PyCelegans PostProcess Documentation

Release 0.5.2

Charlie Wright

April 26, 2013

CHAPTER

ONE

OVERVIEW AND USAGE

The purpose of the functions in this module are to check the output of PyCelegans and to format it for downstream data analysis. The post-processing code is run in two independent modes: **check** and **spline**.

```
postprocess.py check ...
```

The new integrity checks introduced are:

- Find frames where the midline length changes drastically
- Find frames where the head and tail have been misidentified

In addition, integrity checks from PyCelegans are included:

- Frames of looped worms

To identify frames that do not fit the length requirements, the `checklength` module calculates the average length of the worm midline over a sliding window, and locates frames whose length deviates significantly from this value. This comparison can be performed in two modes: *relative* (thresholded by number of standard deviations away from the mean) or *absolute* (thresholded by actual deviation in length away from the mean).

To locate frames where the head/tail orientation is flipped, the `flipheadtail` module compares the current frame either (A) to the surrounding frames in a sliding window or (B) to just the previous frame. If (A), it fits each midline to a polynomial, resulting in m x 2 (where m is the degree of the polynomial) coefficients for each frame. These coefficients are averaged over a sliding window, and the values for each individual frame compared to this mean to find frames where the posture deviates significantly from the local value. This process is performed iteratively to account for outliers, but still relies upon the assumption that the majority of the frames within the window are correct. If the score calculate by comparing the coefficients for one frame to the local average is greater than a maximum threshold, the midline is flipped. If it is less than a minimum threshold, nothing happens. If the score falls between the minimum and maximum thresholds, it is (optionally) marked as "indeterminate" and rejected from further analysis. If (B), it compares the total distance between each point along the current midline and the midline of the previous frame, oriented either head-to-tail or tail-to-head. If the latter number is smaller, a flip is identified.

After running the post-process module in `check` mode, the number of frames that passed each check will be printed. The outputs of each integrity check are saved as a .txt file in the original `properties` directory.

```
postprocess.py spline ...
```

The midline is fit to a smoothing B-spline, with the option to include the results of the integrity checks. The integrity checks are included in two steps: bad frames may be rejected, and flipped worms may optionally be reoriented. If any integrity checks are missing, they are simply ignored; i.e., if `postprocess.py spline` if run before `postprocess.py check`, only the `is_loop` criterion will be used. The length of the midline is also calculated by evaluating the integral of $(dx^2 + dy^2)^{1/2}$ along the midline. The sidepaths may be optionally splined, resulting in a closed B-spline that defines the perimeter of the worm. In this case the worm area is also calculated, by evaluating the integral of $(xdy - ydx) / 2$ along the worm perimeter. These values, along with their corresponding frame number, are finally exported to a .mat file for analysis in Matlab. *Note:* the format of the Matlab ouput will depend upon how the splines are generated; if a constant number of points are used then the midline and sidepaths will be

matrices of size m x n x 2 where m is the number of frames (that passed all integrity checks, if used) and n is the number of points in the spline. If constant spacing is used, n may vary so the results will be cells.

After running the post-process module in `spline` mode, the number of frames that threw no error will be printed. If analyzing data from the `./properties` directory, the splined .txt files will be saved to a separate directory as well as to a .mat file (by default, `./processed` and `./processed.mat`, respectively).

CHAPTER

TWO

RUN INTEGRITY CHECKS

To view the help section for the integrity checks, type

`postprocess.py check --help`

which will display:

```
usage: postprocess.py check [-h] [-p PROPERTIES] [-f FRAME_RANGE FRAME_RANGE]
                            [-v {0,1}] [-ls LEN_SPAN]
                            [-lv LEN_MAX_VAL | -ld LEN_MAX_DEV]
                            [-fd FLIP_DEGREE] [-fi FLIP_ITERS] [-fs FLIP_SPAN]
                            [-fl FLIP_MIN_DEV] [-fu FLIP_MAX_DEV]

optional arguments:
  -h, --help            show this help message and exit
  -p PROPERTIES, --properties PROPERTIES
                        properties directory; default = ./properties
  -f FRAME_RANGE FRAME_RANGE, --frame_range FRAME_RANGE FRAME_RANGE
                        frame range, given as [first frame, last frame);
                        default = '0 inf'
  -v {0,1}, --version {0,1}
                        version of process code used to find midline (0 =
                        Nick's, 1 = Marc's); default = 0
  -ls LEN_SPAN, --len_span LEN_SPAN
                        size of window used to check lengths, over which the
                        average worm length should be constant; default = 6000
  -lv LEN_MAX_VAL, --len_max_val LEN_MAX_VAL
                        maximum allowed value of length outside of average
                        value in the moving window set by the span, set as a
                        percentage; set this value but not len_max_dev to use
                        an absolute value of the threshold instead of
                        comparing to a deviation about the average; default =
                        0.05
  -ld LEN_MAX_DEV, --len_max_dev LEN_MAX_DEV
                        maximum allowed deviation of length from average value
                        of the standard score in the moving window set by the
                        span; default = 0.00
  -fd FLIP_DEGREE, --flip_degree FLIP_DEGREE
                        degree used in polynomial fit to determine correct
                        orientation; default = 10
  -fi FLIP_ITERS, --flip_iters FLIP_ITERS
                        number of times to run flip-check procedure; default =
                        3
  -fs FLIP_SPAN, --flip_span FLIP_SPAN
                        size of window used to check flips, over which the
                        posture of the worm should be approximately constant;
```

```
                        default = 20
  -fl FLIP_MIN_DEV, --flip_min_dev FLIP_MIN_DEV
                        minimum deviation of coefficients used to fit worm
                        midline in moving window, below which the orientation
                        is considered correct; default = 2.00
  -fu FLIP_MAX_DEV, --flip_max_dev FLIP_MAX_DEV
                        maximum deviation of coefficients used to fit worm
                        midline in moving window, above which the orientation
                        is considered flipped; default = 3.00
```

To check the first 100 frames of an experiment in `./properties` using Nick's midline and default parameters:

```
postprocess.py check -p ./properties -f 0 100
```

To check the frames 3000 through 4000 (inclusive), set the `frame_range`:

```
postprocess.py check -f 3000 4001
```

To check every frame of an experiment using Marc's midline, set the `version` flag = 1:

```
postprocess.py check -v 1
```

To use a more restrictive threshold for the length-check, decrease the maximum allowed deviations by setting `len_max_val`:

```
postprocess.py check -lv 0.02
```

To use a less restrictive threshold for the length-check, increase the maximum allowed deviation by setting `len_max_val`:

```
postprocess.py check -lv 0.1
```

To use the standard score instead of the fractional change in length, set `len_max_dev`:

```
postprocess.py check -ld 3.0
```

To change the window size used to check for bad midline lengths (e.g., to 10 min at a frame rate of 10 fps), set `length_span`:

```
postprocess.py check -ls 6000
```

To correct for head/tail flips without rejecting "indeterminate" frames, set `flip_min_dev` and `flip_max_dev` (the lower and upper thresholds, respectively) equal:

```
postprocess.py check -fl 3.0 -fu 3.0
```

To change the window size used to check for flips (e.g., to 3 sec at a frame rate of 10 fps), set `flip_span` to any integer > 1:

```
postprocess.py check -fs 30
```

To check for flips by comparing just to the previous frame (calculating distances between points along a splined midline, instead of comparing the coefficients of the polynomial fit), set `flip_span` = 1:

```
postprocess.py check -fs 1
```

CHAPTER

THREE

FIT TO SMOOTHING SPLINE

To view the help section for splining the data, type

`postprocess.py spline --help`

which will display:

```
usage: postprocess.py spline [-h] [-p PROPERTIES] [-f FRAME_RANGE FRAME_RANGE]
                             [-v {0,1}] [-c] [-r] [-k] [-o OUTPUT]
                             [-s SMOOTHING] [-n NUM_POINTS | -sp SPACING]
                             [-l LENGTH]

optional arguments:
  -h, --help            show this help message and exit
  -p PROPERTIES, --properties PROPERTIES
                        properties directory; default = ./properties
  -f FRAME_RANGE FRAME_RANGE, --frame_range FRAME_RANGE FRAME_RANGE
                        frame range, given as [first frame, last frame);
                        default = '0 inf'
  -v {0,1}, --version {0,1}
                        version of process code used to find midline (0 =
                        Nick's, 1 = Marc's); default = 0
  -c, --use_checks      include this flag to use info from various post-
                        processing checks when calculating splines
  -r, --reorient        include this flag to reorient tail-to-head any worms
                        marked as incorrectly flipped; this flag will be
                        ignored if the '--use_checks' flag is not used
  -k, --keep_sides      include this flag to read the sidepaths and save a
                        single splined worm outline to file
  -o OUTPUT, --output OUTPUT
                        name of directory and .mat file to save splined data;
                        default = ./processed
  -s SMOOTHING, --smoothing SMOOTHING
                        level of smoothing in creating splined version of worm
                        midline; should be of the order of the number of
                        points in the midline; default = 25
  -n NUM_POINTS, --num_points NUM_POINTS
                        number of points in the each splined worm; default =
                        100
  -sp SPACING, --spacing SPACING
                        distance between consecutive (x, y) points in each
                        splined worm; this value is exclusive of the length
                        and specifying it may result in splines with unequal
                        numbers of points; default = 0.00
  -l LENGTH, --length LENGTH
                        desired length of each splined midline; if specified
```

```
all splined worms will be approximately the same
length (depending on whether a uniform number of
points or spacing is given); default = 0.00
```

To spline the first 100 frames of an experiment in `./properties` using Nick's midline and default parameters, and without using any automatic corrections from the integrity checks:

```
postprocess.py spline -f 0 100
```

To spline every frame using Marc's midline without input from integrity checks:

```
postprocess.py spline -v 1
```

To include integrity checks (but ignore flips), just set the `use_checks` flag:

```
postprocess.py spline -c
```

To include integrity checks (and automatically reorient tail-to-head flips), set both the `use_checks` and `reorient` flags:

```
postprocess.py spline -cr
```

To also include sidepaths and worm areas, set the `keep_sides` flag:

```
postprocess.py spline -crk
```

To make the spline smoother, increase the value of `smoothing`:

```
postprocess.py spline -s 50
```

To use half as many points in the midline, set the value of `num_points`:

```
postprocess.py spline -n 50
```

To use a spline in which the points are spaced an equal number of pixels apart (this will produce results with a non-uniform number of points across splines), set the `spacing`:

```
postprocess.py spline -sp 5.0
```

Note: if the sidepaths are also splined, they are automatically generated with twice the number of points in the midline, but with twice the smoothing factor (which should result in approximately the same amount of local smoothing).

CHAPTER

FOUR

POSTPROCESS MODULE

This is the main script for post-processing. It can run in two modes:

1. check: run various checks on the output of PyCelegans
2. spline: spline the worm midline and export to Matlab

`postprocess.main(`*inputs*`)`

Main function for accessing post-process functions.

args: inputs: command line inputs

CHAPTER

FIVE

IMPORTDATA MODULE

Read output of PyCelegans used for further analysis: convert each midline to a numpy array and calculate the distance along the midline from nose to tail.

`importdata.getnosedist(`*nosetail*`)`
Calculate distance of each point along worm from nosetip.

args: nosetail (dict): dict of numpy arrays of coordinates along midline

returns: nosedist (dict): dict of numpy arrays of distance from nose to tail

nosetail (dict): same as input, but with same set of keys as nosedist

`importdata.main(`*path_name, frame_range=[0, inf], version=0*`)`
Import midline and calculate distance from nose to tail.

args: path_name (str): name of directory containing properties .txt files

kwargs: frame_range (tuple): range of frames to read in given as [first, last)

version (int): version of PyCelegans process.py code output to use (0: Nick Labello's midline, 1: Marc Goessling's midline)

returns: nosedist (dict): dict of numpy arrays of distance from nose to tail

nosetail (dict): same as input, but with same set of keys as nosedist

`importdata.readnosetail(`*path_name, frame_range=[0, inf], version=0*`)`
Read worm midline (assuming nose-to-tail), converting to numpy array.

args: path_name (str): name of directory containing properties .txt files

kwargs: frame_range (tuple): range of frames to read in given as [first, last)

version (int): version of PyCelegans process.py code output to use (0: Nick Labello's midline, 1: Marc Goessling's midline)

returns: nosetail (dict): dict of numpy arrays of coordinates along midline

CHAPTER

SIX

CHECKLENGTH MODULE

This module checks the output of PyCelegans by calculating the average length of the worm midline over a sliding window, and rejecting frames whose length deviates significantly from this value.

checklength.checklength(*nosedist*, *span=6000*, *max_dev=0.0*, *max_val=0.05*)

Remove frames whose lengths do not fit the specified criteria: 1. Must fall within the specified number of standard deviations from the mean, calculated over a moving window => set max_dev but not max_val 2. Must fall within a constant value of the mean, calculated over a moving window => set max_val but not max_dev

args: nosedist (dict): dict of numpy arrays of distance from nose to tail

kwargs: span (int): width of window

max_dev (float): maximum allowed difference between Z-score of length

max_val (float): maximum allowed difference between length and average

returns: dict: dict of booleans specifying frames that pass or fail the test

checklength.main(*nosedist*, *span=6000*, *max_dev=0.0*, *max_val=0.05*)

Find frames where the length fails to meet specified criteria

args: nosedist (dict): dict of numpy arrays of distance from nose to tail

kwargs: span (int): width of window

max_dev (float): maximum allowed difference between Z-score of length

max_val (float): maximum allowed difference between length and average

returns: dict: dict of booleans specifying frames that pass or fail the test

CHAPTER

SEVEN

FLIPHEADTAIL MODULE

This module checks the output of PyCelegans by determining the frames where the nose has been incorrectly identified. Each worm midline is fit to a polynomial, and the coefficients from each frame are compared to the average coefficients (calculated over a sliding window) to find frames where the posture deviates significantly from the local value. This process is performed iteratively to account for outliers, but still relies upon the assumption that the majority of the frames within the window are correct. However, if the span is just one frame, only calculate the distance between points along the worm between consecutive frames, identifying flips where this value is smaller where one of the frames has been flipped.

`flipheadtail.checkflips(`*coefs_for*, *coefs_rev*, *span=20*, *min_dev=2.0*, *max_dev=3.0*`)`

Check for nose-tail flips by comparing coefficients corresponding to worm oriented in forward and reverse direction. When a value exceeeds the maximum specified deviation from the local window, flip the coefficients.

args: coefs_for (dict): dict of coefficients corresponding to nose-to-tail

coefs_rev (dict): dict of coefficients corresponding to tail-to-nose

kwargs: span (int): number of points in final midline

min_dev (float): spacing between points

max_dev (float): dict of desired lengths of each midline

returns: coefs_for (dict): adjusted values of coefs_for

coefs_rev (dict): adjusted values of coefs_rev

is_flipped (dict): dict of strings with values of 'False': no flip (value < min_dev) 'True': yes flip (value > max_dev) 'Indeterminate': could not determine (min_dev < value < max_dev)

`flipheadtail.evalpolyfit(`*coefs*, *num_points=100*, *spacing=None*, *lengths=None*`)`

Evaluate the polynomial fit at discrete points. There are four options for constructing the spline: 1) Same number of points, variable lengths, variable spacing => must specify only num_points (default) 2) Variable number of points, variable lengths, same spacing => must specify only spacing 3) Same number of points, same lengths, same spacing => must specify length and either spacing or num_points

args: coefs (dict): dict of coefficients (output of makepolyfit)

kwargs: num_points (int): number of points in final midline

spacing (int): spacing between points

lengths (dict): dict of desired lengths of each midline

returns: nosedist (dict): dict of uniformly-sampled distances from nose to tail

nosetail (dict): dict of midlines evaluated using polynomial fit at each point in the output nosedist

`flipheadtail.main(`*nosedist*, *nosetail*, *degree=10*, *num_iters=3*, *span=20*, *min_dev=2.0*, *max_dev=3.0*, *smoothing=25*, *num_points=100*`)`

Find frames where head/tail identification has failed by making a polynomial representation of the worm and

comparing the value of the coefficients frame-by-frame to the average in a sliding window. If the size of the sliding window is set to just one frame however, do not fit to a polynomial but instead calculate the Euclidean distance between points along splined worms in consecutive frames, counting a flip if the norm between 'forward' and 'reverse' is smaller than between both 'forward'.

args: nosedist (dict): dict of numpy arrays of distance from nose to tail

nosetail (dict): dict of numpy arrays of coordinates along midline

kwargs: degree (int): degree of polynomial fit

num_iters (int): number of times to repeat the head/tail flip check

span (int): number of points in final midline

min_dev (float): spacing between points

max_dev (float): dict of desired lengths of each midline

returns: is_flipped (dict): dict of strings with values of 'False': no flip (value < min_dev) 'True': yes flip (value > max_dev) 'Indeterminate': could not determine (min_dev < value < max_dev)

`flipheadtail.makepolyfit(`*nosedist*, *nosetail*, *degree=10*)

Fit the midline to a polynomial.

args: nosedist (dict): dict of numpy arrays of distance from nose to tail

nosetail (dict): dict of numpy arrays of coordinates along midline

kwargs: degree (int): degree of polynomial fit

returns: dict: dict of coefficients, each of which is m X 2 array for m degrees

CHAPTER

EIGHT

SPLINEFIT MODULE

Construct a uniformly-sampled B-spline of the worm, given the set of points running along the worm midline from nosetip to tailtip and the distance between each of those points.

`splinefit.evalbspline(`*tcks*, *num_points=100*, *spacing=0.0*, *lengths=None*`)`

Evaluate the spline at discrete points. There are three options for constructing the spline: 1) Same number of points, variable lengths, variable spacing => must specify num_points (default) and each length 2) Variable number of points, variable lengths, same spacing => must specify spacing and each length 3) Same number of points, same lengths, same spacing => must specify all same length and either spacing or num_points

args: tcks (dict): dict of spline fit (knots, coefs, degree)

kwargs: num_points (int): number of points in final midline

spacing (int): spacing between points

lengths (dict): dict of desired lengths of each midline

returns: nosedist (dict): dict of uniformly-sampled distances from nose to tail

nosetail (dict): dict of midlines evaluated using polynomial fit at each point in the output nosedist

`splinefit.main(`*us*, *vs*, *smoothing=25*, *num_points=100*, *spacing=0.0*, *length=0.0*, *per=0*`)`

Fit the worm to a smoothing B-spline.

args: us (array): euclidean distance between points

vs (array): array of (x, y) points

kwargs: smoothing (int): smoothing

num_points (int): number of points in final midline

spacing (int): spacing between points

lengths (dict): dict of desired lengths of each midline

per (int): set non-zero for a closed spline, i.e., periodic b.c.

returns: nosedist (dict): dict of uniformly-sampled distances from nose to tail

nosetail (dict): dict of midlines evaluated using polynomial fit at each point in the output nosedist

`splinefit.makebspline(`*us*, *vs*, *s=25*, *per=0*`)`

Fit the a series of (x, y) coordinates to a smoothing B-spline.

args: us (array): euclidean distance between points

vs (array): array of (x, y) points

kwargs: s (int): smoothing

per (int): set non-zero for a closed spline, i.e., periodic boundary conditions (default = 0)

returns: dict: dict of spline fit (knots, coefs, degree)

CHAPTER

NINE

CALC MODULE

This module contains miscellaneous functions for time series data analysis.

`calc.fullmovingavg(`*data*, *span=5*`)`
Moving average, padded on ends with original data.

args: data (iterable): data sequence (1D)

kwargs: span (int): width of window

returns: list: output of func applied to data over the sliding window is a list of the same size as the input data, where instead of padding the ends of the output, the average is calculated but over a smaller window until reaching the ends of the sequence

`calc.moving(`*data*, *func=<function mean at 0x10de0bd70>*, *span=5*`)`
Calculate the value of the function in a moving (sliding) window.

args: data (iterable): data sequence (1D)

kwargs: func: name of function (default = np.mean)

span (int): width of window

returns: list: output of func applied to data over the sliding window is a list of the same size as the input data, where the ends are padded with the first and last values of the input data

`calc.window(`*seq*, *n=2*`)`
Returns a sliding window (of width n) over data from the iterable s -> (s0,s1,...s[n-1]), (s1,s2,...,sn), ... From itertools examples.

args: seq (iterable): data sequence

kwargs: n (int): width of window (default = 2)

returns: iter: sliding window

CHAPTER

TEN

PARSEARGS MODULE

This module creates the argument parser for accepting command line inputs.

parseargs.addchecklength(*parser*)
: Add arguments to parser for checklength.py

parseargs.addflipheadtail(*parser*)
: Add arguments to parser for flipheadtail.py

parseargs.addimportdata(*parser*)
: Add arguments to parser for importdata.py

parseargs.addsplinefit(*parser*)
: Add arguments to parser for splinefit.py

parseargs.main(*inputs*)

CHAPTER

ELEVEN

READWRITE MODULE

This module contains functions for reading and writing PyCelegans output data.

`readwrite.readtxt(`*path_name*, *parameter*, *frame_range=(0, inf)*, *data_type=<type 'float'>*`)`

Read data from text file, e.g., .txt output of collectoutput.py. The expected input format is that each line contains data separated by commas, with the first value corresponding to the frame number. The output is a dictionary with keys corresponding to frame numbers and values corresponding to the data contained in each line.

args: path_name (str): name of directory containing properties .txt files

parameter (str): name of parameter (file name minus extension)

kwargs: frame_range (tuple): range of frames to read in given as [first, last) will read in all data from file)

data_type: expected data type

returns: dict: dict of values; keys are frame numbers

`readwrite.reformat(`*data_dict*`)`

Reformat data dictionary for saving to Matlab. Convert all dictionaries to lists, and construct a new value called frames that corresponds to the full set of keys for each variable.

args: data_dict: dict of dicts

returns: dict: dict of lists

`readwrite.savemat(`*file_name*, *data_dict*`)`

Reformat so that dictionaries become lists and save to .mat file. Input is dictionary of values, i.e., {'midline':..., 'nosetip':..., }

args: file_name (str): name of .mat file to save data

data_dict (dict): dict of dicts

`readwrite.writetxt(`*path_name*, *parameter*, *data_dict*, *frames=[ ]*, *data_format='%0.1f'*`)`

Write data to text file. The exact inverse of the read function. The inputs are the parameter name (while determines the file name) and the data dictionary, and a string describing how to convert the data values to text (e.g., %0.4f for float, %r for boolean).

args: path_name (str): name of directory containing properties .txt files

parameter (str): name of parameter (file name minus extension)

data_dict (dict): dict of values; keys are frame numbers

kwargs: frames (list): list of frames to write to file. If the data_dict does not have a key corresponding to each frame in this list then an error (-1) will be written. If this input is empty, all values from data_dict will be written

data_format: how to format each value to string

CHAPTER

TWELVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*