Appendix B

Spinner processing code

B.1 Workflow

The data processing workflow pictured in Figure B.1 was used to process microscopy of tethered spinning cells. The scripts and associated files can be found at the end of this Appendix in condensed form. The most recent versions of the code files can be found online in the following Git repository: `https://github.com/zajdel/Spinners`. Three programs provide the majority of processing:

1. **StitchStacks.ijm** - ImageJ macro that converts image sequence to binarized stack
2. **1_create_traces.py** - Python script that produces heading traces, using user assistance to identify spinning cells
3. **2_quality_control.py** - Python script that annotates spinning direction, using user assistance to select direction thresholds

Streams of tethered spinner cells are saved as a series of .tif images that are to be stitched together into a stack. At the same time, files are filtered and then binarized for black cell bodies on a white background. This is accomplished with the ImageJ macro **StitchStacks.ijm**. Then, **1_create_traces.py** is used to select the spinner cells and extract heading traces. The program takes the file location of the binarized image as an input. The command line function call looks like this:

```
>> 1_create_traces.py folder\stackname
```

The program averages several hundred frames from the binarized stack, resulting in a circle surrounding the tether point of each spinning cell as shown in Figure B.2. These mean circles are overlaid on top of a looping video of the first few hundred frames of the stack. The user selects an estimated tether point on the mean image and the program searches for the darkest pixel in the region immediately surrounding this selection. The result is a .csv file with the following format:

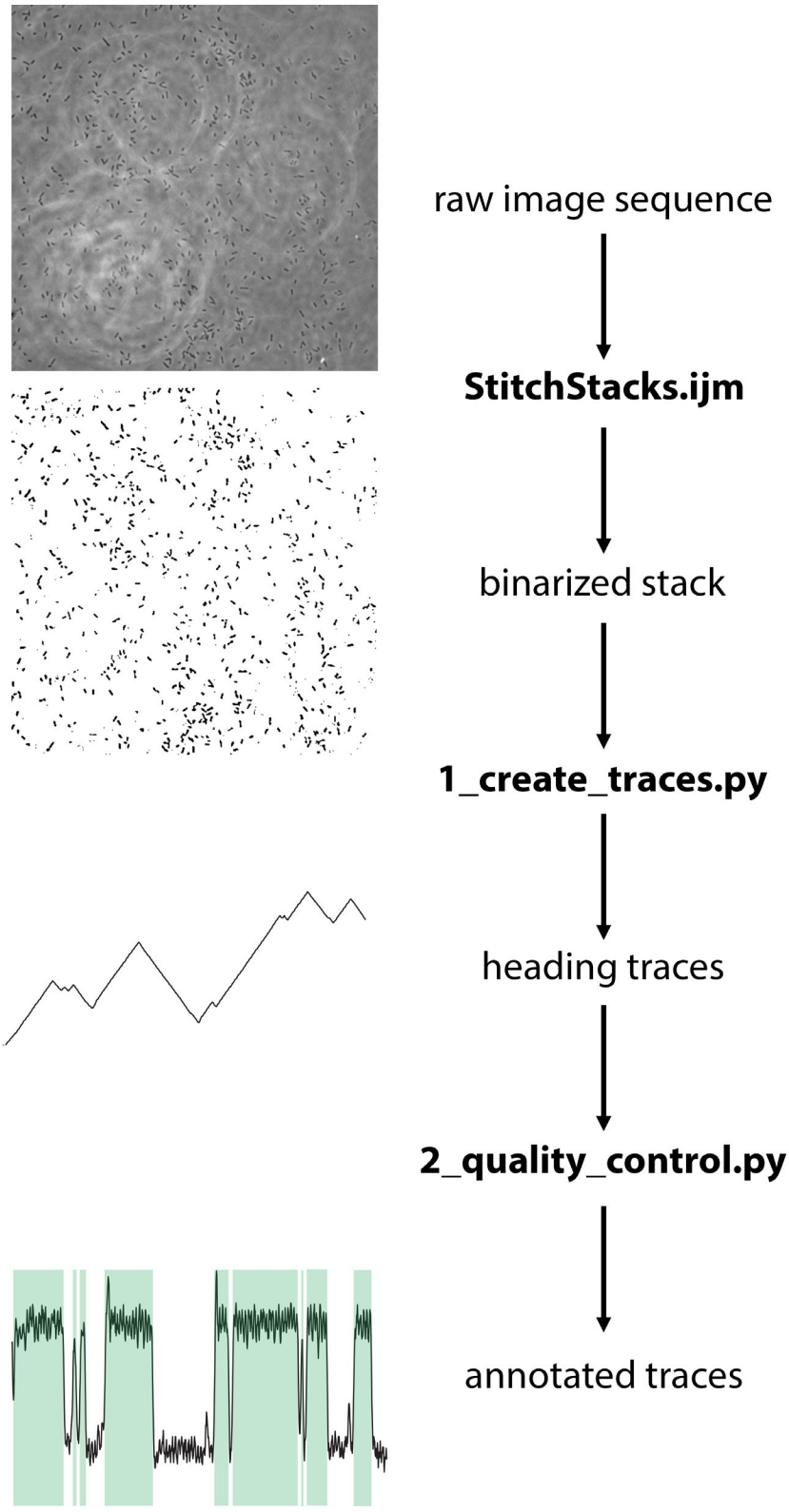

Figure B.1: Spinner signal processing workflow.

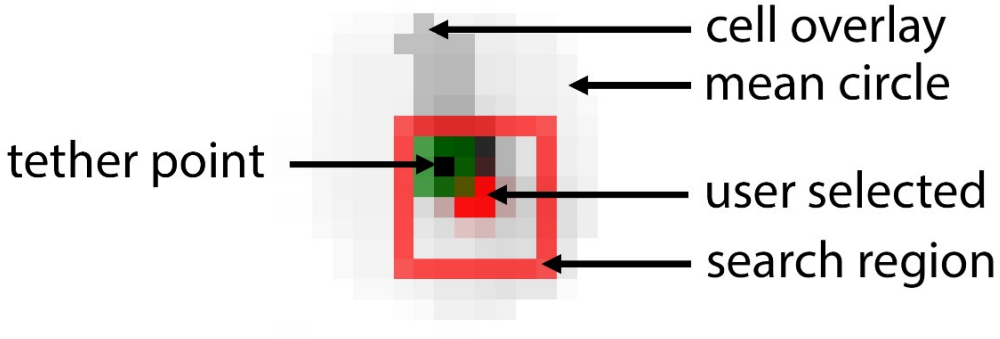

Figure B.2: Detecting the tether point of a spinner cell with **1_create_traces.py**. This is a magnified screen capture of the program at runtime. The solid red square marks the pixel that the user selects, and the program searches within the larger red square outline for the darkest pixel of the mean image, marked in green.

```
x_coord,  y_coord,     tL,     tH,  heading1,  heading2,  heading3, ...
    298,      192,     -1,     -1,   -2.3562,    -0.245,   -0.6435, ...
    255,      114,     -1,     -1,    -0.245,    -0.588,   -1.1071, ...
```

For each selected cell, the (x,y) coordinates of the tether point, -1 is written to both the low direction threshold (tL) and high direction threshold (tH), and the computed headings are all written to the .csv. This file is further processed by **2_quality_control.py**.

```
>> 2_quality_control.py folder\csvname
```

This program sets the high and low thresholds for every cell's velocity, producing a new .csv file with the user-selected thresholds. If a user determines that a given trace should be excluded from analysis, a zero is written to both thresholds. The result is a .csv file with the following format:

```
x_coord,  y_coord,     tL,     tH,  heading1,  heading2,  heading3, ...
    298,      192,   0.10,  -0.36,   -2.3562,    -0.245,   -0.6435, ...
    255,      114,      0,      0,    -0.245,    -0.588,   -1.1071, ...
```

Further analysis can be completed now that the thresholds for direction labeling have been selected and problematic traces have been flagged.

B.2 StitchStacks.ijm

```
/*
-----------------------------------------------------
```

```
StitchStacks.ijm
ImageJ Macro
Purpose: open a folder consisting of frames from a
spinner stream, then filter, segment, and save as zip
-------------------------------------------------------
*/

// pick out the directory that
dir = getDirectory("Pick date to process!");

// retrieve all frames from the folder
list = getFileList(dir);

for (i=0; i<list.length; i++) {
   // if this is a folder, concatenate the images inside!
   if (endsWith(list[i], "/"))
   {
        // open the frames
      experiment = list[i];
      list2 = getFileList(dir+experiment);
      tifname=list2[0];
      run("Image Sequence...", "open="+dir+list[i]+tifname+" sort use");

        // process and binarize the image
        run("8-bit");
      run("Smooth","stack");
      run("Unsharp Mask...", "radius=1 mask=0.90 stack");
      run("Auto Threshold...", "method=Default white stack");

        // save as zip
      title = substring(experiment,0,lengthOf(experiment)-1);
      saveAs("ZIP", dir+title+".zip");
      close();
   }
}
```

B.3 utilities.py

```
# Import packages
from __future__ import division, unicode_literals # , print_function
import argparse
import numpy as np
```

```
from matplotlib import animation
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from scipy.signal import medfilt
import pims
```

B.4 1_create_traces.py

```
from __future__ import division
from utilities import *
#  Ex. python 1_create_traces.py 100u_leu1 [TIF]
from PIL import Image
from Queue import Queue

parser = argparse.ArgumentParser(description="Create traces of cells identified
    in a TIF and output to CSV.")
parser.add_argument("source", help="source file [TIF]")
parser.add_argument("dest", nargs="?", help="destination file [CSV]")
parser.add_argument("-v", "--verbose", help="verbose output: display trace for
    every cell", action="store_true")
args = parser.parse_args()

tif_name = args.source + '.tif'
raw_frames = pims.TiffStack(tif_name, as_grey=False)
frames = np.array(raw_frames, dtype=np.uint8)

# use (and/or overwrite) existing file
centers = []
if args.dest:
    csv_name = args.dest + '.csv'
    data = np.loadtxt(csv_name, delimiter=",")
    centers, _, _ = np.hsplit(data, np.array([2, 3]))

# ^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# ^*^*^*^*^*^*^*^*^*^    Getting Mean Image     *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# ^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^

# use only first 500 frames
mean = np.mean(frames[0:500], axis=0)

# ^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# ^*^*^*^*^*^*^*^*^*^  Overlay Mean on Frames  *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
```

```
# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^

overlay = Image.fromarray(mean).convert('RGB')
new_frames = []
for frame in frames:
    frame = Image.fromarray(frame).convert('RGB')
    new_frames.append(np.asarray(Image.blend(frame, overlay, 0.8)))
frameview = new_frames

# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# *^*^*^*^*^*^*^*^*  Show Frames for Center Selection *^*^*^*^*^*^*^*^*^*^*^*^
# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^

# cycle through first 50 frames
N = 50
show_frames = frameview[0:N]

fig, ax = plt.subplots()
im = ax.imshow(show_frames[0], aspect='equal')

# if using existing CSV, show previously selected centers
for center in centers:
    rect = patches.Rectangle((center[0] - 0.5, center[1] - 0.5), 1, 1,
        linewidth=1, edgecolor='r', facecolor='none')
    ax.add_patch(rect)

# convert centers array to list of tuples and create a set from it
selected_points = set(tuple(map(tuple, centers)))


# Error handling on press: find minimum intensity in mean in 5x5 area around
    selected center
def on_press(event):
    if event.xdata and event.ydata:
        x, y = int(round(event.xdata)), int(round(event.ydata))
        print('You pressed {0} at ({1}, {2}) with mean value of
            {3}.'.format(event.button, x, y, mean[y, x]))

        rect = patches.Rectangle((x - 0.5, y - 0.5), 1, 1, linewidth=1,
            edgecolor='r', facecolor='none')
        ax.add_patch(rect)
        roi = [(i, j) for i in range(x - 2, x + 3) for j in range(y - 2, y + 3)
            if 0 < i < mean.shape[1] and 0 < j < mean.shape[0]]
        min_intensity = min([mean[p[::-1]] for p in roi]) # p[::-1] reverses tuple
```

```
        correct_x, correct_y = np.mean([p for p in roi if mean[p[::-1]] ==
            min_intensity], axis=0)
        selected_points.add(
            (correct_x, correct_y) # use set to avoid duplicates being stored.
                (use tuple because can hash.)
        )
        if correct_x != x or correct_y != y:
            print('Corrected green box at ({0}, {1})'.format(correct_x,
                correct_y))
            area = patches.Rectangle((x - 2.5, y - 2.5), 5, 5, linewidth=0.5,
                edgecolor='r', facecolor='none')
            correction = patches.Rectangle((correct_x - 0.5, correct_y - 0.5), 1,
                1, linewidth=0.5, edgecolor='g', facecolor='none')
            ax.add_patch(area)
            ax.add_patch(correction)


fig.canvas.mpl_connect('button_press_event', on_press)


def init():
    im.set_data(show_frames[0])


def animate(i):
    im.set_data(show_frames[i % N])
    return im


anim = animation.FuncAnimation(fig, animate, init_func=init, interval=100)
plt.show()

num_selected_points = len(selected_points)

# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# *^*^*^*^ Post-Processing: Calculate Angle and Generate Traces *^*^*^*^*^*^*^
# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^

# maximum distance a pixel can be to be considered part of cell defined by some
    chosen center
MAX_DISTANCE = 8

num_frames = len(frames)
```

```
def euclidean_distance(p1, p2):
    return np.linalg.norm(np.asarray(p1) - np.asarray(p2))


def find_furthest_points(center, frame):
    """Given a center and a frame, computes furthest connected points to center
       with distance less than MAX_DISTANCE"""
    # get all points connected to center ("cell"), find furthest points
    nearest_pixel_to_center = (int(round(center[0])), int(round(center[1])))
    fringe = Queue()
    fringe.put(nearest_pixel_to_center)
    cell = set()
    cell.add(nearest_pixel_to_center)
    marked = set()
    marked.add(nearest_pixel_to_center)

    # modified breadth-first search
    while not fringe.empty():
        p = fringe.get()
        p1 = (p[0] - 1, p[1])
        p2 = (p[0] + 1, p[1])
        p3 = (p[0], p[1] - 1)
        p4 = (p[0], p[1] + 1)
        p5 = (p[0] - 1, p[1] - 1)
        p6 = (p[0] - 1, p[1] + 1)
        p7 = (p[0] + 1, p[1] - 1)
        p8 = (p[0] + 1, p[1] + 1)
        for p in [p1, p2, p3, p4, p5, p6, p7, p8]:
            if (p not in marked
                    and 0 <= p[0] < len(frames[frame][1])
                    and 0 <= p[1] < len(frames[frame][0])
                    and euclidean_distance(center, p) <= MAX_DISTANCE
                    and frames[frame][p[1], p[0]] == 0):
                marked.add(p)
                cell.add(p)
                fringe.put(p)

    cell = list(cell)
    max_dist = max([euclidean_distance(p, center) for p in cell])
    return [p for p in cell if euclidean_distance(p, center) == max_dist]


# calculate angle by using furthest point from center
```

```
wrapped_traces = []
for center in selected_points:
    ellipses = []
    trace = []
    for i in range(num_frames):
        furthest_points = find_furthest_points((center[0], center[1]), i)
        if len(furthest_points) > 1:
            if len(trace):
                # take point whose angle is closest to previous angle
                furthest_point = min(furthest_points, key=lambda x: abs(trace[i -
                    1] % (2 * np.pi) - np.arctan2(x[0] - center[0], center[1] -
                    x[1]) % (2 * np.pi)))
            else:
                # TODO: what to do if first frame is ambiguous
                furthest_point = furthest_points[0]
        else:
            furthest_point = furthest_points[0]
        # define angle to increase positively clockwise
        ang = np.arctan2(center[1] - furthest_point[1], furthest_point[0] -
            center[0])
        trace.append(ang)

    # add wrapped trace to CSV output
    # prepend center_x, center_y, -1, -1 (unverified status)
    wrapped_traces.append(np.append([center[0], center[1]], np.append([-1, -1],
        trace)))

    if args.verbose:
        # unwrap trace and apply 1D median filter (default kernel size 3)
        unwrapped = medfilt(np.unwrap(np.asarray(trace[2:])))

        plt.xlabel('Frame', fontsize=20)
        plt.ylabel('Angle', fontsize=20)
        plt.title('Trace ({0}, {1})'.format(center[0], center[1]), fontsize=20)
        plt.plot(unwrapped, 'r-', lw=1)
        plt.grid(True, which='both')
        plt.show()

# output: center_x, center_y, status, status, trace
np.savetxt(args.dest or args.source + ".csv", wrapped_traces,
    fmt=','.join(["%.4f"] * 2 + ["%i"] * 2 + ["%.4f"] * num_frames))
```

B.5 2_quality_control.py

```
from utilities import *
#  Ex. python 2_quality_control.py 1mM_asp1
from math import sin, cos
from matplotlib.widgets import Button
from matplotlib.widgets import Slider

parser = argparse.ArgumentParser(description="Perform quality control on
    generated traces and/or determine thresholds for switches.")
parser.add_argument("source", help="source file [CSV]")
parser.add_argument("frames", nargs="?", help="frames [TIF]")
parser.add_argument("-d", "--dest", help="destination file [CSV]")
parser.add_argument("-t", "--type", type=int, choices=[0, 1, 2, 3], default=2,
                    help="""type of quality control: 0 - show trace,
                                                     1 - show reconstructed cells
                                                         overlaid on actual video,
                                                     2 - show velocity graph
                                                         processed from trace with
                                                         manual thresholding,
                                                     3 - show velocity graph
                                                         processed from trace with
                                                         automated threhsolding"""
                    )
args = parser.parse_args()

if args.frames is None and args.type == 2:
    parser.error("frames required when type is 1")

if args.frames:
    tif_name = args.frames + '.tif'
    raw_frames = pims.TiffStack(tif_name, as_grey=False)
    frames = np.array(raw_frames, dtype=np.uint8)

data_name = args.source + '.csv'
data = np.loadtxt(data_name, delimiter=",", ndmin=2)
num_cells = data.shape[0]
# Status code: (-1, -1): unverified,
#              (0, 0): verified - bad,
#              (1, 1): verified - good,
#              (x, y): verified - good with lower threshold x and upper threshold
    y
centers, status, trace = np.hsplit(data, np.array([2, 4]))
# for backward compatibility when status did not include threshold
```

```
if status.shape[1] == 1:
    status = np.hstack((status, status))

# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# *^*^*^*^*^*^*^*^*^*^*^*^ Helper Functions  *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^


def moving_average(values, window=8):
    weights = np.repeat(1.0, window) / window
    sma = np.convolve(values, weights, 'valid')
    return sma


def hysteresis_threshold(tr, thresh_high, thresh_low):
    direction = np.zeros(len(tr))
    prev_direction = 1

    for k in range(0, len(tr)):
        if tr[k] < thresh_low:
            direction[k] = -1
            prev_direction = -1
        elif tr[k] > thresh_high:
            direction[k] = 1
            prev_direction = 1
        else:
            direction[k] = prev_direction

    return direction


def mad(arr):
    """Computes the median absolute deviation (MAD) of an input array."""
    med = np.median(arr)
    return np.median(np.abs(arr - med))


def threshold(y, lag, thresh_high, thresh_low, influence):
    """Dynamically thresholds input data array.
    Uses median, median absolute deviation, and asymmetric treatment of up and
        down signals to threshold input.
    Essentially a parametrized peak-trough detection algorithm.
    Args:
        y: input data
```

```
    lag: lag of moving window used to smooth data
    thresh_high: number of median absolute deviations data point differs
        above median to threshold up
    thresh_low: number of median absolute deviations data point differs below
        median to threshold down
    influence: influence (between 0 and 1) of new signals on median and
        median absolute deviation
Returns:
    signals, median filter, median absolute deviation filter
adapted from https://stackoverflow.com/q/22583391
"""
signals = np.zeros(len(y))
filtered_y = np.array(y)
med_filter = [0] * len(y)
mad_filter = [0] * len(y)
med_filter[lag - 1] = np.median(y[0:lag])
mad_filter[lag - 1] = mad(y[0:lag])
# for first lag signals, do not have prior data, so evaluate based on sign
for i in range(0, lag):
    signals[i] = 1 if y[i] > 0 else -1
for i in range(lag, len(y)):
    # UP: (y[i] - med_filter[i - 1] > threshold AND difference > 0.1) OR y[i]
        > 0.5 + min of 4 previous y[i]
    if (y[i] - med_filter[i - 1] > thresh_high * mad_filter[i - 1] and y[i] -
        med_filter[i - 1] > 0.3) or y[i] - min(y[i - 4:i]) > 0.4:
        signals[i] = 1

        filtered_y[i] = influence * y[i] + (1 - influence) * filtered_y[i - 1]
        med_filter[i] = np.median(filtered_y[(i-lag):i])
        mad_filter[i] = mad(filtered_y[(i-lag):i])
    # DOWN: (y[i] - med_filter[i - 1] < threshold AND difference < -0.1) OR
        y[i] < -0.5 + max of 4 previous y[i]
    elif (y[i] - med_filter[i - 1] < -thresh_low * mad_filter[i - 1] and y[i]
        - med_filter[i - 1] < -0.3) or y[i] - max(y[i - 4:i]) < -0.4:
        signals[i] = -1

        filtered_y[i] = influence * y[i] + (1 - influence) * filtered_y[i - 1]
        med_filter[i] = np.median(filtered_y[(i - lag):i])
        mad_filter[i] = mad(filtered_y[(i - lag):i])
    # NO CHANGE: signals[i] = signals[i - 1]
    else:
        signals[i] = signals[i - 1]
        filtered_y[i] = y[i]
        med_filter[i] = np.median(filtered_y[(i - lag):i])
```

```
            mad_filter[i] = mad(filtered_y[(i - lag):i])

    return np.asarray(signals), np.asarray(med_filter), np.asarray(mad_filter)

# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# *^*^*^*^*^*^*^*^*^*^*^*^*^*^ QC Type 1 *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^


def animate_frames_overlay(counter):
    num_subplots = 9
    num_frames = data.shape[1]
    radius = 8

    fig, ax = plt.subplots(3, 3)
    animations = []
    cells = []
    time_text = fig.text(0.147, 0.92, '', horizontalalignment='left',
        verticalalignment='top')

    def init():
        for i in range(num_subplots):
            center_x, center_y = centers[counter].astype(np.int)
            ax[i % 3, i // 3].set_xlim(center_x - 8, center_x + 8)
            ax[i % 3, i // 3].set_ylim(center_y - 8, center_y + 8)
            animations.append(ax[i % 3, i // 3].imshow(frames[num_frames /
                num_subplots * i, center_y - 8:center_y + 8, center_x - 8:center_x
                + 8], aspect='equal', extent=[center_x - 8, center_x + 8, center_y
                - 8, center_y + 8]))
            x = [center_x, center_x + radius * cos(trace[counter, num_frames /
                num_subplots * i])]
            y = [center_y, center_y + radius * sin(trace[counter, num_frames /
                num_subplots * i])]
            cells.append(ax[i % 3, i // 3].plot(x, y)[0])
        time_text.set_text('Frame 0 of %d' % (num_frames / num_subplots))

    def animate(frame):
        for i in range(num_subplots):
            center_x, center_y = centers[counter].astype(np.int)
            animations[i] = ax[i % 3, i // 3].imshow(frames[(num_frames /
                num_subplots * i) + frame % (num_frames / num_subplots), center_y
                - 8:center_y + 8, center_x - 8:center_x + 8], aspect='equal',
                extent=[center_x - 8, center_x + 8, center_y - 8, center_y + 8])
```

```
            # angle is calculated with respect to numpy array, i.e. arctan(x/y),
                so we correct with
            # x = center_x + sin(theta) and y = center_y + cos(theta)
            x = [center_x, center_x + radius * cos(trace[counter, (num_frames /
                num_subplots * i) + frame % (num_frames / num_subplots)])]
            y = [center_y, center_y + radius * sin(trace[counter, (num_frames /
                num_subplots * i) + frame % (num_frames / num_subplots)])]
            cells[i].set_data(x, y)
        time_text.set_text('Frame %d of %d' % (frame % (num_frames /
            num_subplots), num_frames / num_subplots - 1))

    anim = animation.FuncAnimation(fig, animate, init_func=init, interval=80)
    plt.show()

# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# *^*^*^*^*^*^*^*^*^*^*^*^*^ QC Types 2 and 3  *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^
# *^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^*^


def show_trace(counter):
    fig, ax = plt.subplots()

    def record_yes(event):
        status[counter] = thresh
        plt.close()

    def record_no(event):
        status[counter] = [0, 0]
        plt.close()

    unwrapped = np.unwrap(np.asarray(trace[counter]))
    ma_trace = moving_average(unwrapped, 8) # 8*1/32 fps ~ 250 ms moving average
        filter window
    velocity = np.convolve([-0.5, 0.0, 0.5], ma_trace, mode='valid')

    plt.xlabel('Frame', fontsize=20)
    plt.ylabel('Angle', fontsize=20)
    plt.title('Trace ({0}, {1}): {2} of {3}'.format(centers[counter][0],
        centers[counter][1], counter + 1, num_cells), fontsize=20)

    if args.type == 0:
        plt.plot(unwrapped, 'r-', lw=1)
    elif args.type == 2:
        thresh = [-1, -1]
```

```
        vel_range = np.abs(np.nanmax(velocity) - np.nanmin(velocity))
        # if thresh has previously been set
        if not np.array_equal(status[counter], [0, 0]) or not
            np.array_equal(status[counter], [-1, -1]):
            thresh_high, thresh_low = status[counter]
        else:
            thresh_high = np.nanmax(velocity) - vel_range * 0.50
            thresh_low = np.nanmin(velocity) + vel_range * 0.25
        thresh = [thresh_high, thresh_low]
        d = hysteresis_threshold(velocity, *thresh)

        def update_sensitivity(val):
            thresh_high = s_high_thresh.val
            thresh_low = s_low_thresh.val
            thresh[0], thresh[1] = thresh_high, thresh_low
            dd = hysteresis_threshold(velocity, *thresh)
            f1.set_ydata(dd)
            f2.set_ydata((thresh_high, thresh_high))
            f3.set_ydata((thresh_low, thresh_low))
            fig.canvas.draw_idle()

        s_high_thresh = Slider(fig.add_axes([0.20, 0.15, 0.65, 0.03]), 'Upper
            Threshold', -2.0, 2.0, valinit=thresh_high)
        s_low_thresh = Slider(fig.add_axes([0.20, 0.1, 0.65, 0.03]), 'Lower
            Threshold', -2.0, 2.0, valinit=thresh_low)
        s_high_thresh.on_changed(update_sensitivity)
        s_low_thresh.on_changed(update_sensitivity)
        f1, = plt.plot(range(0, len(velocity)), d, 'b-')
        f2, = plt.plot((0, len(velocity)), (thresh_high, thresh_high), 'g', lw=3)
        f3, = plt.plot((0, len(velocity)), (thresh_low, thresh_low), 'k', lw=3)
    elif args.type == 3:
        thresh = [1, 1] # no threshold in type 3, so set default to [1, 1]

        signals, med_filter, mad_filter = threshold(velocity, 4, 10, 3.5, 0.2)
        plt.plot(range(0, len(velocity)), velocity, 'r-')
        # FOR DEBUGGING USE, DISPLAY TRIGGERS FOR SWITCH
        # plt.plot(range(0, len(velocity)), med_filter, 'k', lw=0.5)
        # plt.plot(range(0, len(velocity)), med_filter + 10 * mad_filter, 'g',
            lw=0.5)
        # plt.plot(range(0, len(velocity)), med_filter - 3.5 * mad_filter, 'g',
            lw=0.5)
        plt.plot(range(0, len(velocity)), signals, 'b-')
        plt.ylim((-2, 2))
```

```
        plt.xlim((0, 1875))

    plt.grid(True, which='both')

    b_yes = Button(fig.add_axes([0.65, 0.9, 0.1, 0.03]), 'Yes')
    b_no = Button(fig.add_axes([0.80, 0.9, 0.1, 0.03]), 'No')
    b_yes.on_clicked(record_yes)
    b_no.on_clicked(record_no)

    figManager = plt.get_current_fig_manager()
    figManager.window.showMaximized()

    plt.show()


for i in range(num_cells):
    if args.type == 1:
        animate_frames_overlay(i)
    else:
        show_trace(i)

# output: center_x, center_y, status/upper threshold, status/lower threshold,
    trace
np.savetxt(args.source + "_checked.csv", np.hstack((centers, status, trace)),
    fmt=','.join(["%.4f"] * 2 + ["%.4f"] * 2 + ["%.4f"] * trace.shape[1]))
```