `os.path.abspath()` to convert any relative or complete path to an absolute path. For example:

```
import os

relpath = '../data/example.tif'
abspath = os.path.abspath(relpath)
```

Wrangling the optical flow ouput data

`OpticalFlow.m` returns 5 data objects `[X,Y,Vx,Vy,mov]` that are saved to output files when using `OpticalFlowOutput.m`. Given a user input which defines the base of the output file names hereafter refered to as `<name>`, `OpticalFlowOutput` saves the 5 data objects.

Table 2: Optical Flow Output

Object	Filename	Description
X	<name>_X.csv	A vector with the unique x positions for the vector grid
Y	<name>_Y.csv	A vector with the unique y positions for the vector grid
Vx	<name>_Vx.csv	Contains the X component of the velocity vector. An MxN matrix where there are M columns corresponding to time and N rows corresponding the the spatial positions stored in X and Y
Vy	<name>_Vy.csv	Contains the Y component of the velocity vector. An MxN matrix where there are M columns corresponding to time and N rows corresponding the the spatial positions stored in X and Y
mov	<name>.avi	A movie of the original input data with the corresponding vector field overlaid using green arrows.

The function *gbeflow.tidy_vector_data()* loads the 4 vector files saved by optical flow ouput by looking for the root of the filename <name>. The four data files are compiled into a single dataframe that contains five columns: frame, x, y, vx, and vy. The dataframe which the function returns can be saved to a csv file using `pandas.DataFrame.to_csv()`.

In order to facilitate interpolation and plotting, we also want to transform the data into an array based structure as opposed to a tidy dataframe where each row corresponds to a single point/velocity vector. The function *gbeflow.reshape_vector_data()* accepts the dataframe output by *gbeflow.tidy_vector_data()* as an input. It creates a set of arrays that conform to the following dimensions: # of time points $\times$ # of unique x values $\times$ # of unique y values. The function returns five arrays following this convention: `tt`, `xx`, `yy`, `vx`, and `vy`. If we use the same index to select a value from each of the 5 arrays, we will get the t, x and y positions with the corresponding vx and vy velocity components.

The following code is an example of how to import the results of optical flow after running the example code above.

```
import gbeflow

# Define file paths and names
outpath = 'abs/path/name'

# Read in data to a single dataframe
df = gbeflow.tidy_vector_data(outpath)
```

(continues on next page)

(continued from previous page)

```
# Convert to array based format
tt,xx,yy,vx,vy = gbeflow.reshape_vector_data(df)
```

1.5.4 Simulated Cell Tracking

Objective

While the calculation of the vector field gives us new information about our samples, we still need to develop metrics that enable quantification of the movement of the germband and comparison to other samples. Since there is already an established set of metrics for comparint single cell tracks, we will try to generate single cell tracks from our vector field.

Managing time units

When we run the optical flow algorithm, it takes a parameter for δt however, an initial exploration of the role of this parameter indicates that changing the value does not impact the output of the algorithm. This finding raises the question of what are the units for the vectors output by the algorithm. We will use dimensional analysis of the advection equation to determine the units.

The advection equation describes the movement of a particle carried by a flow

$$\Delta I = I(t + \Delta t) - I(t) = -\Delta t(\mathbf{v} \cdot \nabla I)$$

ΔI and ∇I are defined in arbitrary units (au) and describe the change in intensity and the gradient of intensity respectively. The unit of time is not explicitly specified, but we will measure it in seconds (s). Given these units, we can calculate the units of $\mathbf{v}$ ($[\mathbf{v}]$).

$$\begin{aligned} \text{au} &= \text{s}([\mathbf{v}] \cdot \text{au}) && (1.1)\\ \frac{\text{au}}{\text{au}} &= \text{s}[\mathbf{v}] && (1.2)\\ \frac{1}{\text{s}} &= [\mathbf{v}] && (1.3)\\ && (1.4) \end{aligned}$$

We find that our velociy vector $\mathbf{v}$ has units of $1/\text{s}$. When we are calculating the movement of a simulated cell through our fector field, we need to multiply the velocity by the time step in order to calculate the change in position.

$$\begin{aligned} \mathbf{x}_n &= \begin{bmatrix} x_n \\ y_n \end{bmatrix} && (1.5)\\ \mathbf{x}_{n+1} &= \mathbf{x}_n + \Delta t(1.6) \end{aligned}$$

Selecting start points

For efficency, we currently ask the user to select a set of points to use as start points for tracks that will be generated using interpolation. We will take advantage of the bebi103 module by using `bebi103.viz.record_clicks`. In order to record data, the user needs to select the tab on the right side of the figure that shows three points. Then an click on the image will be recorded according to its position.

Interpolation

Currently, the function that interpolates the vector fields for simulated cell tracking *gbeflow.VectorField* relies on `scipy`'s interpolate.RectBivariateSpline to estimate the vector for each simulated cell. When *gbeflow.VectorField.initialize_interpolation()* runs, it saves the `scipy` interpolation object to *gbeflow.VectorField.Ldx* or *gbeflow.VectorField.Ldy* for each timepoint. In order to calculate the trajectory of a single cell, the function *gbeflow.VectorField.calc_track()* uses the previously calculated interpolation objects stored in *gbeflow.VectorField.Ldx* and *gbeflow.VectorField.Ldy* and evaluates the position of the cell to generate the x and y components of the next vector.

Code Examples

Data Import

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

import tifffile

import bebi103

import bokeh.io
# This variable needs to be specified for bokeh to render plots in notebook
notebook_url = 'localhost:8888'
bokeh.io.output_notebook()

import os
import sys
import glob
from imp import reload
import tqdm
import gbeflow
```

We'll start by grabbing a list of csv files that were generated by `OpticalFlowOutput.m`.

```
csvs = glob.glob('*_Vx.csv')
```

```
['yolk3_Vx.csv',
 '20180110_htl_glc_sc6_mmzm_rotate_brt_Vx.csv',
 'yolk_Vx.csv',
 'original_Vx.csv',
 'test3_Vx.csv',
 '20180112_htlglc_tl_sc4_resille_rotate_brt_Vx.csv',
 '20180108_htl_glc_sc2_mmzm_wp_rotate_brt_Vx.csv',
 '20180110_htl_glc-CreateImageSubset-02_sc11_htl_rotate_brt_Vx.csv',
 '20180108_htl_glc_sc9_mmzp_rotate_brt_Vx.csv',
 '20180108_htl_glc_sc11_mmzm_rotate_brt_Vx.csv',
 'test2_Vx.csv',
 '20180112_htlglc_tl_sc11_mmzp_rotate_brt_Vx.csv',
 'test_Vx.csv',
 'yolk2_Vx.csv',
 '20180110_htl_glc_sc15_mmzm_rotate_brt_Vx.csv',
 'sc11_Vx.csv',
 '20180110_htl_glc_sc14_mmzp_rotate_brt_Vx.csv',
```

(continues on next page)

(continued from previous page)

```
 'test4_Vx.csv',
 '20180110_htl_glc-CreateImageSubset-01_sc10_wt_rotate_brt_Vx.csv']
```

In order to load the data in using *gbeflow.tidy_vector_data()* we need to isolate the root <name> in the set of file names we collected.

```
names = set([f[:-7] for f in csvs])
```

```
{'20180108_htl_glc_sc11_mmzm_rotate_brt',
 '20180108_htl_glc_sc2_mmzm_wp_rotate_brt',
 '20180108_htl_glc_sc9_mmzp_rotate_brt',
 '20180110_htl_glc-CreateImageSubset-01_sc10_wt_rotate_brt',
 '20180110_htl_glc-CreateImageSubset-02_sc11_htl_rotate_brt',
 '20180110_htl_glc_sc14_mmzp_rotate_brt',
 '20180110_htl_glc_sc15_mmzm_rotate_brt',
 '20180110_htl_glc_sc6_mmzm_rotate_brt',
 '20180112_htlglc_tl_sc11_mmzp_rotate_brt',
 '20180112_htlglc_tl_sc4_resille_rotate_brt',
 'original',
 'sc11',
 'test',
 'test2',
 'test3',
 'test4',
 'yolk',
 'yolk2',
 'yolk3'}
```

Now we can define a list that just contains the root file names that we are interested in.

```
fs = ['20180108_htl_glc_sc11_mmzm_rotate_brt',
 '20180108_htl_glc_sc2_mmzm_wp_rotate_brt',
 '20180108_htl_glc_sc9_mmzp_rotate_brt',
 '20180110_htl_glc-CreateImageSubset-01_sc10_wt_rotate_brt',
 '20180110_htl_glc-CreateImageSubset-02_sc11_htl_rotate_brt',
 '20180110_htl_glc_sc14_mmzp_rotate_brt',
 '20180110_htl_glc_sc15_mmzm_rotate_brt',
 '20180110_htl_glc_sc6_mmzm_rotate_brt',
 '20180112_htlglc_tl_sc11_mmzp_rotate_brt',
 '20180112_htlglc_tl_sc4_resille_rotate_brt']
```

We can now initialize the object *gbeflow.VectorField*. This object will facilitate importing the data, interpolating over the vector field, and generating simulated cell tracks.

```
vf = {}
for f in fs:
    vf[f] = gbeflow.VectorField(f)
```

Each item in the dictionary `vf` is a vector field object with a key based on the root file name.

```
vf.keys()
```

```
dict_keys(['20180108_htl_glc_sc11_mmzm_rotate_brt',
        '20180108_htl_glc_sc2_mmzm_wp_rotate_brt',
        '20180108_htl_glc_sc9_mmzp_rotate_brt',
```

(continues on next page)

(continued from previous page)

```
        '20180110_htl_glc-CreateImageSubset-01_sc10_wt_rotate_brt',
        '20180110_htl_glc-CreateImageSubset-02_sc11_htl_rotate_brt',
        '20180110_htl_glc_sc14_mmzp_rotate_brt',
        '20180110_htl_glc_sc15_mmzm_rotate_brt',
        '20180110_htl_glc_sc6_mmzm_rotate_brt',
        '20180112_htlglc_tl_sc11_mmzp_rotate_brt',
        '20180112_htlglc_tl_sc4_resille_rotate_brt'])
```

Now we can import the image data that matches each vector field object.

```
for f in vf.keys():
    vf[f].add_image_data(os.path.join('../data',vf[f].name+'.tif'))
```

Track Calculation

Using the image data, we can pick starting points for our tracks. We are going to save the bokeh plotting object generated by *gbeflow.VectorField.pick_start_points()* into a list so that we can extract the click record when we are done.

```
L = []
for f in vf.keys():
    L.append(vf[f].pick_start_points())
```

After points have been selected on each image, we will save the click record back into each `vf` object.

```
for i,f in enumerate(vf.keys()):
    vf[f].save_start_points(L[i])
```

We're now ready to use interpolation to generate the tracks for each start point.

```
for f in vf.keys():
    vf[f].calc_track_set(vf[f].starts,60,name='dt60')
```

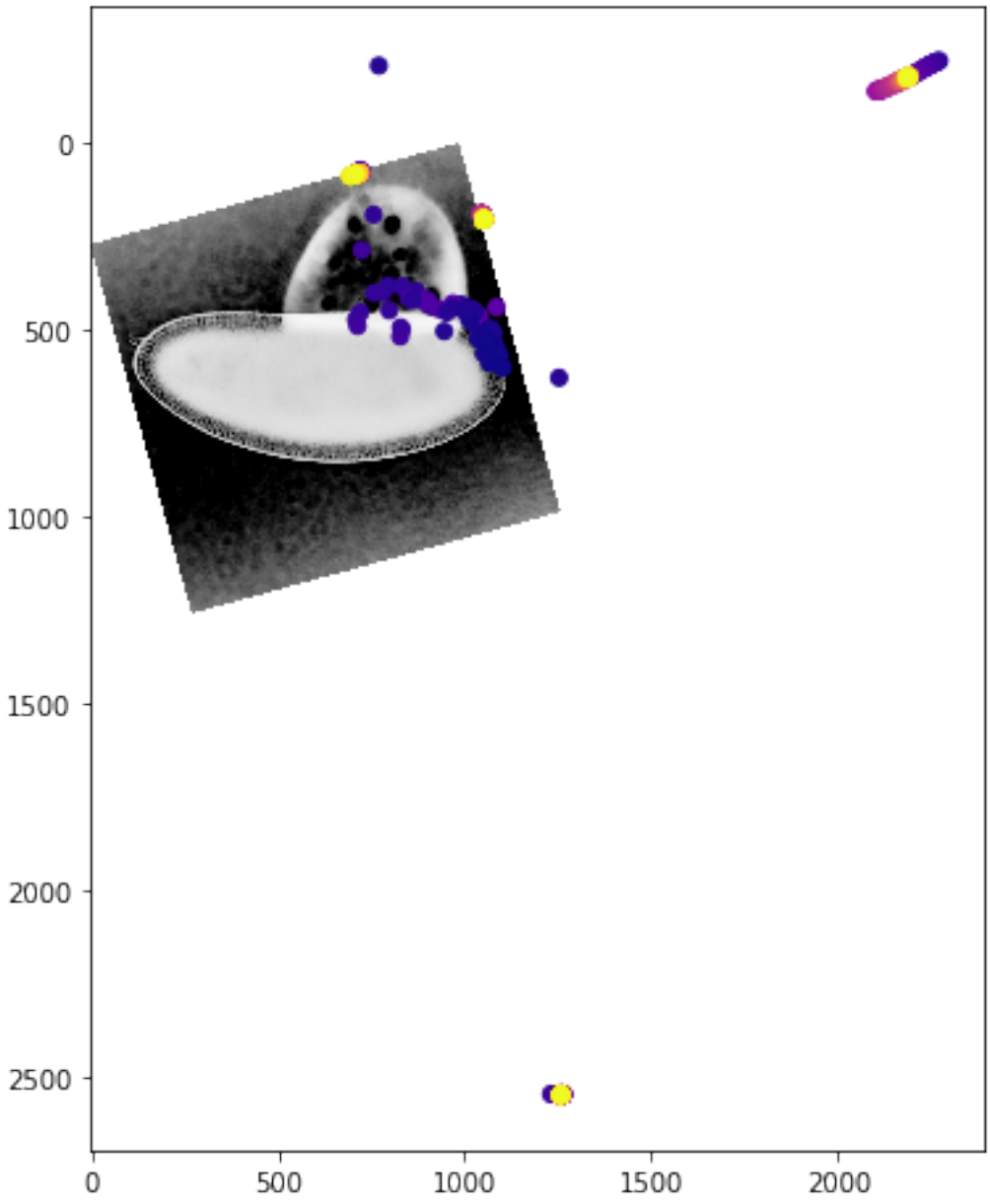
Now we can extract the set of tracks from each `vf` object and save it to a single dataframe for plotting.

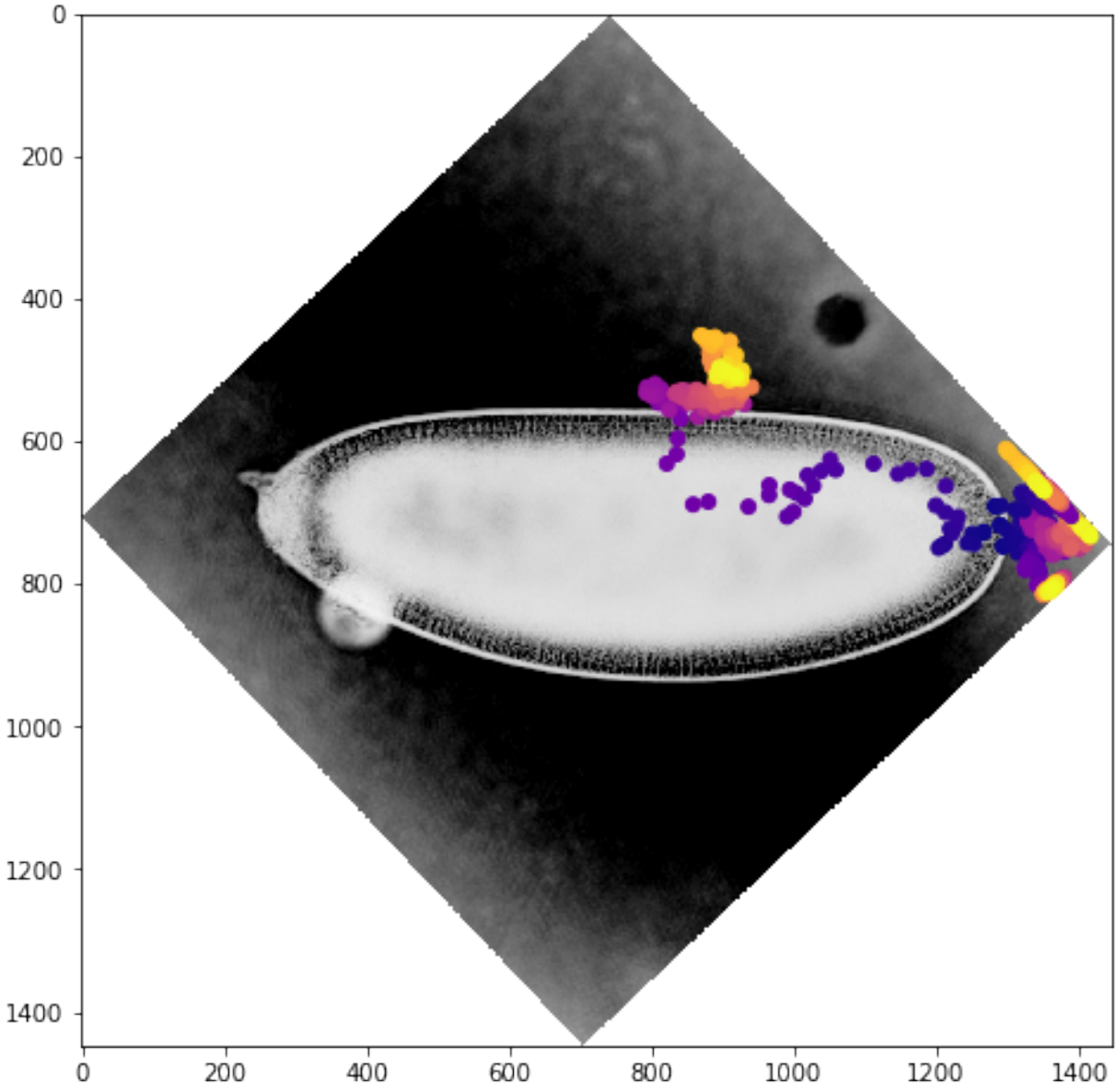
```
# Create a list of track dataframes
Ldf = []
for f in vf.keys():
    Ldf.append(vr[f].tracks)

# Join list of dataframes into a single dataframe
tracks = pd.concat(Ldf,keys=fs)

# Clean up the structure of the dataframe for clarity
tracks = tracks[tracks['name']=='dt60'].reset_index(
                                    ).drop(columns=['level_1']
                                    ).rename(columns={'level_0':'f'})

# Save tracks to csv for later follow up
tracks.to_csv('tracking.csv')
```

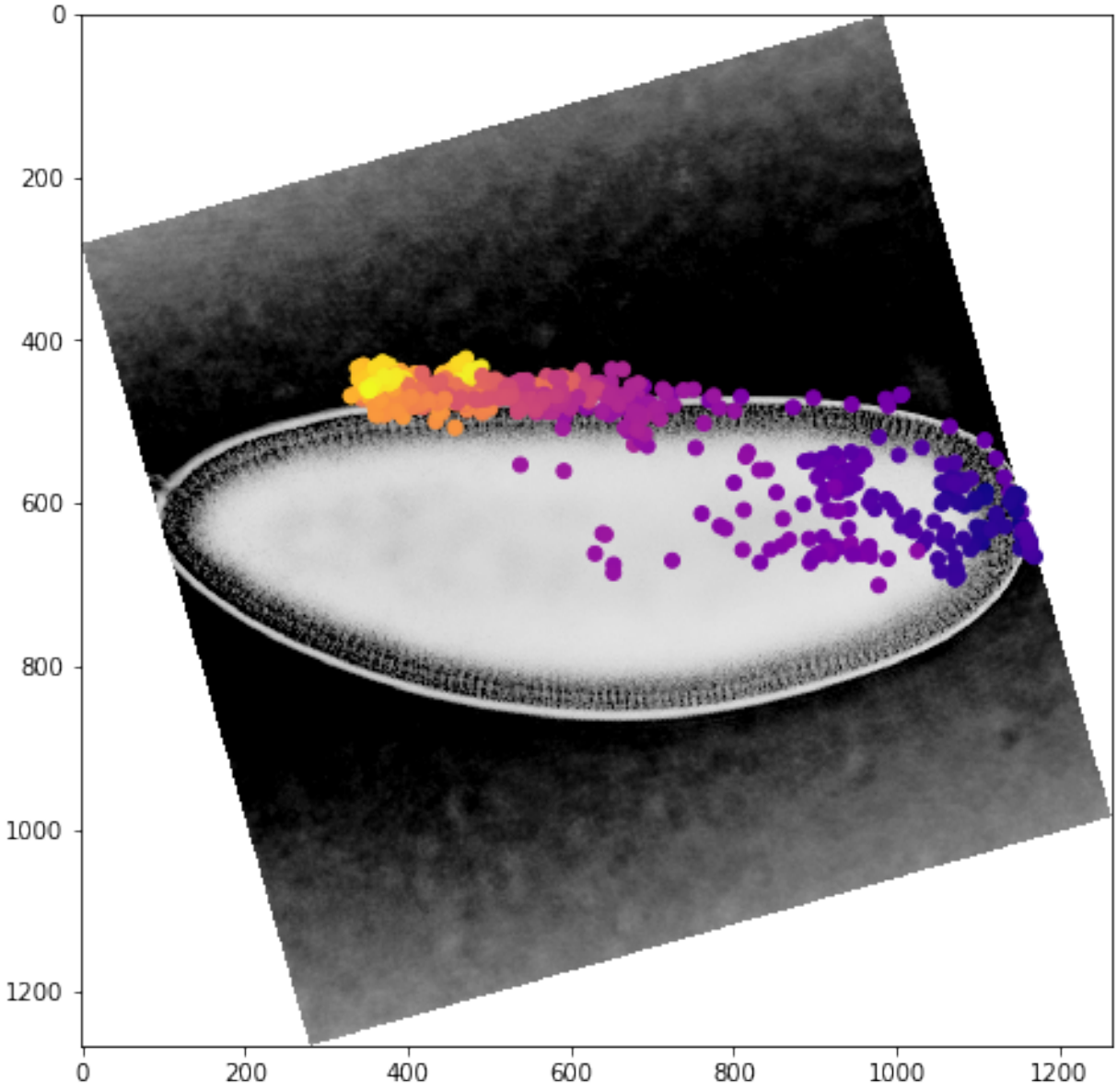
Static Track Visualization

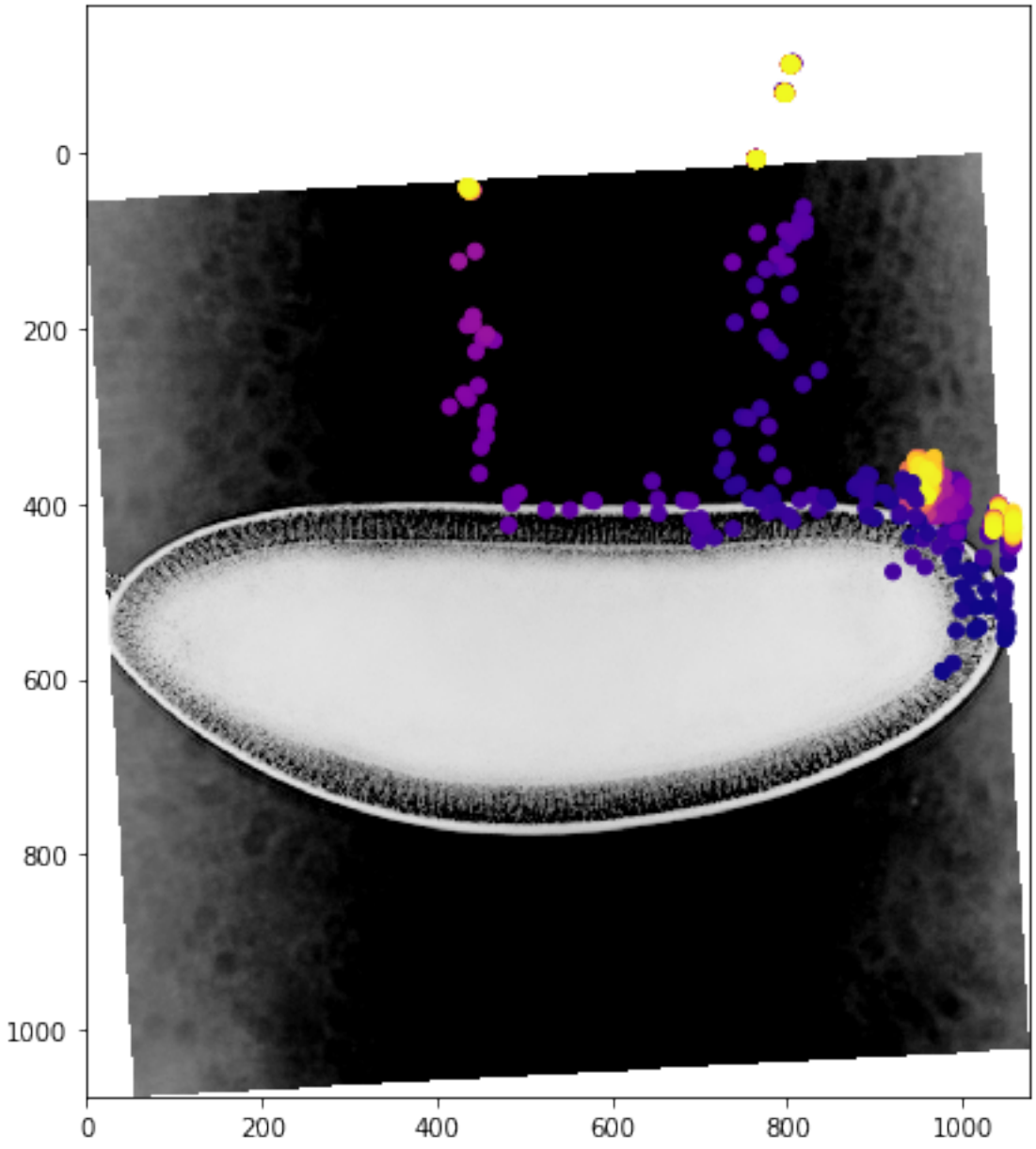
We can get a better sense of the tracking results by plotting the tracks directly on top of the image of the embryo. We'll look just at the first timepoint and display the image in grey so that we can easily plot on top of it. To show time in our tracks, we'll color code each point according to its time value using the plasma colormap.

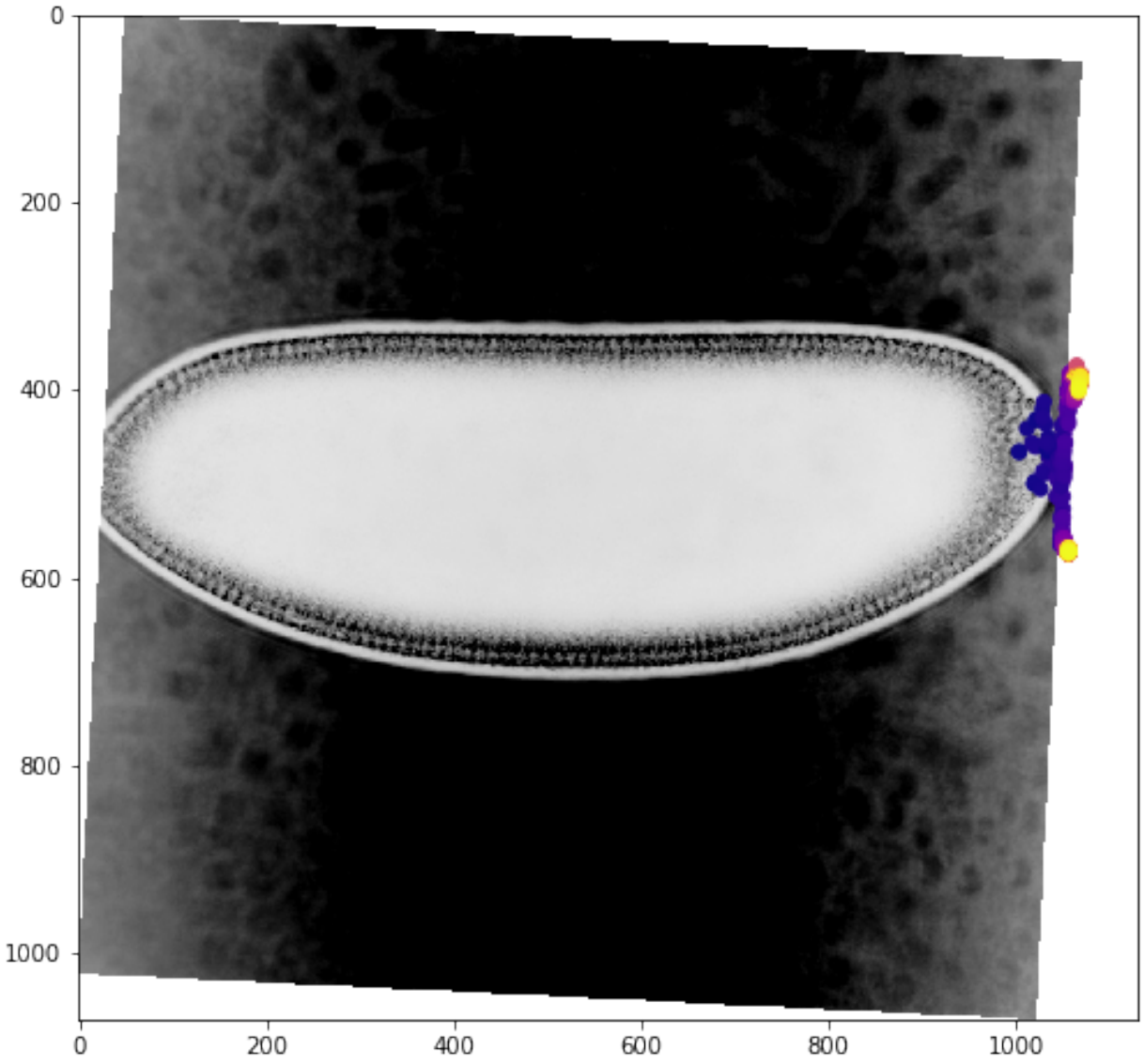
```
for f in fs:
    fig,ax = plt.subplots(figsize=(10,8))
    ax.imshow(vf.img[0],cmap='Greys')
    sb = tracks[tracks['f']==f]
    ax.scatter(sb.x,sb.y,c=sb['t'].values,cmap='plasma')
```

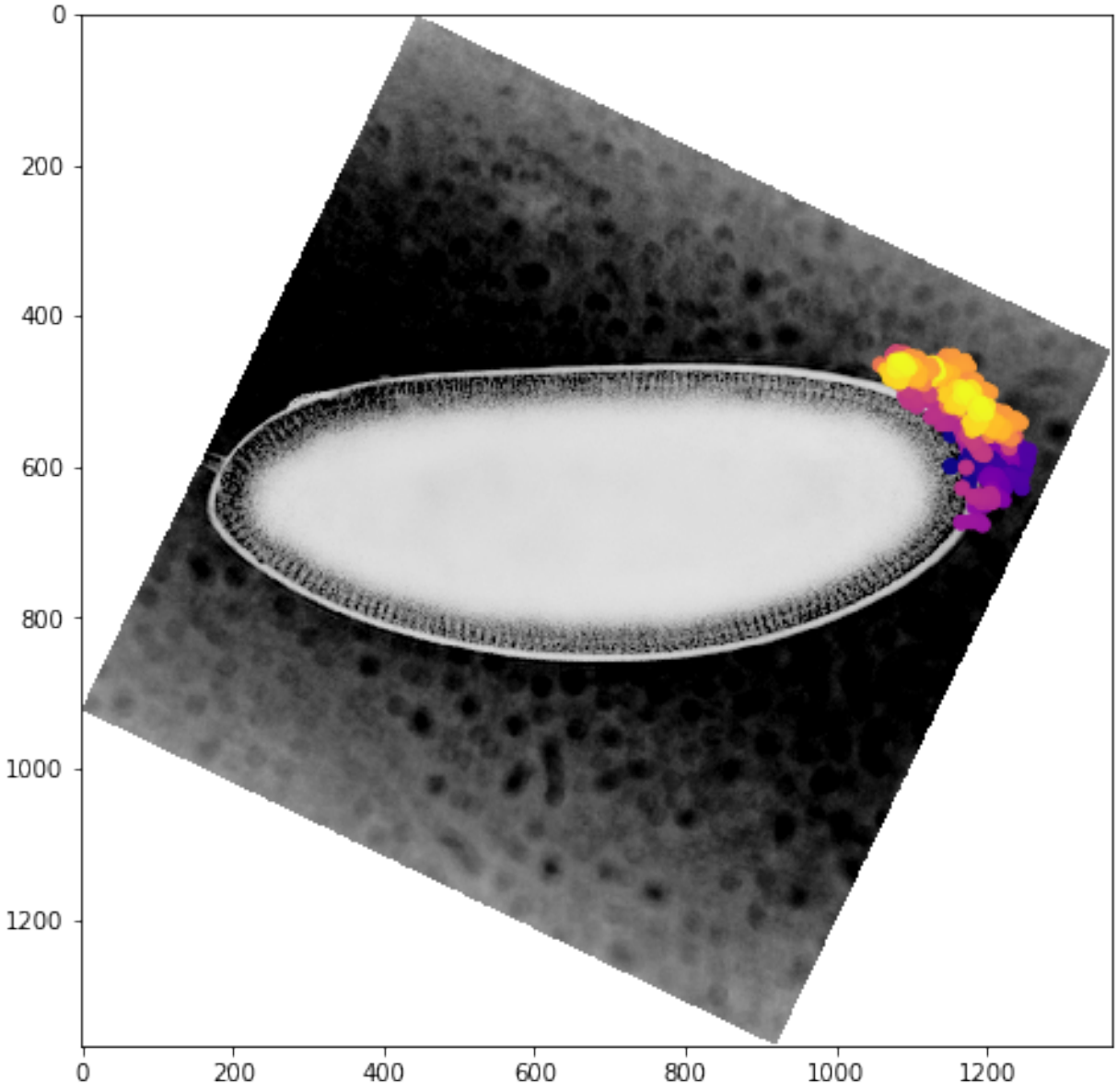


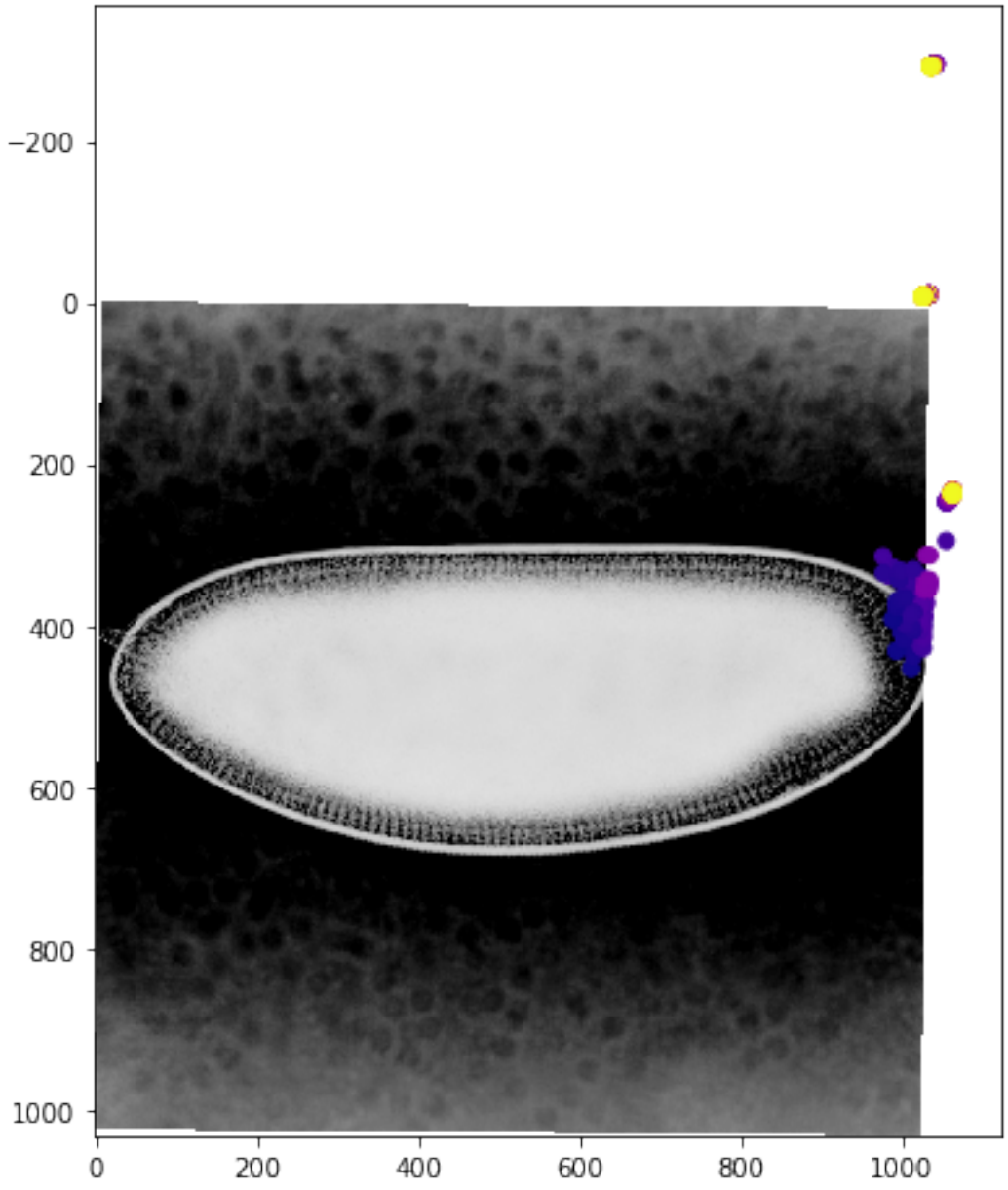

0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400

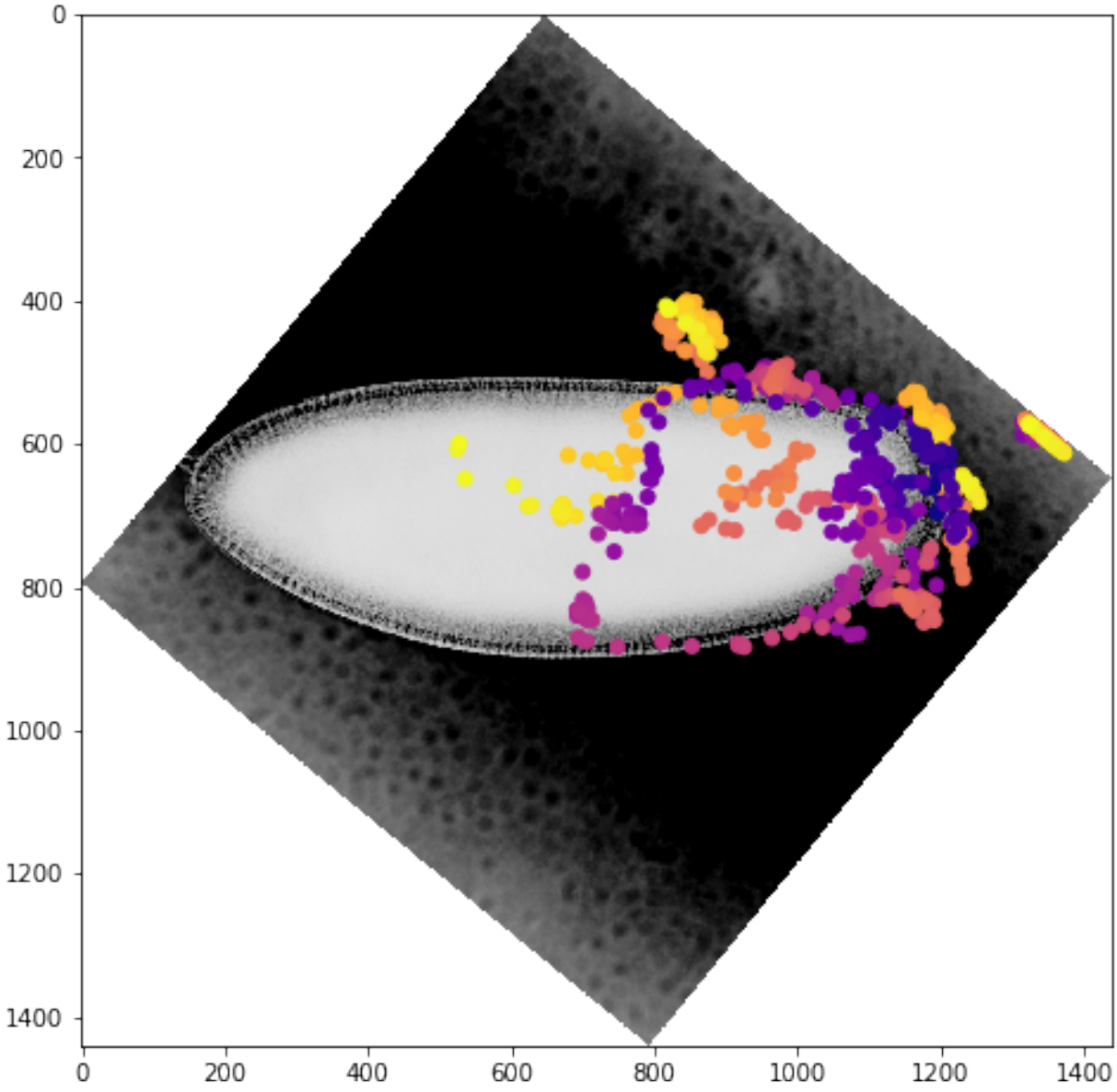

0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200

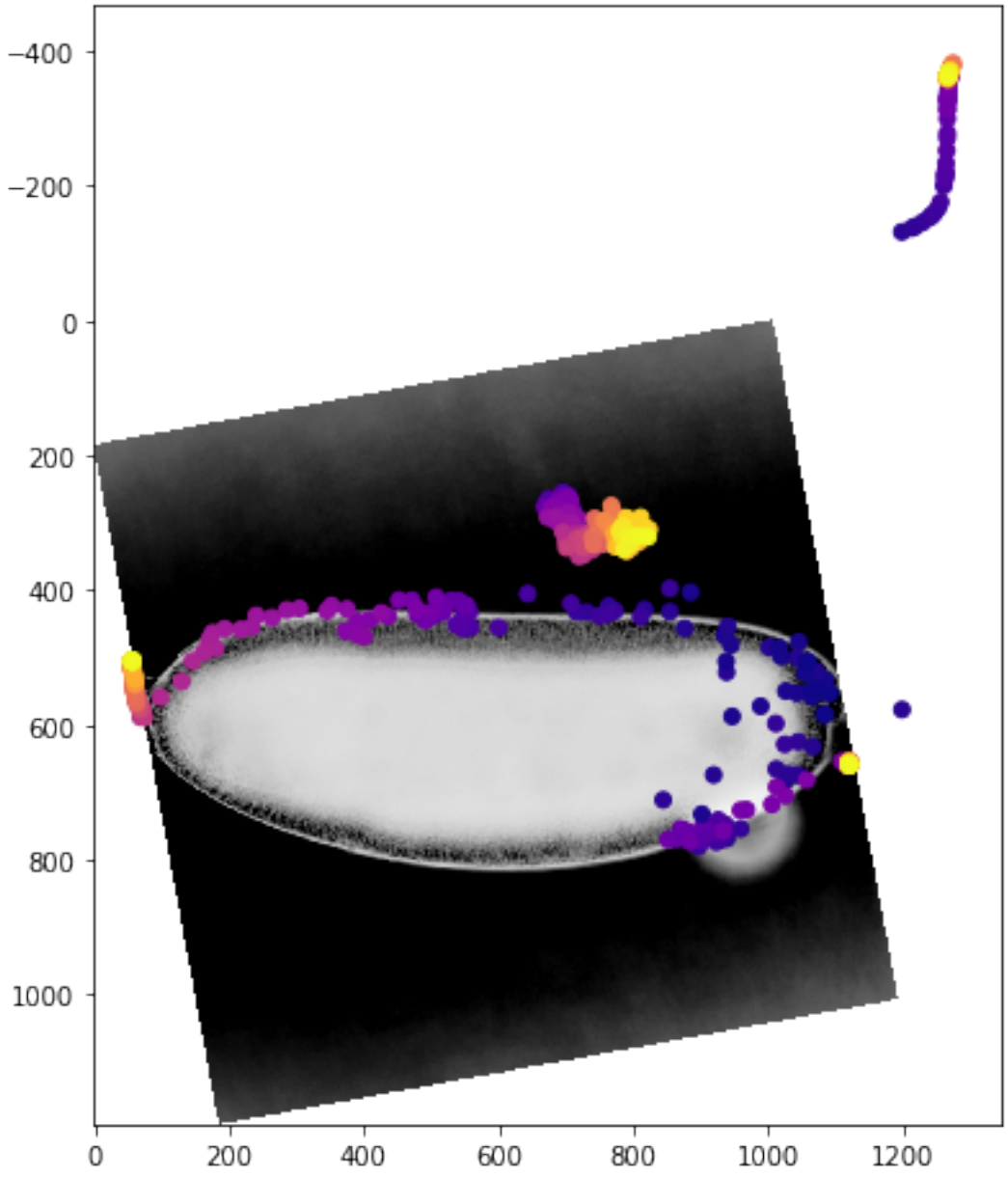

0
200
400
600
800
1000
0
200
400
600
800
1000

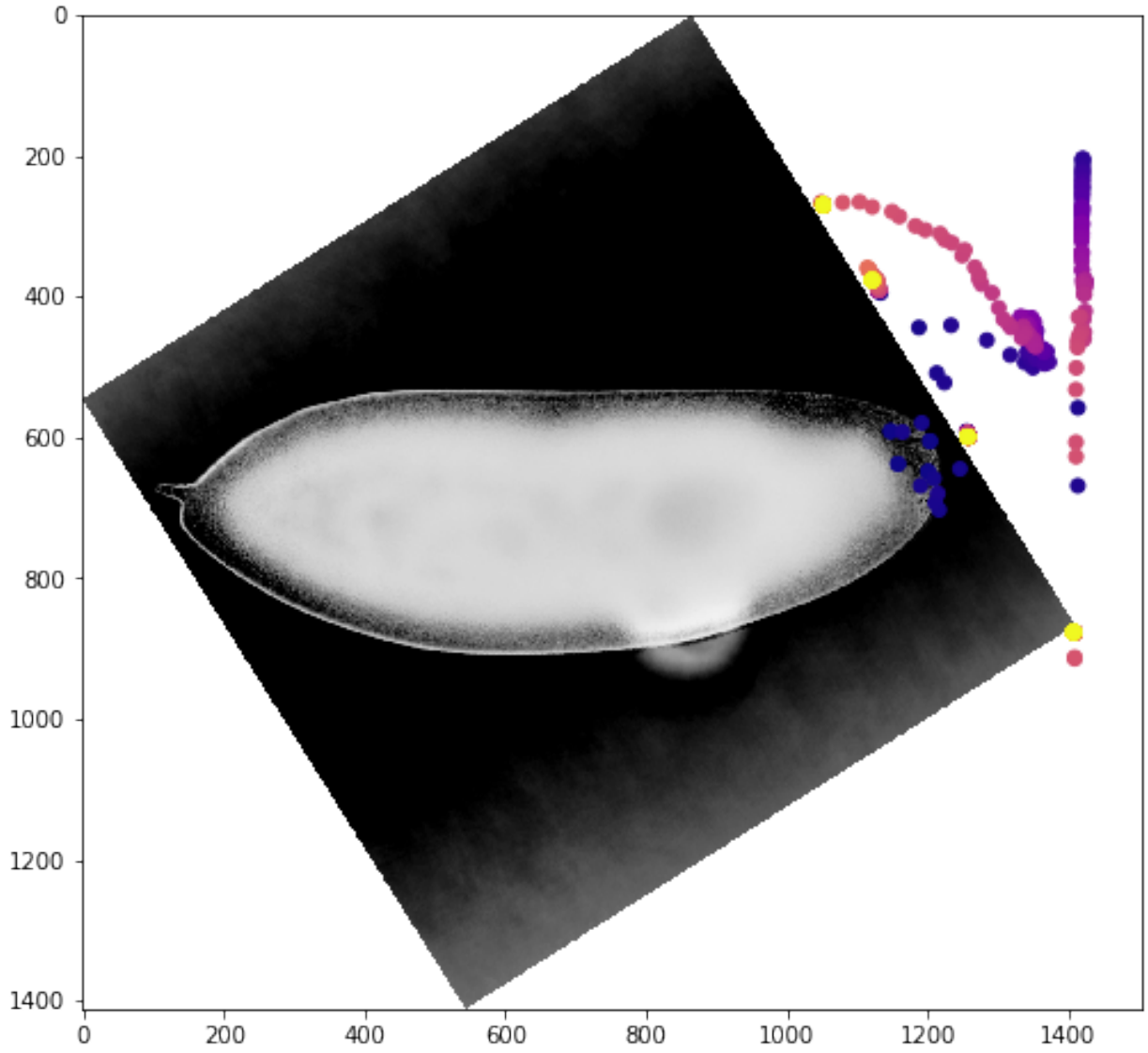

0
200
400
600
800
1000
0
200
400
600
800
1000


0
200
400
600
800
1000
1200
0
200
400
600
800
1000
1200


−200
0
200
400
600
800
1000
0
200
400
600
800
1000


0
200
400
600
800
1000
1200
1400
0
200
400
600
800
1000
1200
1400


−400
−200
0
200
400
600
800
1000
0
200
400
600
800
1000
1200



Dynamic Track Visualization

In order to get a better sense of how the tracks relate to movement in the embryo, we'll take advantage of the movie that is automatically generated by `OpticalFlowOutput.m`. We can read the avi file in as a 3D array using *`gbeflow.load_avi_as_array()`*. We can then plot tracks on top of each frame from the avi using *`gbeflow.make_track_movie()`*. This second functions saves the new 3D array as a tiff stack, which can be opened in Fiji and converted to an avi if necessary.

```
for f in fs:
    try:
        gbeflow.make_track_movie(f+'.avi',
                                 tracks[tracks.f==f],
                                 c='r',
                                 name=f+'_tracks')
    except:
        print('Failed:',f)
```

The python framework of `try except` is convenient here because the avi output from `OpticalFlowOutput.m` does not always work and can result in the creation of a file without any data. By using `try except`, we will get a report if any particular file fails, but our loop will continue running to finish the remaining files.

1.5.5 Detecting Mesoderm Invagination

Calculate the time when mesoderm invagination occurs in order to align samples in time and to start point tracking at the start of germband invagination

```
import numpy as np
import pandas as pd

import matplotlib
matplotlib.use('agg')
import matplotlib.pyplot as plt

import tifffile
import av

import tqdm

import gbeflow
```

```
fs = ['20180108_htl_glc_sc11_mmzm_rotate_brt',
 '20180108_htl_glc_sc2_mmzm_wp_rotate_brt',
 '20180108_htl_glc_sc9_mmzp_rotate_brt',
 '20180110_htl_glc-CreateImageSubset-01_sc10_wt_rotate_brt',
 '20180110_htl_glc-CreateImageSubset-02_sc11_htl_rotate_brt',
 '20180110_htl_glc_sc14_mmzp_rotate_brt',
 '20180110_htl_glc_sc15_mmzm_rotate_brt',
 '20180110_htl_glc_sc6_mmzm_rotate_brt',
 '20180112_htlglc_tl_sc11_mmzp_rotate_brt',
 '20180112_htlglc_tl_sc4_resille_rotate_brt']
```

Select a single sample as a test case

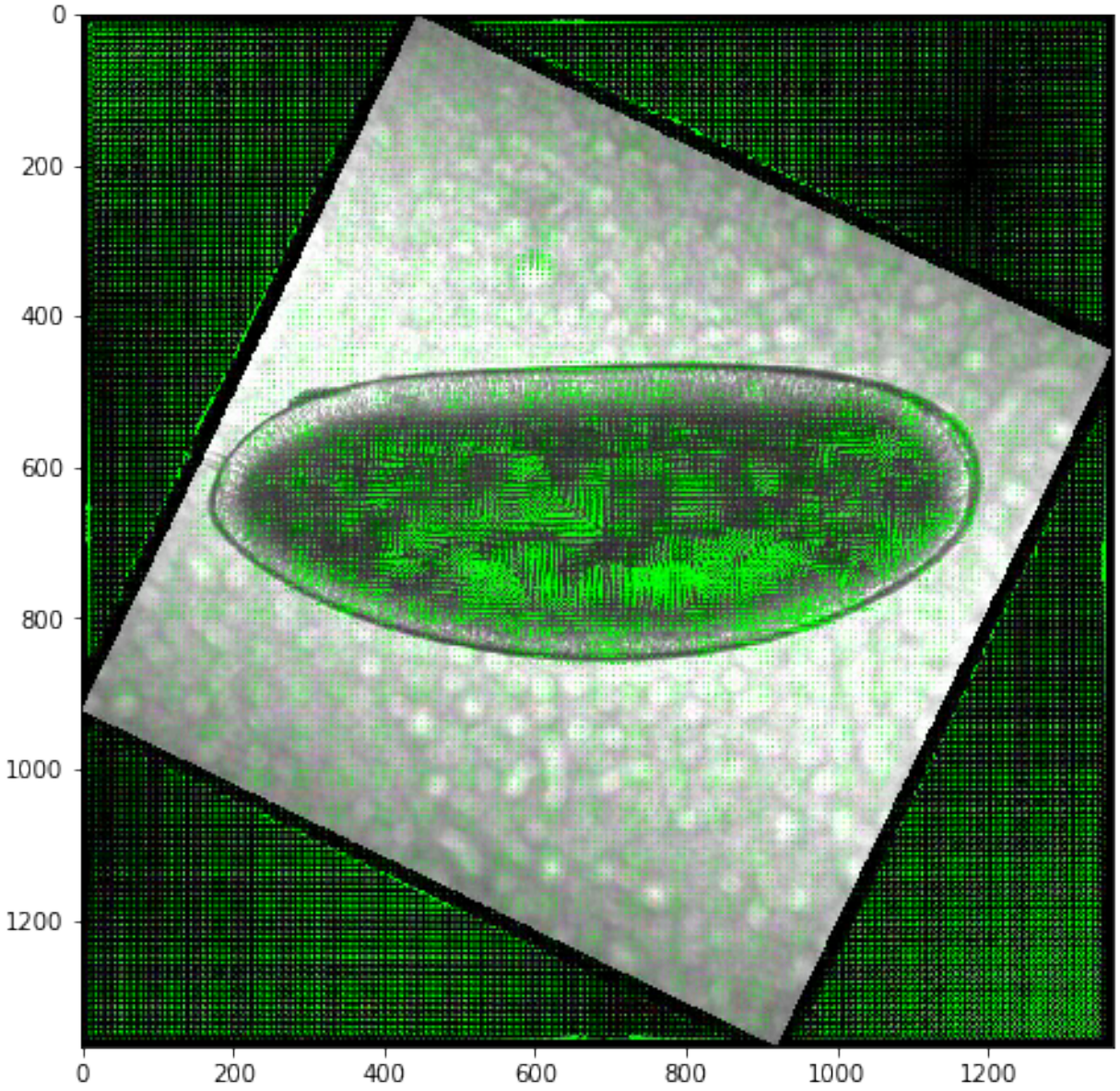
```
f = '20180110_htl_glc_sc14_mmzp_rotate_brt'
```

Load vector data and movie mesoderm_figs/output

```
# vimg = gbeflow.load_avi_as_array(f+'.avi')
vf = gbeflow.VectorField(f)
```

```
expected = 35
```

```
fig,ax = plt.subplots(figsize=(10,8))
ax.imshow(vimg[expected])
```



Test metrics for detecting mesoderm invagination

Sum of y component of vectors over time

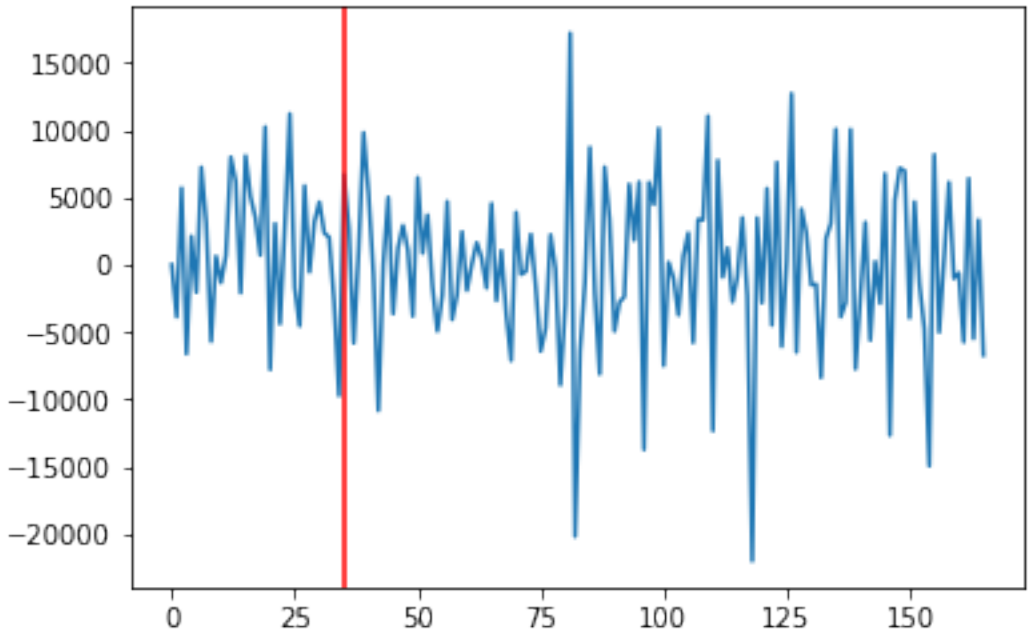
```
vf.vy.shape
```

```
(166, 270, 270)
```

```
ysum = np.sum(vf.vy,axis=(1,2))
```

```
fig,ax = plt.subplots()
ax.plot(ysum)
ax.axvline(expected,c='r')
```

```
<matplotlib.lines.Line2D at 0x109358d30>
```

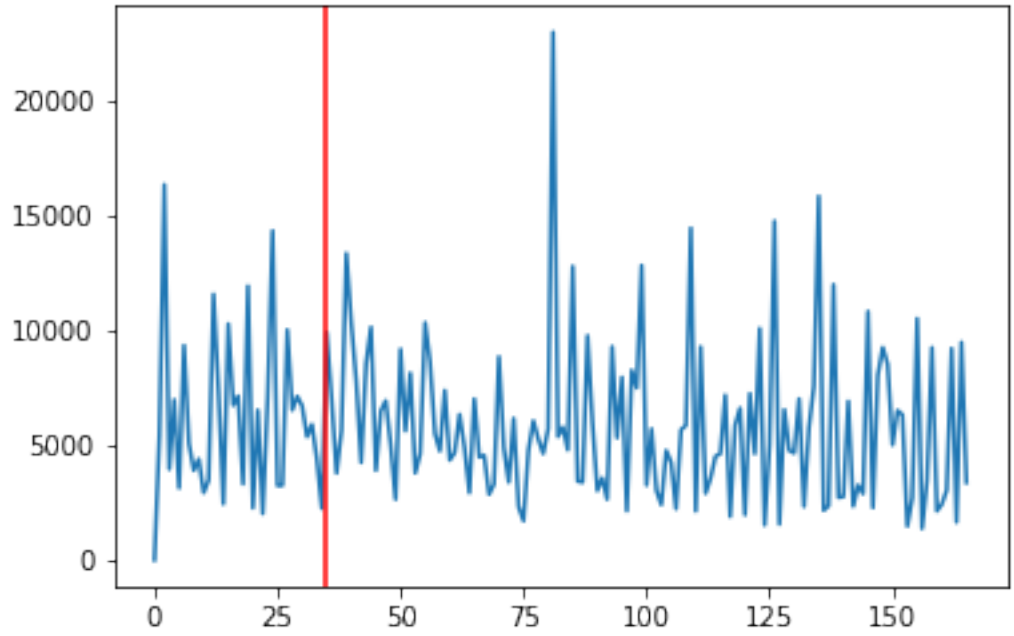


Try looking at the sum of only positive vy components

```
vfpos = np.copy(vf.vy)
vfpos[vfpos<0] = 0
```

```
ysum = np.sum(vfpos,axis=(1,2))
fig,ax = plt.subplots()
ax.plot(ysum)
ax.axvline(expected,c='r')
```

```
<matplotlib.lines.Line2D at 0x1d21e1fc50>
```

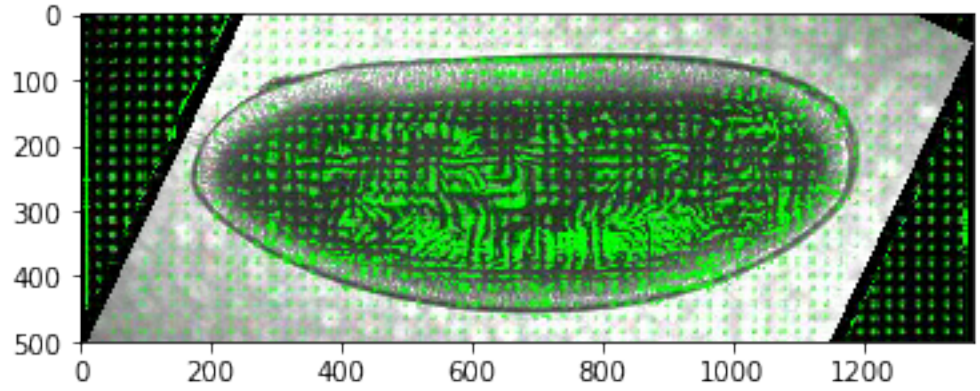


What about within an roi

```
roi = np.s_[:,400:900,:]
```

```
fig,ax = plt.subplots()
ax.imshow(vimg[roi][35])
```

```
<matplotlib.image.AxesImage at 0x11579f0b8>
```




```
np.interp(a, (a.min(), a.max()), (-1, +1))
```

```
roirange = np.interp([400,900], (0,1368), (0,270))
roirange
```

```
array([ 78.94736842, 177.63157895])
```

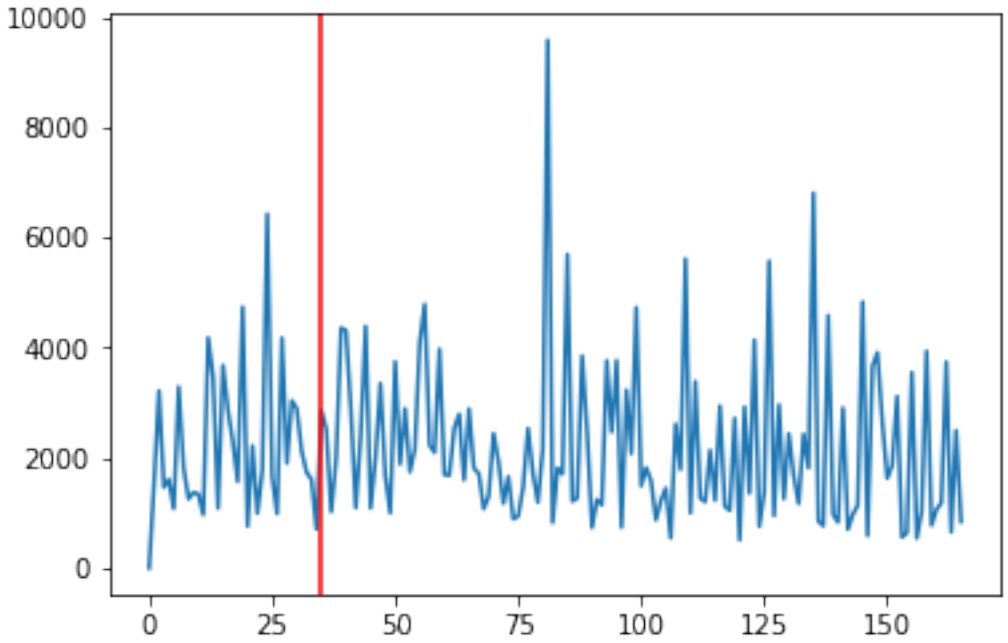
```
vroi = np.s_[:,int(roirange[0]):int(roirange[1]),:]
```

```
vfpos[vroi].shape
```

```
(166, 99, 270)
```

```
ysum = np.sum(vfpos[vroi],axis=(1,2))
fig,ax = plt.subplots()
ax.plot(ysum)
ax.axvline(expected,c='r')
```

```
<matplotlib.lines.Line2D at 0x1d239af9e8>
```

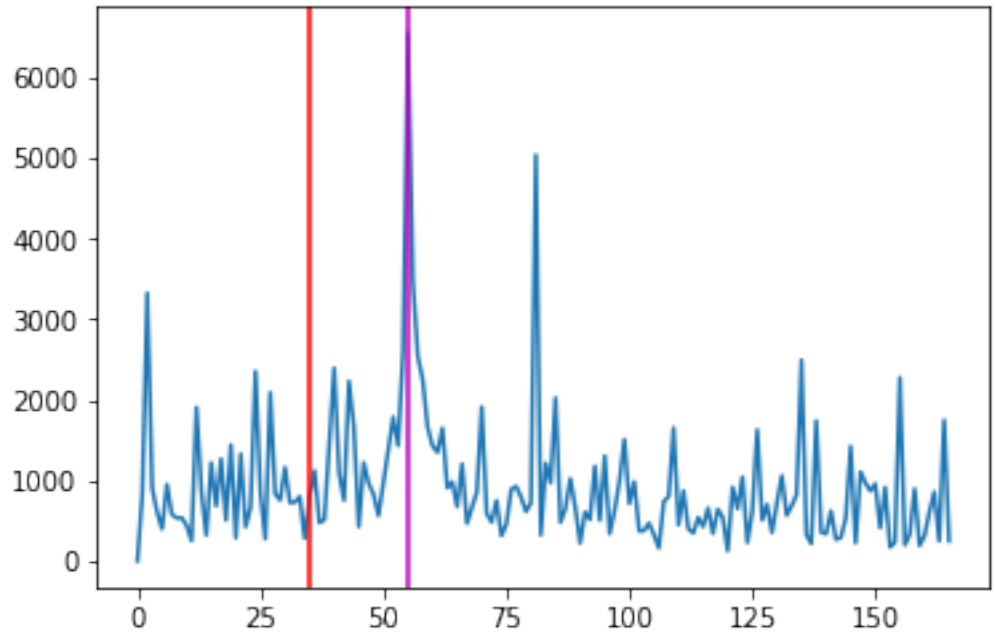


Sum of the squared positive vy

```
ysum = np.sum(np.power(vfpos[vroi],2),axis=(1,2))
```

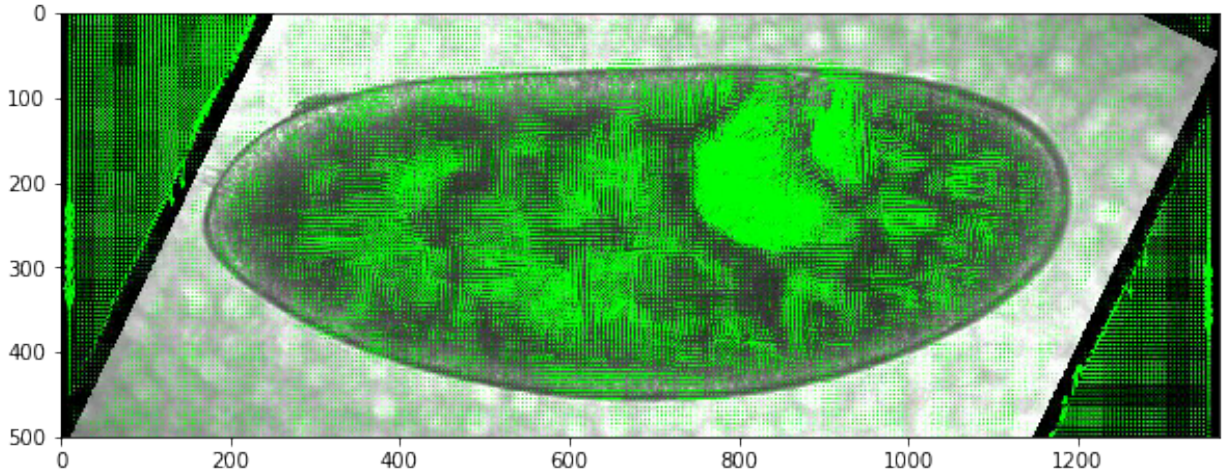
```
fig,ax = plt.subplots()
ax.plot(ysum)
ax.axvline(expected,c='r')
ax.axvline(55,c='m')
```

```
<matplotlib.lines.Line2D at 0x1d24b19eb8>
```




```
fig,ax = plt.subplots(figsize=(10,8))
ax.imshow(vimg[roi][55])
```

```
<matplotlib.image.AxesImage at 0x1d24c678d0>
```



While we are not detecting mesoderm invagination here, this feature does mark germband extension which could be useful in itself.

Let's try this on other samples to see if the feature is consistent

```
Dvimg = {}
for f in fs:
```

(continues on next page)

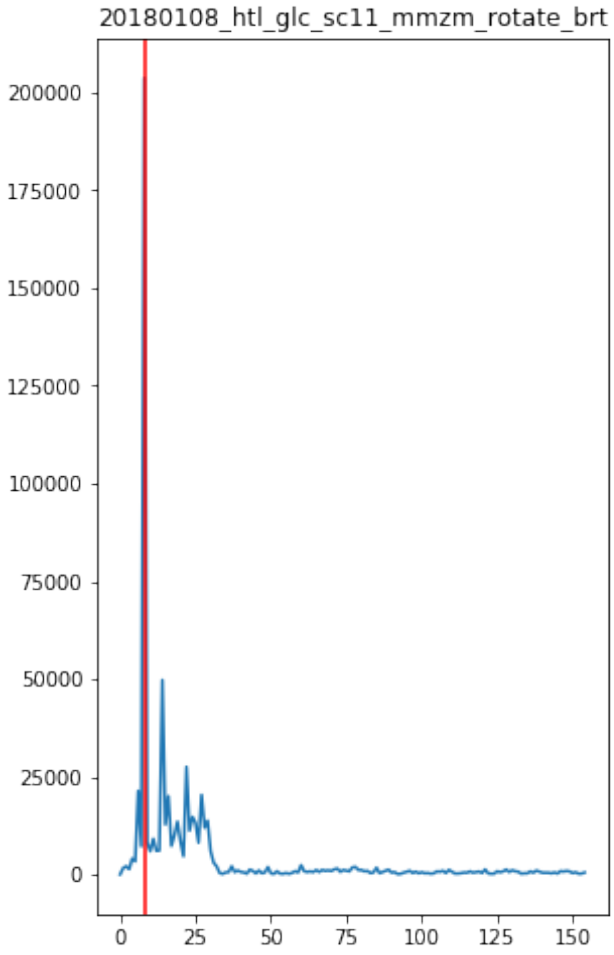
(continued from previous page)

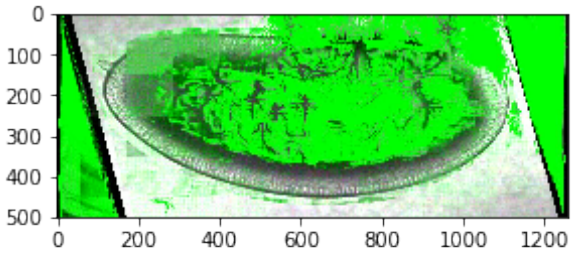
```
    try:
        Dvimg[f] = gbeflow.load_avi_as_array(f+'.avi')
    except:
        print('Video import failed',f)
```

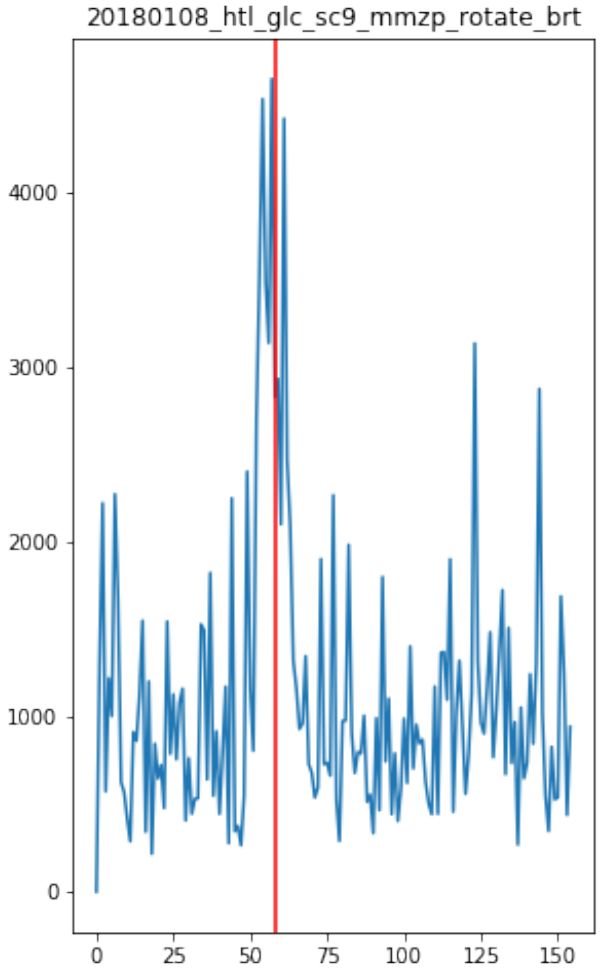
```
Dvf = {}
for f in Dvimg.keys():
    try:
        Dvf[f] = gbeflow.VectorField(f)
    except:
        print('Import failed',f)
```

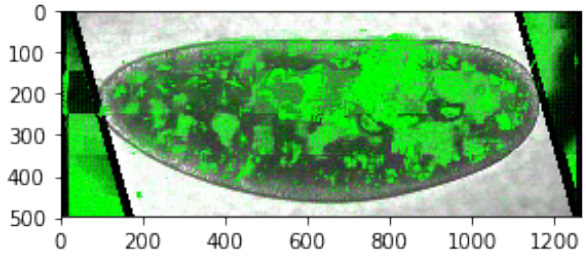
```
Dsum = {}
for f in Dvimg.keys():
    vfpos = np.copy(Dvf[f].vy)
    vfpos[vfpos<0] = 0
    Dsum[f] = np.sum(np.power(vfpos[vroi],2),axis=(1,2))
```

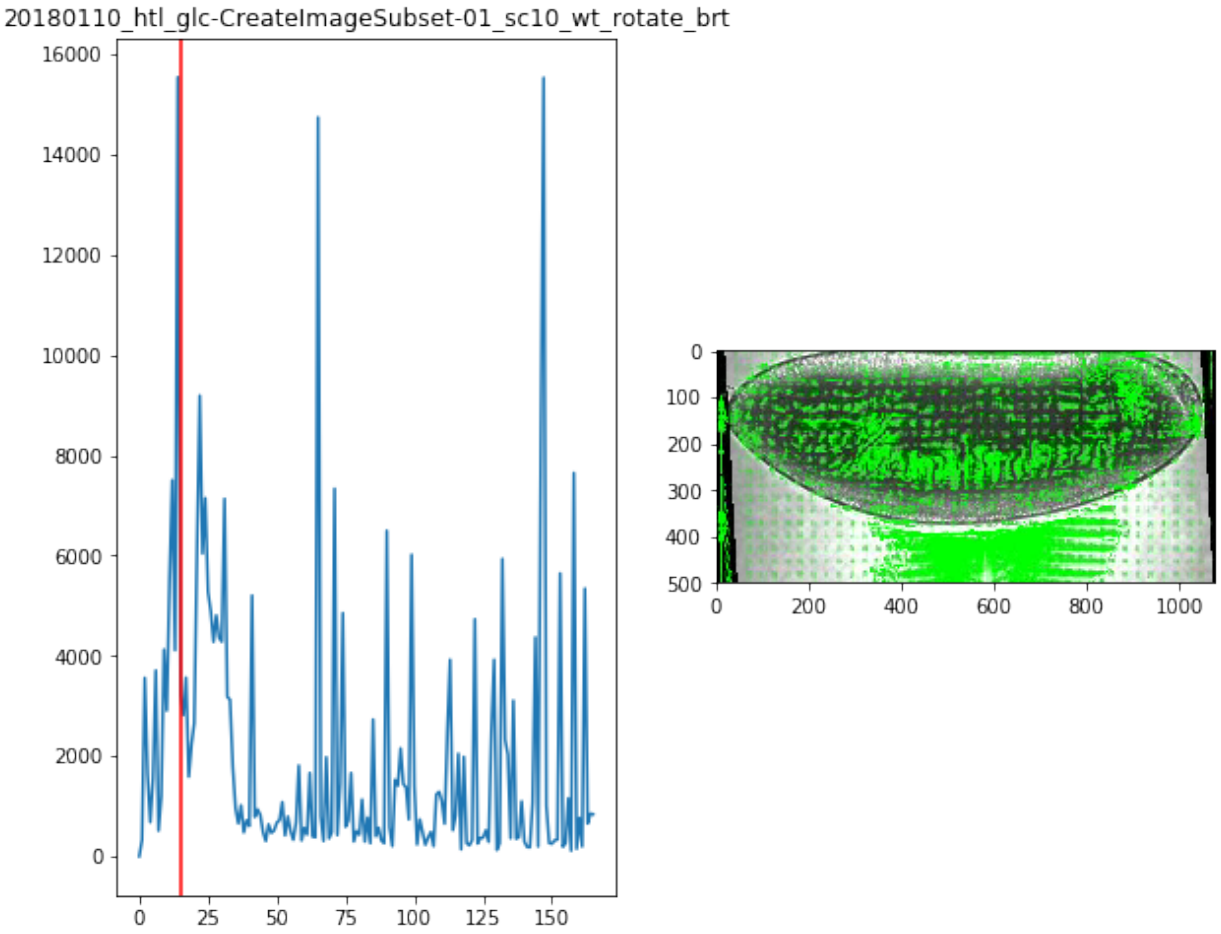
```
for f in Dsum.keys():
    fig,ax = plt.subplots(1,2,figsize=(10,8))
    ax[0].plot(Dsum[f])
    ax[0].axvline(maxsum[f],c='r')
    ax[0].set_title(f)
    ax[1].imshow(Dvimg[f][roi][maxsum[f]])
```

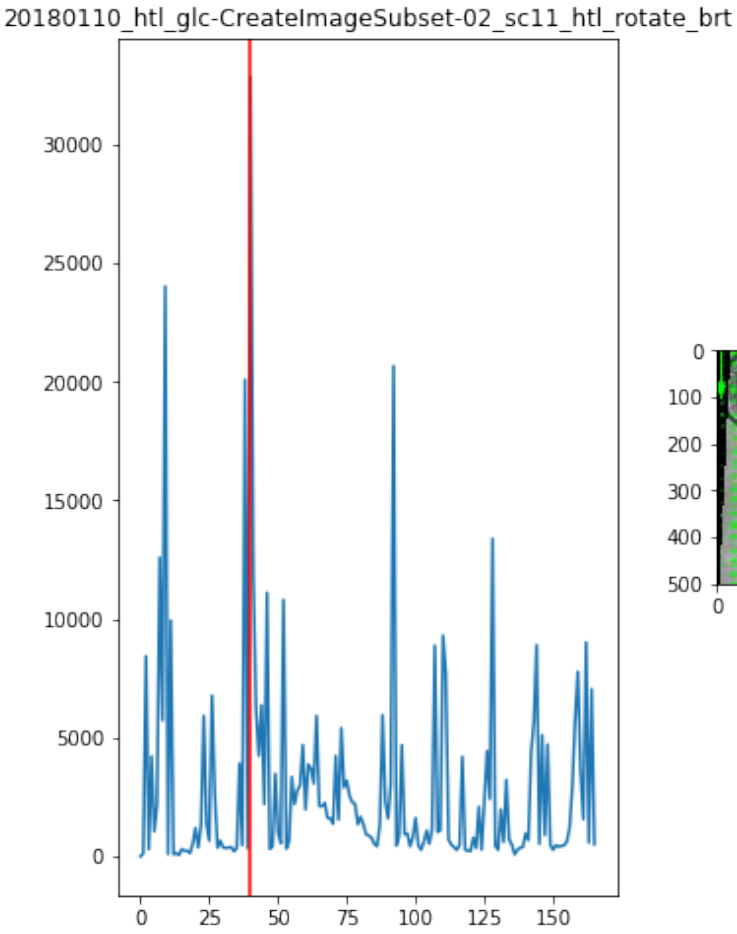

20180108_htl_glc_sc11_mmzm_rotate_brt
200000
175000
150000
125000
100000
75000
50000
25000
0
0
25
50
75
100
125
150

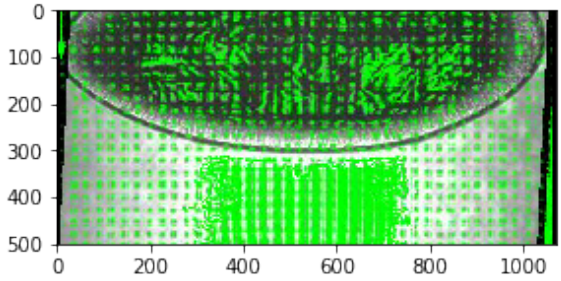

0
100
200
300
400
500
0
200
400
600
800
1000
1200

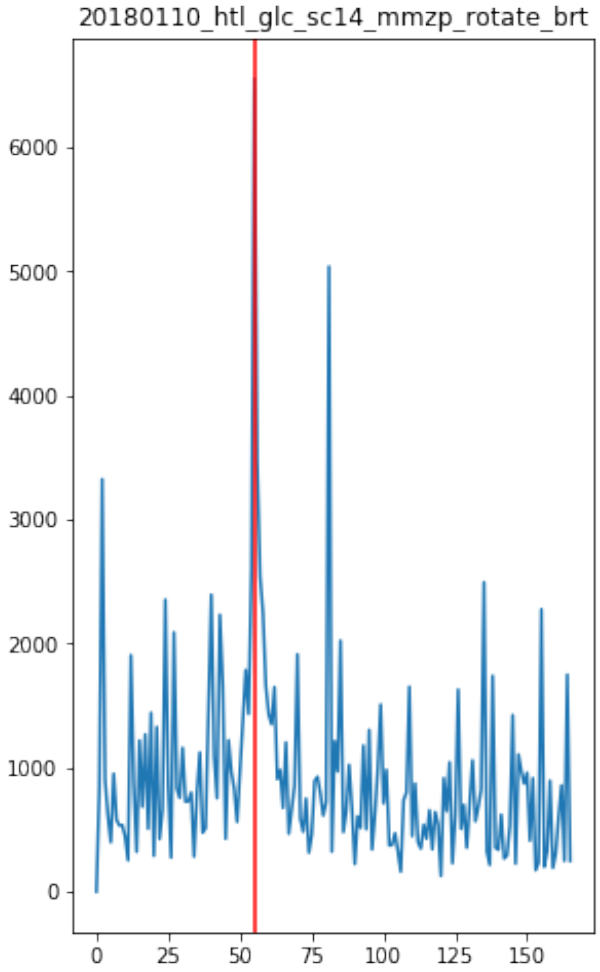

20180108_htl_glc_sc9_mmzp_rotate_brt
4000
3000
2000
1000
0
0
25
50
75
100
125
150

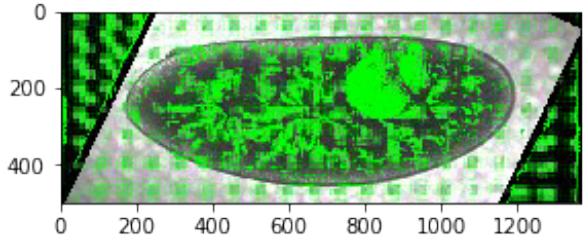

0
100
200
300
400
500
0
200
400
600
800
1000
1200

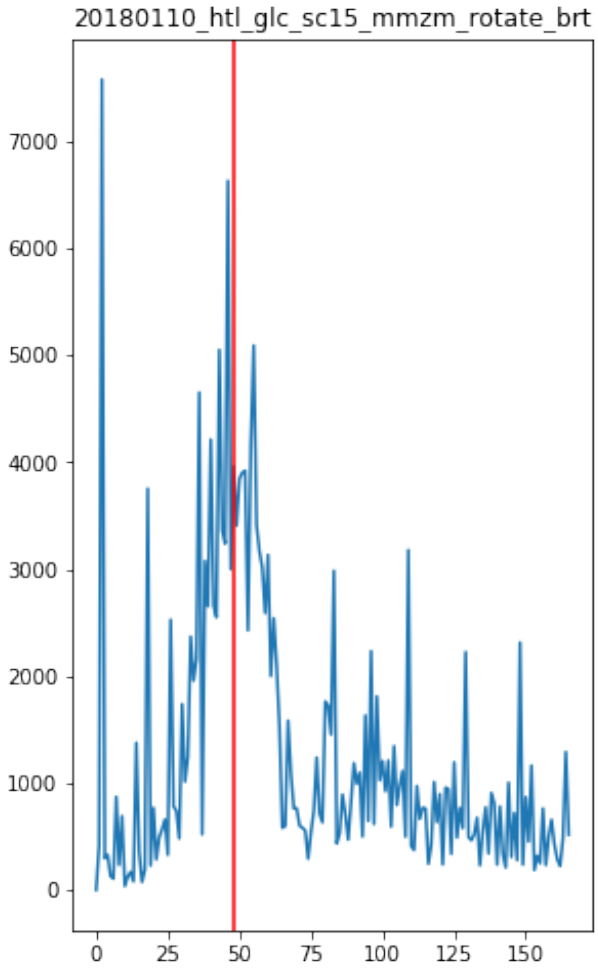

20180110_htl_glc-CreateImageSubset-01_sc10_wt_rotate_brt
16000
14000
12000
10000
8000
6000
4000
2000
0
0
25
50
75
100
125
150
0
100
200
300
400
500
0
200
400
600
800
1000

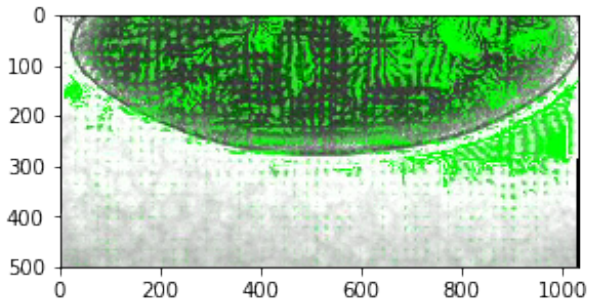

20180110_htl_glc-CreateImageSubset-02_sc11_htl_rotate_brt
30000
25000
20000
15000
10000
5000
0
0
25
50
75
100
125
150

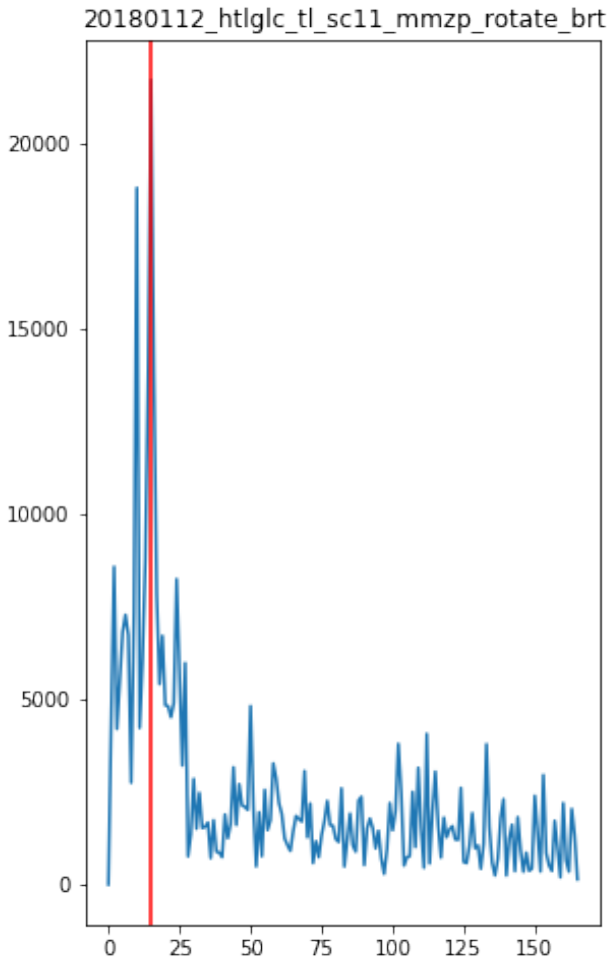

0
100
200
300
400
500
0
200
400
600
800
1000

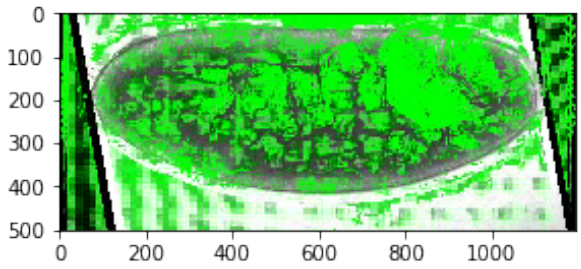

20180110_htl_glc_sc14_mmzp_rotate_brt
6000
5000
4000
3000
2000
1000
0
0
25
50
75
100
125
150


0
200
400
0
200
400
600
800
1000
1200


20180110_htl_glc_sc15_mmzm_rotate_brt
7000
6000
5000
4000
3000
2000
1000
0
0
25
50
75
100
125
150


0
100
200
300
400
500
0
200
400
600
800
1000






```
maxsum = {
    '20180108_htl_glc_sc11_mmzm_rotate_brt':8,
    '20180108_htl_glc_sc9_mmzp_rotate_brt':58,
    '20180110_htl_glc-CreateImageSubset-01_sc10_wt_rotate_brt':15,
    '20180110_htl_glc-CreateImageSubset-02_sc11_htl_rotate_brt':40,
    '20180110_htl_glc_sc14_mmzp_rotate_brt':55,
    '20180110_htl_glc_sc15_mmzm_rotate_brt':48,
    '20180112_htlglc_tl_sc11_mmzp_rotate_brt':15
}
```

For the time being it looks like manually selecting the timepoint of mesoderm invagination will be more expedient.

Check manual assignments

```
tpoints = pd.read_csv('mesoderm_invagination.csv')
tpoints
```

```
tpoints[tpoints['File']==f+'.avi'].values[-1,-1]
```

```
62
```

```
for f in Dvimg.keys():
    fig,ax = plt.subplots(figsize=(10,8))
    t = tpoints[tpoints['File']==f+'.avi'].values[-1,-1]
    ax.imshow(Dvimg[f][t])
    ax.set_title(f)
```

Test track interpolation starting at mesoderm invagination

```
tracks = pd.read_csv('20181128-tracking.csv')
tracks.head()
```

```
starts = tracks[tracks.t==0][['f','x','y']]
```

```
trange = range(maxsum[f],np.max(vf.tval))
(trange)
```

```
xpos = [x0]*(maxsum[f]+1)
ypos = [y0]*(maxsum[f]+1)
```

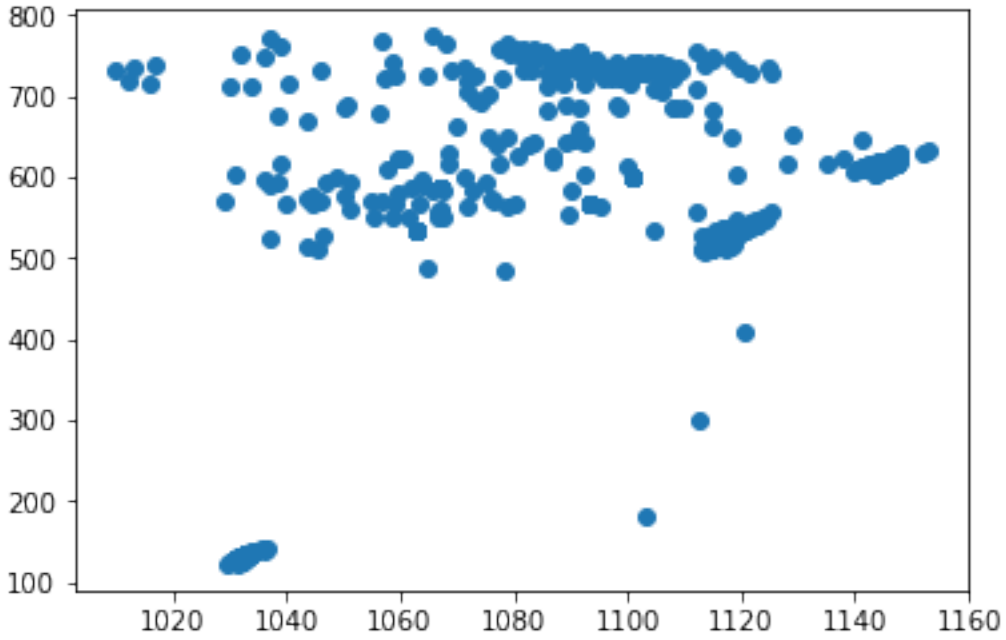
```
for t in trange:
    dx = vf.Ldx[t].ev(xpos[t],ypos[t])
    dy = vf.Ldy[t].ev(xpos[t],ypos[t])

    xpos.append(xpos[t] + dx*60)
    ypos.append(ypos[t] + dy*60)
```

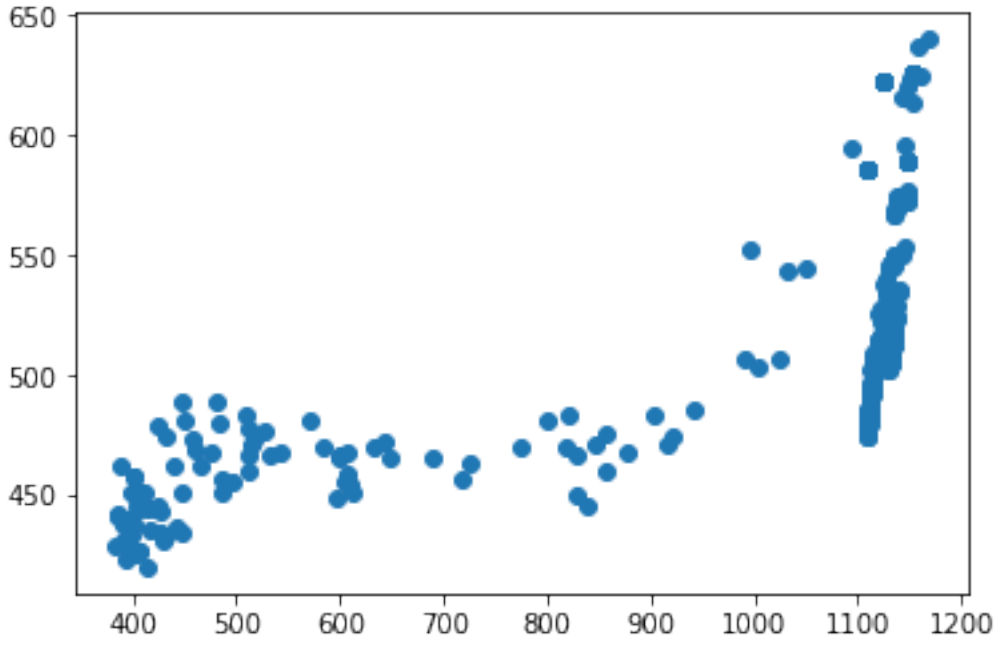
```
track = np.array([xpos,ypos])
trackdf = pd.DataFrame({'x':track[0,:],'y':track[1,:],'t':vf.tval,
                        'track':[i]*track.shape[-1],
                        'name':['test']*track.shape[-1]})
```

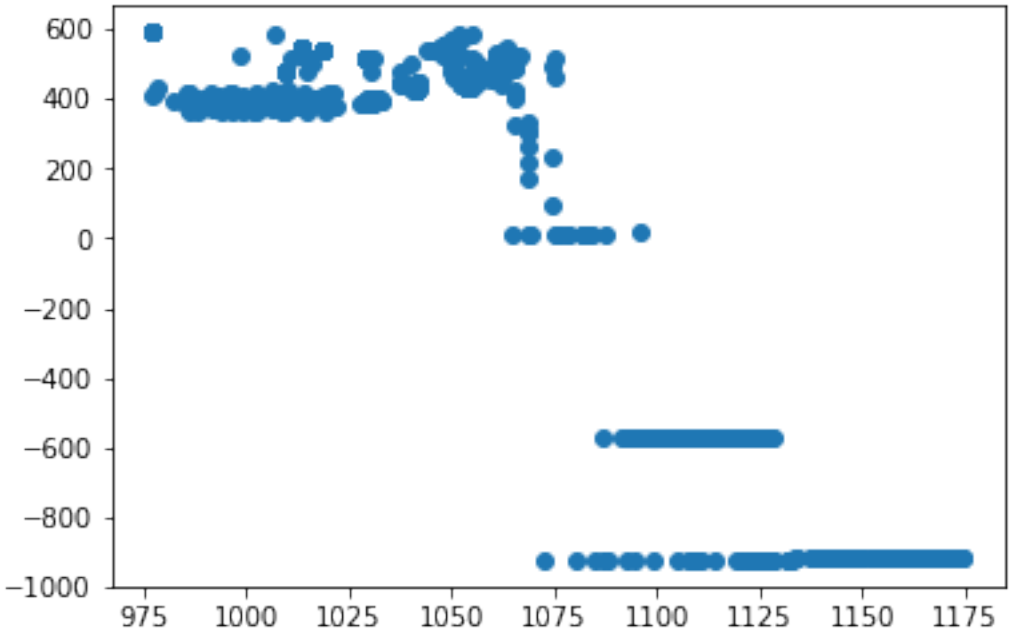
Track visualization

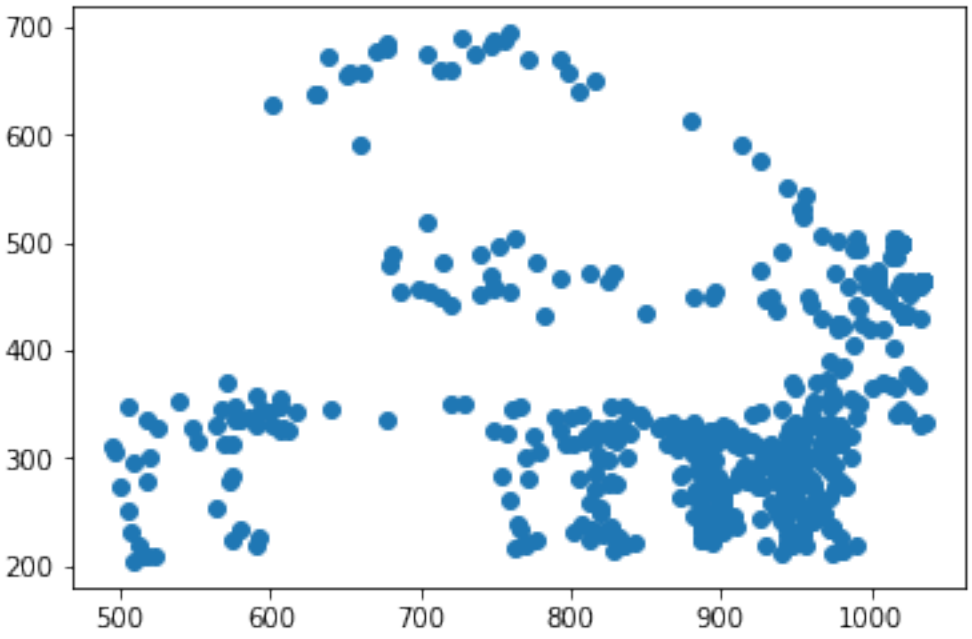
```
tracks = pd.concat(Ltrack,keys=list(Dvf.keys())
                  ).reset_index(
                  ).drop(columns=['level_1']
                  ).rename(columns={'level_0':'file'})
```

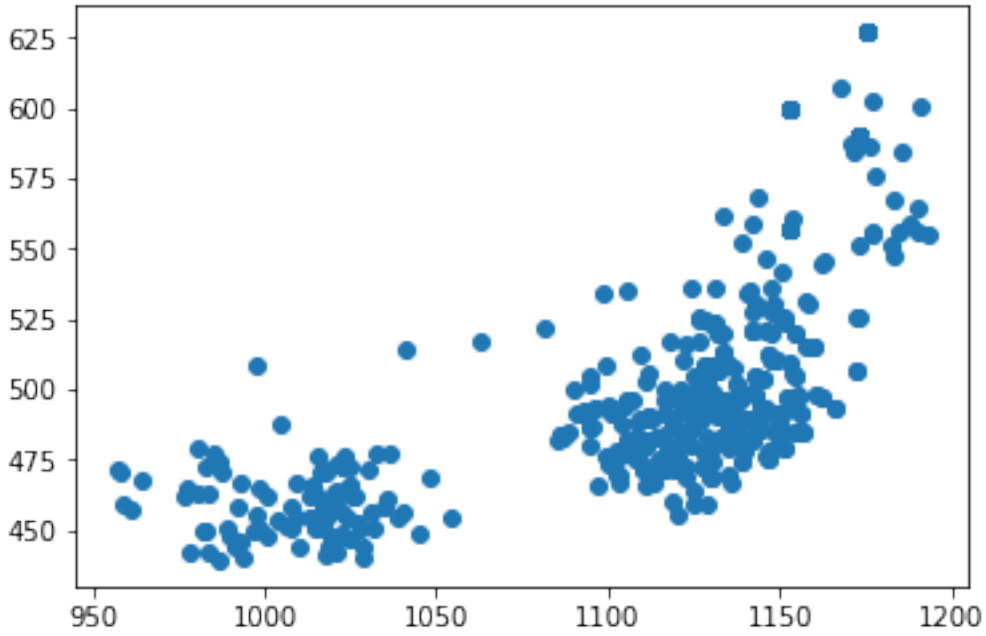
```
for f in tracks['file'].unique():
    fig,ax = plt.subplots()
    sb = tracks[tracks['file']==f]
    ax.scatter(sb.x,sb.y)#,c=sb.t,cmap='plasma')
```

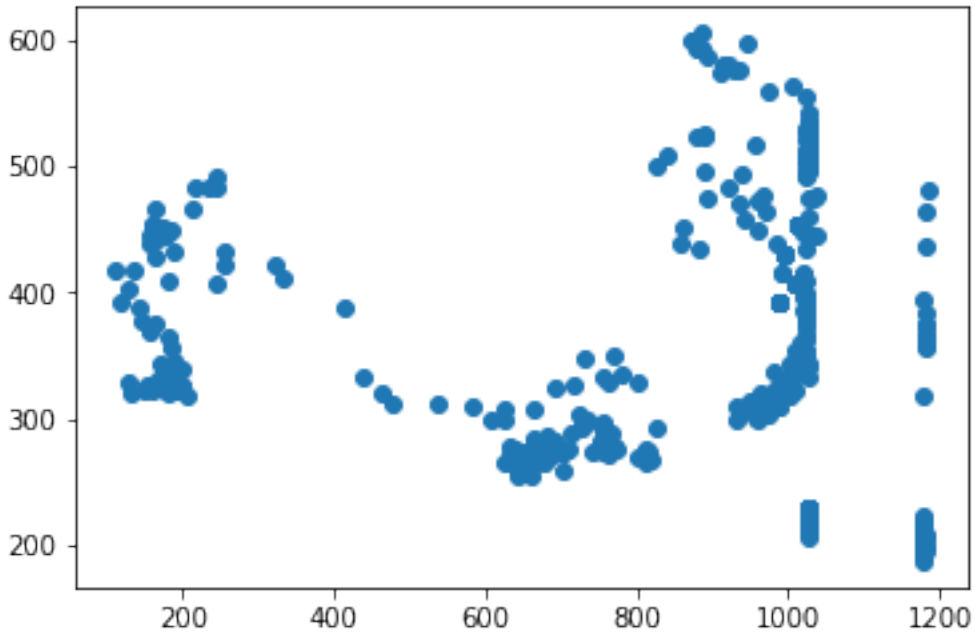

800
700
600
500
400
300
200
100
1020
1040
1060
1080
1100
1120
1140
1160

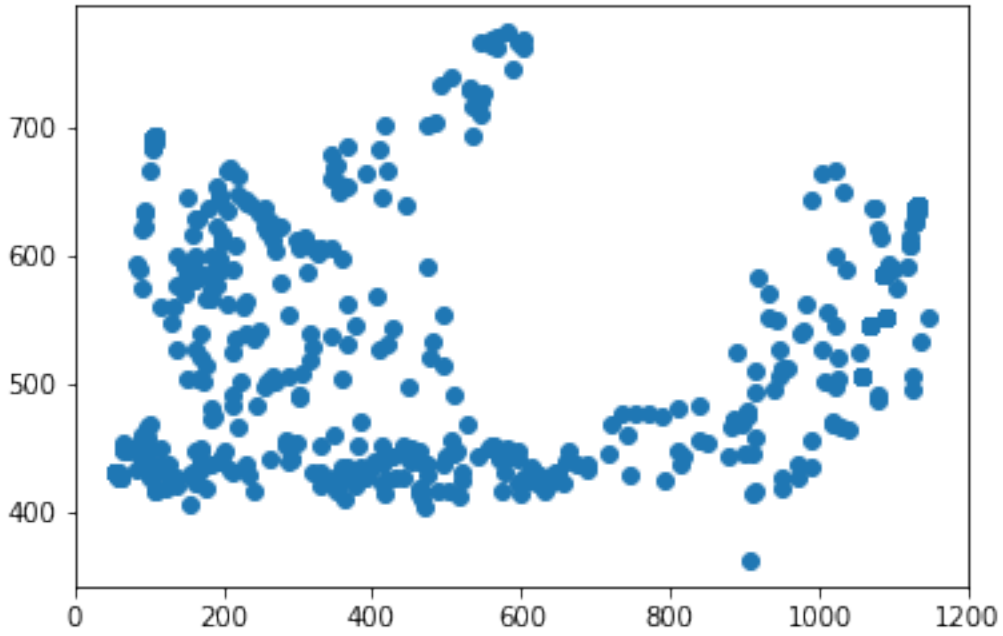

650
600
550
500
450
400
500
600
700
800
900
1000
1100
1200


600
400
200
0
−200
−400
−600
−800
−1000
975
1000
1025
1050
1075
1100
1125
1150
1175


700
600
500
400
300
200
500
600
700
800
900
1000


625
600
575
550
525
500
475
450
950
1000
1050
1100
1150
1200


600
500
400
300
200
200
400
600
800
1000
1200




```
for f in tracks['file'].unique():
    gbeflow.make_track_movie(Dvimg[f],tracks[tracks.file==f],c='r',
                             name='20181202_'+f+'_tracks')
```

Attempting to start interpolating the tracks at the time of mesoderm invagination does not appear to improve the results of the tracking since there is still a large fraction that wanders out of the frame entirely.

CHAPTER 2

Indices and tables

- modindex
- search

Python Module Index

g

Index

Symbols

A

C

D

G

I

L

M

N

P

R

S

T

V

W

X

Y