

Behavpy

- [About](#)
- [Basic methods](#)
- [Visualising your data](#)
- [Visualising mAGO data](#)

About

Behavpy is the default object in ethoscropy, a way of storing your metadata and data in a single structure, whilst adding methods to help you manipulate and analyse your data.

Metadata is crucial for proper statistical analysis of the experimental data. In the context of the Ethoscope, the data is a long time series of recorded variables, such as position and velocity for each individual. It is easier and neater to always keep the data and metadata together. As such, behavpy class is a child class of Pandas dataframe, the widely used data table package in python, but enhanced by the addition of a metadata as a class variable. Both the metadata and data must be linked by a common unique identifier for each individual (the 'id' column), which is automatically done if you've loaded with Ethoscropy (If not see the bottom of the page for the requirements for creating a behavpy object from alternative data sources).

The behavpy class has variety of methods designed to augment, filter, and inspect your data which we will go into later. However, if you're not already familiar with [Pandas](#), take some time to look through their guide to get an understanding of its many uses.

Initialising behavpy

To create a behavpy object you need matching metadata and data. To match they both need to have an id column that has linked/same id's in each. Don't worry, if you downloaded/formatted the data using the built in ethoscropy functions shown previously they'll be in there already.

```
import ethoscropy as etho

# You can load you data using the previously mentioned functions
meta = etho.link_meta_index(meta_loc, remote, local)
data = etho.load_ethoscope(meta, reference_hour = 9.0, FUN = partial(etho.sleep_annotation,
time_window_length = 60, min_time_immobile = 300))

# Or you can load then from a saved object of your choice into a pandas df,
# mine is a pickle file
meta = pd.read_pickle('users\folder\experiment1_meta.pkl')
data = pd.read_pickle('users\folder\experiment1_data.pkl')

# to initialise a behavpy object, just call the class with the data and meta as arguments. Have check = True to
ensure the ids match between metadata and data.
# As of version 1.3.0 you can choose the colour palette for the plotly plots - see
https://plotly.com/python/discrete-color for the choices
# The default is 'Safe' (the one used before), but this example we'll use 'Vivid'
df = etho.behavpy(data = data, meta = meta, colour = 'Vivid', check = True)
```

Using behavpy with non-ethoscope data

The behavpy class is made to work with the ethoscope system, utilising the data structure it records to create the analytics pipeline you'll see next. However, you can still use it on non-ethoscope data if you follow the same structure.

Data sources:

You will need the metadata file as discussed prior, however you will need to manually create a column called `id` that contains a unique id per specimen in the experiment.

Additionally, you will need a data source where each row is a log of a time per specimen. Each row must have the following

- **id** column, that references back to the unique id in the metadata
- **t** column, the time (in seconds) the row is describing, i.e. 0, 60, 120, 180, 240
- **moving** column, a boolean column (true/false) of whether the specimen is moving

The above columns are necessary for all the methods to work, but feel free to add other columns with extra information per timestamp. Both these data sources must be converted to pandas DataFrames, which can then be used to create a behavpy class object as shown above.

Basic methods

Behavpy has lots of built in methods to manipulate your data. The next few sections will walk you through a basic methods to manipulate your data before analysis.

Filtering by the metadata

One of the core methods of behavpy. This method creates a new behavpy object that only contains specimens whose metadata matches your inputted list. Use this to separate out your data by experimental conditions for further analysis.

```
# filter your dataset by variables in the metadata with .xmv()
# the first argument is the column in the metadata
# the second can be the variables in a list or as subsequent arguments

df = df.xmv('species', ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak', 'D.sims'])
# or
df = df.xmv('species', 'D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak', 'D.sims')

# the new data frame will only contain data from specimens with the selected variables
```

Removing by the metadata

The inverse of `.xmv()`. Remove from both the data and metadata any experimental groups you don't want. This method can be called also on individual specimens by specifying their `id` and their unique identifier.

```
# remove specimens from your dataset by the metadata with .remove()
# remove acts like the opposite of .xmv()

df = df.remove('species', ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak', 'D.sims'])
# or
df = df.remove('species', 'D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak', 'D.sims')

# both .xmv() and .remove() can filter/remove by the unique id if the first argument = 'id'
df = df.remove('id', '2019-08-02_14-21-23_021d6b|01')
```

Filtering by time

Often you will want to remove from the analyses the very start of the experiments, when the data isn't as clean because animals are habituating to their new environment. Or perhaps you'll want to just look at the baseline data before something occurs. Use `.t_filter()` to filter the dataset between two time points.

```
# filter you dataset by time with .t_filter()
# the arguments take time in hours
# the data is assumed to be represented in seconds

df = df.t_filter(start_time = 24, end_time = 48)

# Note: the default column for time is 't', to change use the parameter t_column
```

Concatenate

Concatenate allows you to join two or more behavpy data tables together, joining both the data and metadata of each table. The two tables do not need to have identical columns: where there's a mismatch the column values will be replaced with `NaNs`.

```
# An example of concatenate using .xmv() to create separate data tables
df1 = df.xmv('species', 'D.vir')
df2 = df.xmv('species', 'D.sec')
df3 = df.xmv('species', 'D.ere')

# a behavpy wrapper to expand the pandas function to concat the metadata
new_df = df1.concat(df2)

# .concat() can process multiple data frames
new_df = df1.concat(df2, df3)
```

Analyse a single column

Sometimes you want to get summary statistics of a single column per specimen. This is where you can use `.analyse_column()`. The method will take all the values in your desired column per specimen and apply a summary statistic. You can choose from a basic selection, e.g. mean, median, sum. But you can also use your own function if you wish (the function must work on array data and return a single output).

```
# Pivot the data frame by 'id' to find summary statistics of a selected columns
# Example summary statistics: 'mean', 'max', 'sum', 'median'...
```

```
pivot_df = df.analyse_column('interactions', 'sum')
```

output:

	interactions_sum
id	
2019-08-02_14-21-23_021d6b 01	0
2019-08-02_14-21-23_021d6b 02	43
2019-08-02_14-21-23_021d6b 03	24
2019-08-02_14-21-23_021d6b 04	15

```
2020-08-07_12-23-10_172d50|18      45
2020-08-07_12-23-10_172d50|19      32
2020-08-07_12-23-10_172d50|20      43
```

```
# the output column will be a string combination of the column and summary statistic
# each row is a single specimen
```

Re-join

Sometimes you will create an output from the pivot table or just a have column you want to add to the metadata for use with other methods. The column to be added must be a pandas series of matching length to the metadata and with the same specimen IDs.

```
# you can add these pivoted data frames or any data frames with one row per specimen to the metadata with
.rejoin()

# the joining dataframe must have an index 'id' column that matches the metadata

df = df.rejoin(pivot_df)
```

Binning time

Sometimes you'll want to aggregate over a larger time to ensure you have consistent readings per time points. For example, the ethoscope can record several readings per second, however sometimes tracking of a fly will be lost for short time. Binning the time to 60 means you'll smooth over these gaps. However, this will just be done for 1 variable so will only be useful in specific analysis. If you want this applied across all variables remember to set it as your time window length in your loading functions.

```
# Sort the data into bins of time with a single column to summarise the bin

# bin time into groups of 60 seconds with 'moving' the aggregated column of choice
# default aggregating function is the mean
bin_df = df.bin_time('moving', 60)
```

output:

```
          t_bin  moving_mean
id
2019-08-02_14-21-23_021d6b|01  86400      0.75
2019-08-02_14-21-23_021d6b|01  86460      0.5
2019-08-02_14-21-23_021d6b|01  86520      0.0
2019-08-02_14-21-23_021d6b|01  86580      0.0
2019-08-02_14-21-23_021d6b|01  86640      0.0
...
2020-08-07_12-23-10_172d50|19  431760      1.0
2020-08-07_12-23-10_172d50|19  431820      0.75
```

```
2020-08-07_12-23-10_172d50|19 431880    0.5
2020-08-07_12-23-10_172d50|19 431940    0.25
2020-08-07_12-23-10_172d50|20 215760    1.0
```

the column containing the time and the aggregating function can be changed

```
bin_df = df.bin_time('moving', 60, t_column = 'time', function = 'max')
```

output:

```

            time_bin  moving_max
id
2019-08-02_14-21-23_021d6b|01  86400    1.0
2019-08-02_14-21-23_021d6b|01  86460    1.0
2019-08-02_14-21-23_021d6b|01  86520    0.0
2019-08-02_14-21-23_021d6b|01  86580    0.0
2019-08-02_14-21-23_021d6b|01  86640    0.0
...
2020-08-07_12-23-10_172d50|19 431760    1.0
2020-08-07_12-23-10_172d50|19 431820    1.0
2020-08-07_12-23-10_172d50|19 431880    1.0
2020-08-07_12-23-10_172d50|19 431940    1.0
2020-08-07_12-23-10_172d50|20 215760    1.0
```

Wrap time

The time in the ethoscope data is measured in seconds, however these numbers can get very large and don't look great when plotting data or showing others. Use this method to change the time column values to be a decimal of a given time period, the default is the normal day (24) and will change time to be in hours from reference hour or experiment start.

```
# Change the time column to be a decimal of a given time period, e.g. 24 hours
# wrap can be performed inplace and will not return a new behavpy
df.wrap_time(24, inplace = True)
# however if you want to create a new dataframe leave inplace False
new_df = df.wrap_time(24)
```

Remove specimens with low data points

Sometimes you'll run an experiment and have a few specimens that were tracked poorly or just have fewer data points than the rest. This can be really affect some analysis, so it's best to remove it.

Specify the minimum number of data points you want per specimen, any lower and they'll be removed from the metadata and data. Remember the minimum points per a single day will change with the frequency of your measurements.

```
# removes specimens from both the metadata and data when they have fewer data points than the user specified amount
```

```
# 1440 is 86400 / 60. So the amount of data points needed for 1 whole day if the data points are measured every minute
```

```
new_df = df.curate(points = 1440)
```

Remove specimens that aren't sleep deprived enough

In the Gilestro lab we'll sleep deprive flies to test their response. Sometimes the method won't work and you'll a few flies mixed in that have slept normally. Call this method to remove all flies that have been asleep for more than a certain percentage over a given time period. This method can return two difference outputs depending on the argument for the remove parameter. If it's a integer between 0 and 1 then any specimen with more than that fraction as asleep will be removed. If left as False then a pandas data frame is returned with the sleep fraction per specimen.

```
# Here we are removing specimens that have slept for more than 20% of the time between the period of 24 and 48 hours.
```

```
dfn = df.remove_sleep_deprived(start_time = 24, end_time = 48, remove = 0.2, sleep_column = 'asleep',  
t_column = 't')
```

```
# Here we will return the sleep fraction per specimen
```

```
df_sleep_fraction = df.remove_sleep_deprived(start_time = 24, end_time = 48, sleep_column = 'asleep',  
t_column = 't')
```

Interpolate missing results

Sometimes you'll have missing data points, which is not usually too big of a problem. However, sometimes you'll need to do some analysis that requires regularly measured data. Use the `.interpolate()` method to set a recording frequency and interpolate any missing points from the surrounding data. Interpolate is a wrapper for the scipy interpolate function.

```
# Set the variable you want to interpolate and sampling frequency you'd like (step_size)
```

```
# step size is given in seconds. Below would interpolate the data for every 5 mins from the min time to max time
```

```
new_df = df.interpolate(variable = 'x', step_size = 300)
```

Baseline

Not all experiments are run at the same time and you'll often have differing number of days before an interaction (such as sleep deprivation) occurs. To have all the data aligned so the interaction day is the same include in your metadata `.csv` file a column called `baseline`. Within this column, write the number of additional days that needs to be added to align to the longest set of baseline experiments.


```
# add additional time to specimens time column to make specific interaction times line up when the baseline
time is not consistent

# the metadata must contain a a baseline column with an integer from 0 - infinity

df = df.baseline(column = 'baseline')

# perform the operation inplace with the inplace parameter
```

Add day numbers and phase

Add new columns to the data, one called phase will state whether it's light or dark given your reference hour and a normal circadian rhythm (12:12). However, if you're working with different circadian hours you can specify the time it turns dark.

```
# Add a column with the a number which indicates which day of the experiment the row occurred on
# Also add a column with the phase of day (light, dark) to the data
# This method is performed in place and won't return anything.
# However you can make it return a new dataframe with the inplace = False
df.add_day_phase(t_column = 't') # default parameter for t_column is 't'

# if you're running circadian experiments you can change the length of the days the experiment is running
# as well as the time the lights turn off, see below.

# Here the experiments had days of 30 hours long, with the lights turning off at ZT 15 hours.
# Also we changed inplace to False to return a modified behavpy, rather than modify it in place.
df = df.add_day_phase(day_length = 30, lights_off = 15, inplace = False)
```

Estimate Feeding

If you're using the ethoscope we can approximate the amount of time feeding by labelling micro-movements near the end of the tube with food in it as feeding times. This method relies upon your data having a micro column which should be generated if you load the data with the motion_detector or sleep_annotation loading function.

This method will return a new behavpy object that has an additional column called 'feeding' with a boolean label (True/False). The subsequent new column can then be plotted as is shown on the next page.

```
# You need to declare if the food is positioned on the outside or inside so the method knows which end to look
at
new_df = df.feeding(food_position = 'outside', dist_from_food = 0.05, micro_mov = 'micro', x_position = 'x') #
micro_mov and x_position are the column names and defaults
# The default for distance from food is 0.05, which is a hedged estimate. Try looking at the spread of the x
position to get a better idea what the number should be for your data
```

Automatically remove dead animals

Sometimes the specimen dies or the tracking is lost. This method will remove all data of the specimen after they've stopped moving for a considerable length of time.

```
# a method to remove specimens that havent moved for certain amount of time
# only data past the point deemed dead will be removed per specimen
new_df = df.curate_dead_animals()

# Below are the standard numbers and their variable names the function uses to remove dead animals:
# time_window = 24: The window in which death is defined, set to 24 houurs or 1 day
# prop_immobile = 0.01: The proportion of immobility that counts as "dead" during the time window
# resoltion = 24: How much the scanning window overlap, expressed as a factor
```

Find lengths of bouts of sleep

```
# break down a specimens sleep into bout duration and type

bout_df = df.sleep_bout_analysis()

output:
      duration asleep      t
id
2019-08-02_14-21-23_021d6b|01    60.0  True  86400.0
2019-08-02_14-21-23_021d6b|01   900.0 False  86460.0
...
2020-08-07_12-23-10_172d50|05   240.0  True  430980.0
2020-08-07_12-23-10_172d50|05   120.0 False  431760.0
2020-08-07_12-23-10_172d50|05    60.0  True  431880.0

# have the data returned in a format ready to be made into a histogram
hist_df = df.sleep_bout_analysis(as_hist = True, max_bins = 30, bin_size = 1, time_immobile = 5, asleep = True)

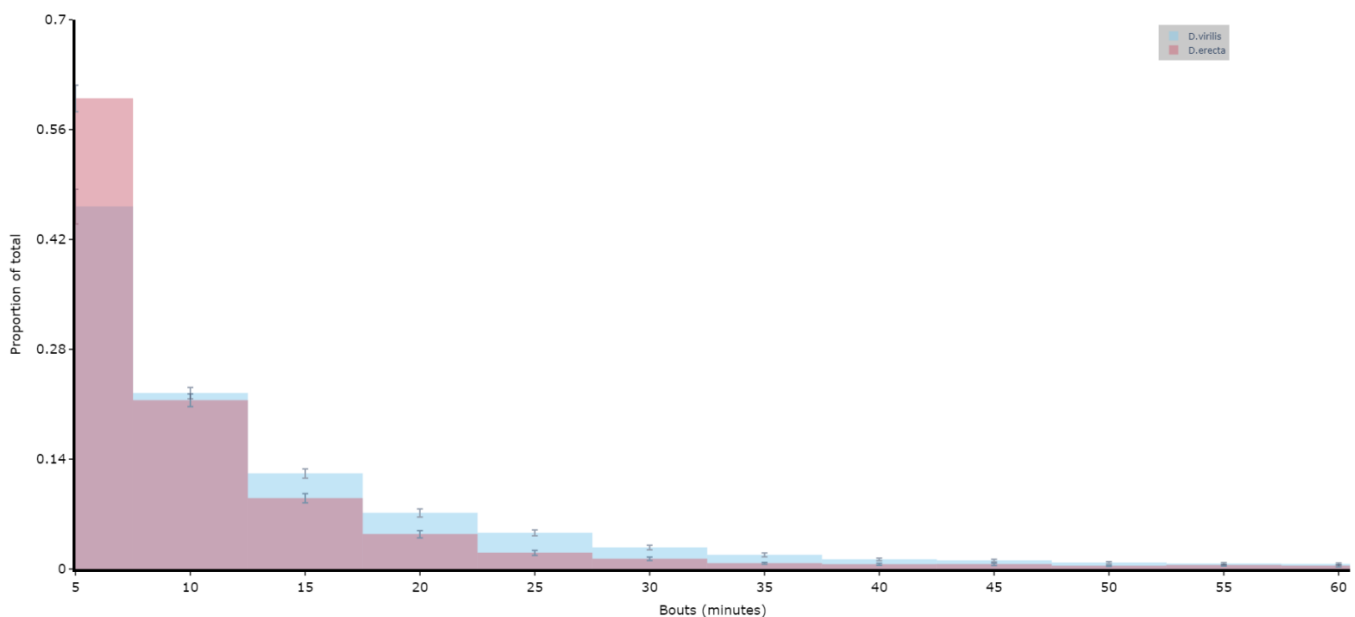
output:
      bins count  prob
id
2019-08-02_14-21-23_021d6b|01    60    0 0.000000
2019-08-02_14-21-23_021d6b|01   120   179 0.400447
2019-08-02_14-21-23_021d6b|01   180    92 0.205817
...
2020-08-07_12-23-10_172d50|05  1620    1 0.002427
2020-08-07_12-23-10_172d50|05  1680    0 0.000000
2020-08-07_12-23-10_172d50|05  1740    0 0.000000
```

```
# max_bins is the largest bout you want to include
# bin_size is the what length runs together, i.e. 5 would find all bouts between factors of 5 minutes
# time_immobile is the time in minutes sleep was defined as prior. This removes anything that is small than this
as produced by error previously.
# if asleep is True (the default) the return data frame will be for asleep bouts, change to False for one for awake
bouts
```

Plotting a histogram of sleep_bout_analysis

```
# You can take the output from above and create your own histograms, or you can use this handy method to
plot a histogram with error bars from across your specimens
# Like all functions you can facet by your metadata
# Here we'll compare two of the species and group the bouts into periods of 5 minutes, with up to 12 of them (1
hour)
# See the next page for more information about plots
```

```
fig = df.plot_sleep_bouts(
    sleep_column = 'asleep',
    facet_col = 'species',
    facet_arg = ['D.vir', 'D.ere'],
    facet_labels = ['D.virilis', 'D.erecta'],
    bin_size = 5,
    max_bins = 12
)
fig.show()
```



Find bouts of sleep

```
# If you haven't already analysed the dataset to find periods of sleep,0
# but you do have a column containing the movement as True/False.
# Call this method to find contiguous bouts of sleep according to a minimum length

new_df = df.sleep_contiguous(time_window_length = 60, min_time_immobile = 300)
```

Sleep download functions as methods

```
# some of the download functions mentioned previously can be called as methods if the data wasn't previously
analysed
# don't call this method if your data was already analysed!
# If it's already analysed it will be missing columns needed for this method

new_df = df.motion_detector(time_window_length = 60)
```

Visualising your data

Once the behavpy object is created, the print function will just show your data structure. If you want to see your data and the metadata at once, use the built in method `.display()`

```
# first load your data and create a behavpy instance of it
```

```
df.display()
```

```
==== METADATA ====

id          date  machine_name  region_id  species          strain  baseline_days  strain_no  exp  time
2019-08-02_14-21-23_021d6b|01 2019-08-02  ETHOSCOPE_021  1    D.vir          15010-1051.87_#9  0      1    species  14-21-23
2019-08-02_14-21-23_021d6b|02 2019-08-02  ETHOSCOPE_021  2    D.vir          15010-1051.87_#9  0      1    species  14-21-23
2019-08-02_14-21-23_021d6b|03 2019-08-02  ETHOSCOPE_021  3    D.vir          15010-1051.87_#9  0      1    species  14-21-23
2019-08-02_14-21-23_021d6b|04 2019-08-02  ETHOSCOPE_021  4    D.vir          15010-1051.87_#9  0      1    species  14-21-23
2019-08-02_14-21-23_021d6b|05 2019-08-02  ETHOSCOPE_021  5    D.sec          14021-0248.25_#3  0      1    species  14-21-23
...
2022-02-10_18-34-55_22982f|16 2022-02-10  ETHOSCOPE_229  16   empty          empty              0      1    wildcaught  18-34-55
2022-02-10_18-34-55_22982f|17 2022-02-10  ETHOSCOPE_229  17   wild_d.mel    Darren_obbard_wildcaught_3  0      2    wildcaught  18-34-55
2022-02-10_18-34-55_22982f|18 2022-02-10  ETHOSCOPE_229  18   wild_d.mel    Darren_obbard_wildcaught_3  0      2    wildcaught  18-34-55
2022-02-10_18-34-55_22982f|19 2022-02-10  ETHOSCOPE_229  19   wild_d.mel    Darren_obbard_wildcaught_3  0      2    wildcaught  18-34-55
2022-02-10_18-34-55_22982f|20 2022-02-10  ETHOSCOPE_229  20   wild_d.mel    Darren_obbard_wildcaught_3  0      2    wildcaught  18-34-55

[883 rows x 9 columns]
===== DATA =====

t      x      y      w      h      phi  ...  beam_crosses  moving  micro  walk  is_interpolated  asleep
id
2019-08-02_14-21-23_021d6b|01 19380  0.623679  0.043643  0.027288  0.012393  80.312500  ...  0.0  True  False  True  False  False
2019-08-02_14-21-23_021d6b|01 19440  0.607520  0.043756  0.043559  0.021442  9.265625  ...  0.0  True  True  False  False  False
2019-08-02_14-21-23_021d6b|01 19500  0.605259  0.043394  0.049732  0.024752  25.476190  ...  0.0  True  True  False  False  False
2019-08-02_14-21-23_021d6b|01 19560  0.602718  0.044650  0.057040  0.027235  22.603175  ...  0.0  True  True  False  False  False
2019-08-02_14-21-23_021d6b|01 19620  0.601776  0.046648  0.062721  0.028235  19.476190  ...  0.0  True  True  False  False  False
...
2022-02-10_18-34-55_22982f|20 287520  0.414123  0.036036  0.050669  0.025371  72.646465  ...  0.0  False  False  False  False  True
2022-02-10_18-34-55_22982f|20 287580  0.413698  0.036036  0.050494  0.025421  55.156627  ...  0.0  False  False  False  False  True
2022-02-10_18-34-55_22982f|20 287640  0.412763  0.036036  0.050429  0.025676  38.297619  ...  0.0  False  False  False  False  True
2022-02-10_18-34-55_22982f|20 287700  0.412742  0.036036  0.050434  0.025631  47.198198  ...  0.0  False  False  False  False  True
2022-02-10_18-34-55_22982f|20 287760  0.413639  0.036036  0.050177  0.025636  91.151899  ...  0.0  False  False  False  False  True
```

You can also get quick summary statistics of your dataset with `.summary()`

```
df.summary()
```

```
# an example output of df.summary()
```

output:

behavpy table with:

```
individuals      675
metavariable      9
variables        13
measurements 3370075
```

```
# add the argument detailed = True to get information per fly
```

```
df.summary(detailed = True)
```

output:

	data_points	time_range
id		
2019-08-02_14-21-23_021d6b 01	5756	86400 -> 431940
2019-08-02_14-21-23_021d6b 02	5481	86400 -> 431940

Be careful with the pandas method `.groupby()` as this will return a pandas object back and not a behavpy object. Most other common pandas actions will return a behavpy object.

Visualising your data

Whilst summary statistics are good for a basic overview, visualising the variable of interest over time is usually a lot more informative.

Heatmaps

The first port of call when looking at time series data is to create a heatmap to see if there are any obvious irregularities in your experiments.

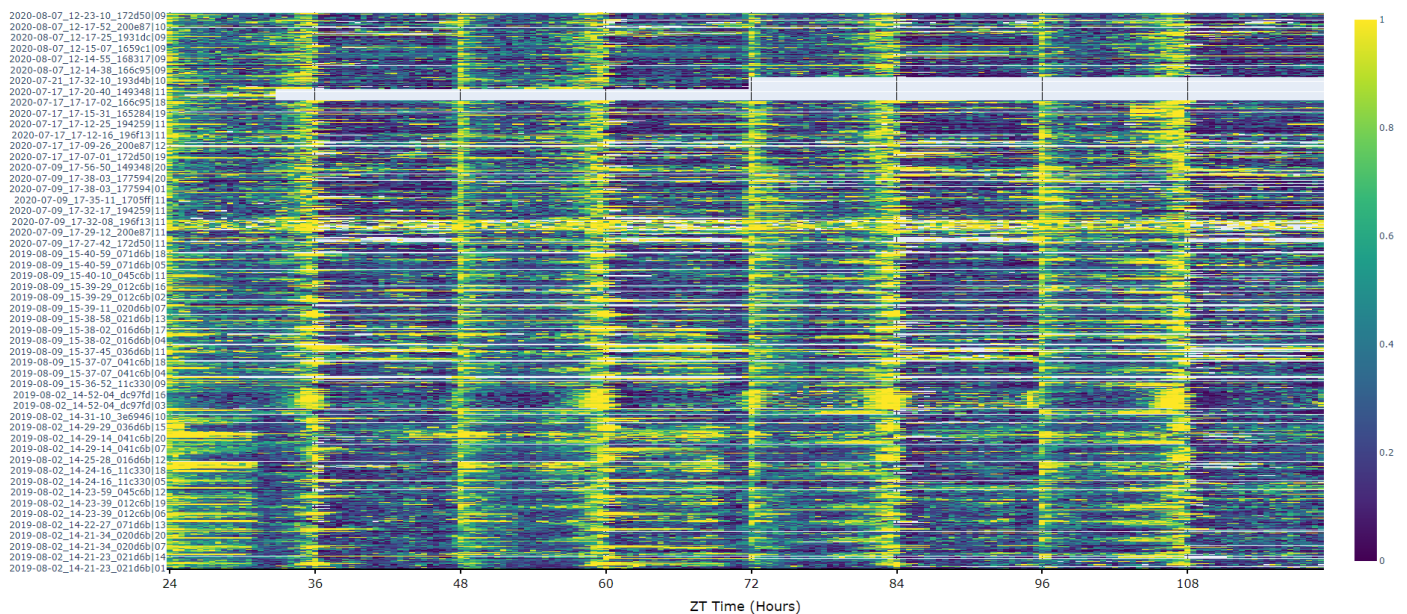
```
# To create a heatmap all you need to write is one line of code!
```

```
# All plot methods will return the figure, the usual etiquette is to save the variable as fig
```

```
fig = df.heatmap('moving') # enter as a string which ever numerical variable you want plotted inside the brackets
```

```
# Then all you need to do is the below to generate the figure
```

```
fig.show()
```



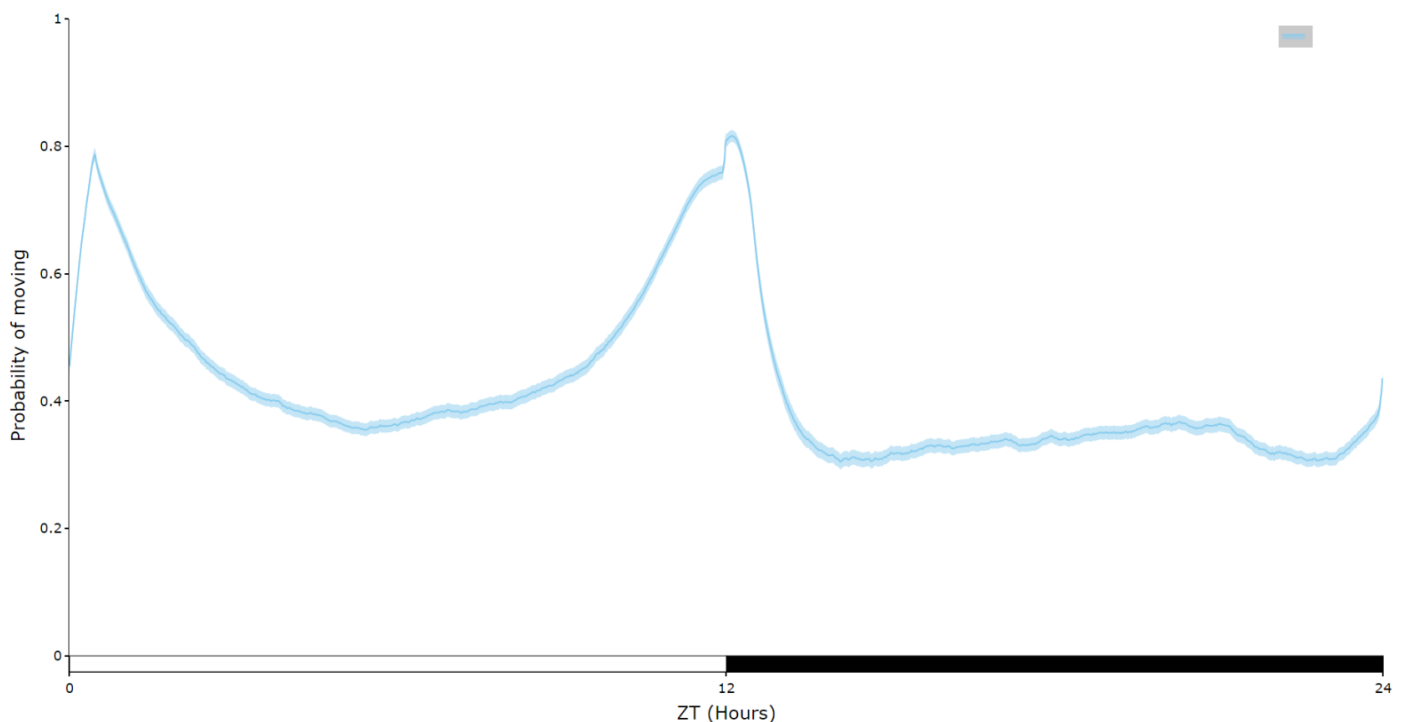
Plots over time

For an aggregate view of your variable of interest over time, use the `.plot_overtime()` method to visualise the mean variable over your given time frame or split it into sub groups using the information in your metadata.

```
# If wrapped is True each specimens data will be aggregated to one 24 day before being aggregated as a whole.
# If you want to view each day seperately, keep wrapped False.
# To achieve the smooth plot a moving average is applied, we found averaging over 30 minutes gave the best
# results
# So if you have your data in rows of 10 seconds you would want the avg_window to be 180 (the default)
# Here the data is rows of 60 seconds, so we only need 30

fig = df.plot_overtime(
    variable = 'moving',
    wrapped = True,
    avg_window = 30
)
fig.show()

# the plots will show the mean with 95% confidence intervals in a lighter colour around the mean
```



```
# You can separate out your plots by your specimen labels in the metadata. Specify which column you want
# from the metadata with facet_col and then specify which groups you want with facet_args
# What you enter for facet_args must be in a list and be exactly what is in that column in the metadata
# Don't like the label names in the column, rename the graphing labels with the facet_labels parameter. This
# can only be done if you have a same length list for facet_arg. Also make sure they are the same order
```

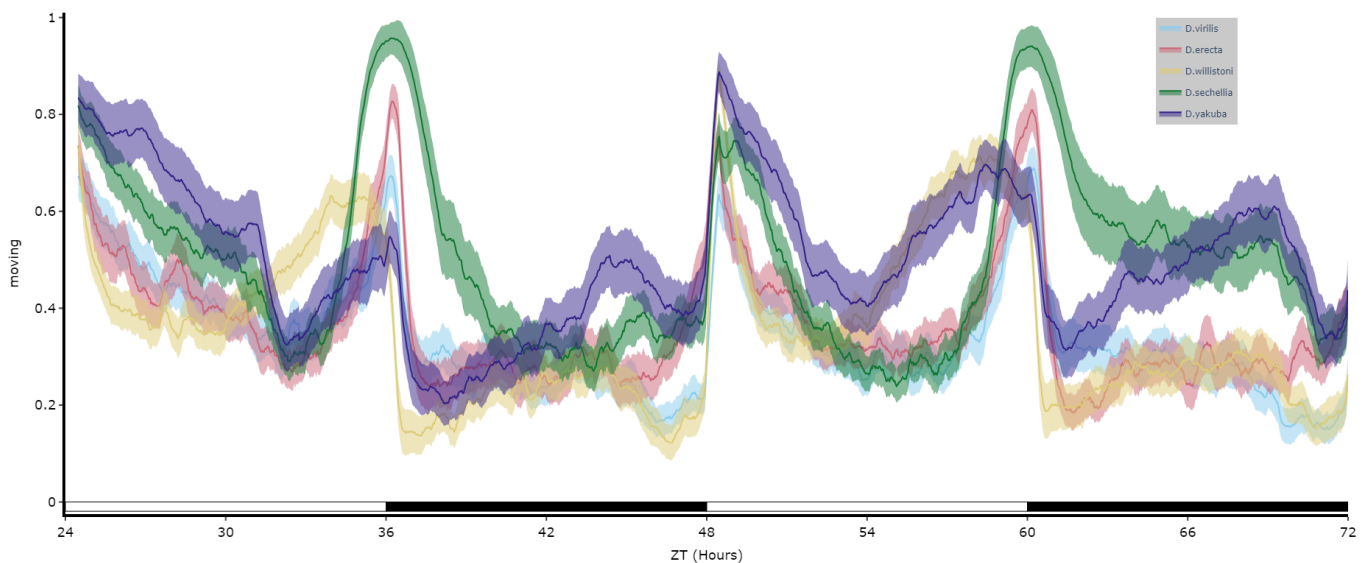
```
fig = df.plot_overtime(
```

```

variable = 'moving',
facet_col = 'species',
facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak'],
facet_labels = ['D.virilis', 'D.erecta', 'D.willistoni', 'D.sechellia', 'D.yakuba']
)
fig.show()

```

if you're doing circadian experiments you can specify when night begins with the parameter `circadian_night` to change the phase bars at the bottom. E.g. `circadian_night = 18` for lights off at ZT 18.



Quantifying sleep

The plots above look nice, but we often want to quantify the differences to prove what our eyes are telling us. The method `.plot_quantify()` plots the mean and 95% CI of the specimens to give a strong visual indication of group differences.

```

# Quantify parameters are near identical to .plot_overtime()
# For all plotting methods in behavpy we can add grid lines to better view the data and you can also add a title,
see below for how
# All quantifying plots still return two objects, the first will be the plotly figure as normal and the second a
pandas dataframe with
# the calculated averages per specimen for you to perform statistics on. You can do this via the common
statistical package scipy or
# we recommend a new package DABEST, that produces non p-value related statistics and visualtion too.
# We'll be looking to add DABEST into our ecosystem soon too!

```

```

fig, stats_df = df.plot_quantify(
variable = 'moving',

```

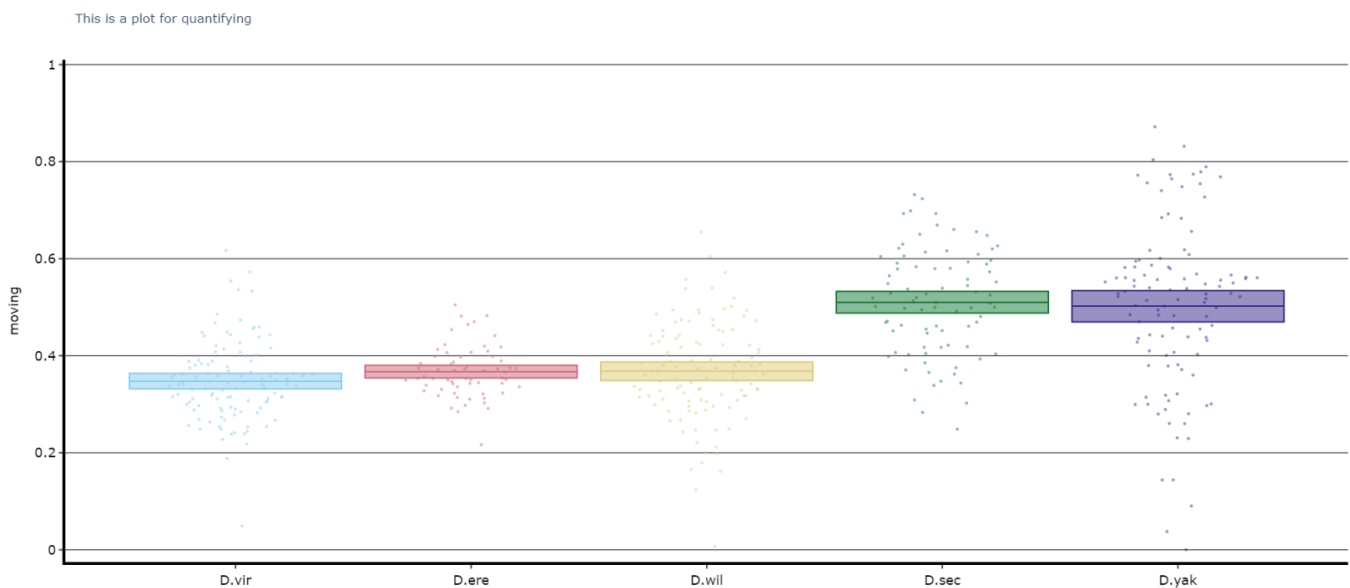


```

facet_col = 'species',
facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak']
title = 'This is a plot for quantifying',
grids = True
)
fig.show()

```

Tip! You can have a facet_col argument and nothing for facet_args and the method will automatically plot all groups within that column. Careful though as the order/colour might not be preserved in similar but different plots



Performing Statistical tests

If you want to perform normal statistical tests then it's best to use Scipy, a very common python package for statistics with lots of online tutorial on how and when to use it. We'll run through a demonstration of one below. Below you'll see an example table of the data from the plot above.

	D.vir	D.ere	D.wil	D.sec	D.yak
0	0.279889	0.305839	0.486261	0.484013	0.523901
1	0.294987	0.403151	0.386917	0.584337	0.627201
2	0.354461	0.383920	0.460455	0.439296	0.550845
3	0.371630	0.385240	0.565994	0.440222	0.535946
4	0.271885	0.372018	0.394810	0.491663	0.582313
...
107	0.263657	NaN	0.385701	NaN	0.291204
108	0.382526	NaN	0.349498	NaN	0.455093
109	0.337037	NaN	NaN	NaN	NaN
110	0.406380	NaN	NaN	NaN	NaN
111	0.268494	NaN	NaN	NaN	NaN

112 rows × 5 columns

Given we don't know the distribution type of the data and that each group is independent we'll use a non-parametric group test, the Kruskal-Wallis H-test (the non-parametric version of a one-way ANOVA).

```
# First we need to import scipy stats
from scipy import stats

# Next we need the data for each specimen in a numpy array
# Here we iterate through the columns and send each column as a numpy array into a list
stat_list = []
for col in stats_df.columns.tolist():
    stat_list.append(stats_df[col].to_numpy())

# Alternatively you can set each one as it's own variable
dv = stats_df['D.vir'].to_numpy()
de = stats_df['D.ere'].to_numpy()
... etc

# Now we call the Kruskal function, remember to always have the nan_policy set as 'omit' otherwise the output
will be a NaN

# If using the first method (the * unpacks the list)
stats.kruskal(*stat_list, nan_policy = 'omit')

# If using the second
stats.kruskal(dv, de, ..., nan_policy = 'omit')
```

```
## output: KruskalResult(statistic=134.17956605297556, pvalue=4.970034343600611e-28)
```

```
# The pvalue is below 0.05 so we can say that not all the groups have the same distribution
```

```
# Now we want to do some post hoc testing to find inter group significance
```

```
# For that we need another package scikit_posthocs
```

```
import scikit_posthocs as sp
```

```
p_values = sp.posthoc_dunn(stat_list, p_adjust = 'holm')
```

```
print(p_values)
```

```
## output: 1 2 3 4 5
```

```
1 1.000000e+00 2.467299e-01 0.000311 1.096731e-11 1.280937e-16
```

```
2 2.467299e-01 1.000000e+00 0.183924 1.779848e-05 7.958752e-08
```

```
3 3.106113e-04 1.839239e-01 1.000000 3.079278e-03 3.884385e-05
```

```
4 1.096731e-11 1.779848e-05 0.003079 1.000000e+00 4.063266e-01
```

```
5 1.280937e-16 7.958752e-08 0.000039 4.063266e-01 1.000000e+00
```

```
print(p_values > 0.05)
```

```
## output: 1 2 3 4 5
```

```
1 True True False False False
```

```
2 True True True False False
```

```
3 False True True False False
```

```
4 False False False True True
```

```
5 False False False True True
```

Quantify day and night

Often you'll want to compare a variable between the day and night, particularly total sleep. This variation of `.plot_quantify()` will plot the mean and 96% CI of a variable for a user defined day and night.

```
# Quantify parameters are near identical to .plot_quantify() with the addition of day_length and lights_off which takes (in hours) how long the day is for the specimen (defaults 24) and at what point within that day the lights turn off (night, defaults 12)
```

```
fig, stats = df.plot_day_night(
```

```
variable = 'asleep',
```

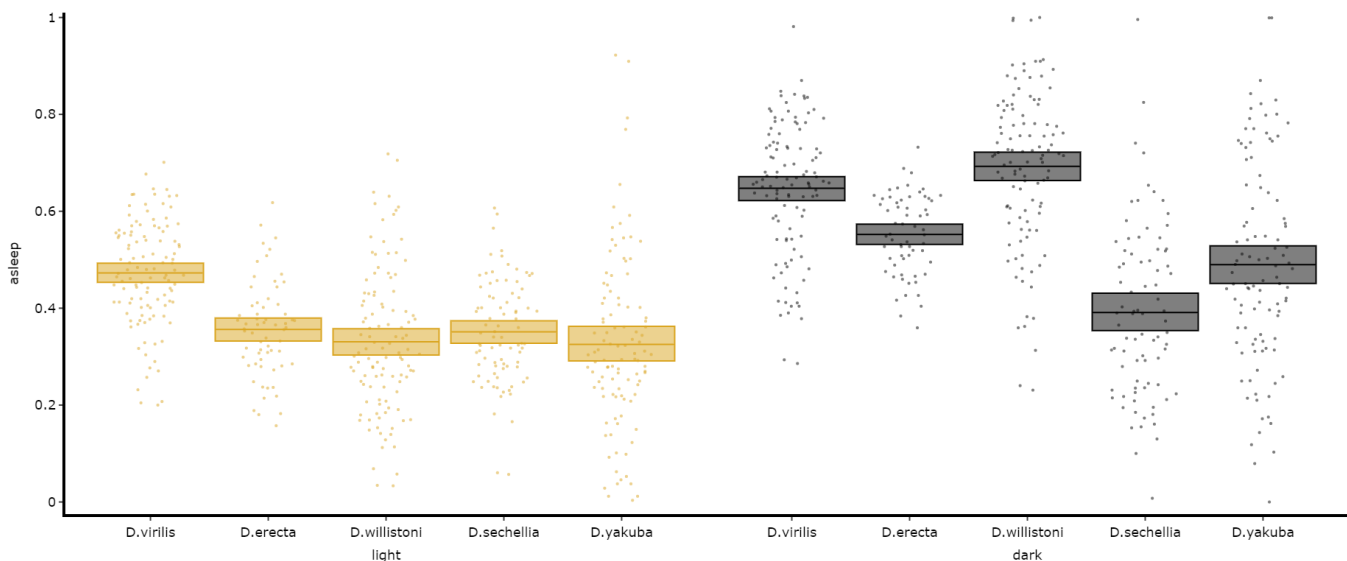
```
facet_col = 'species',
```

```
facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak'],
```

```
facet_labels = ['D.virilis', 'D.erecta', 'D.willistoni', 'D.sechellia', 'D.yakuba'],
```

```
day_length = 24,
```

```
lights_off = 12
)
fig.show()
```



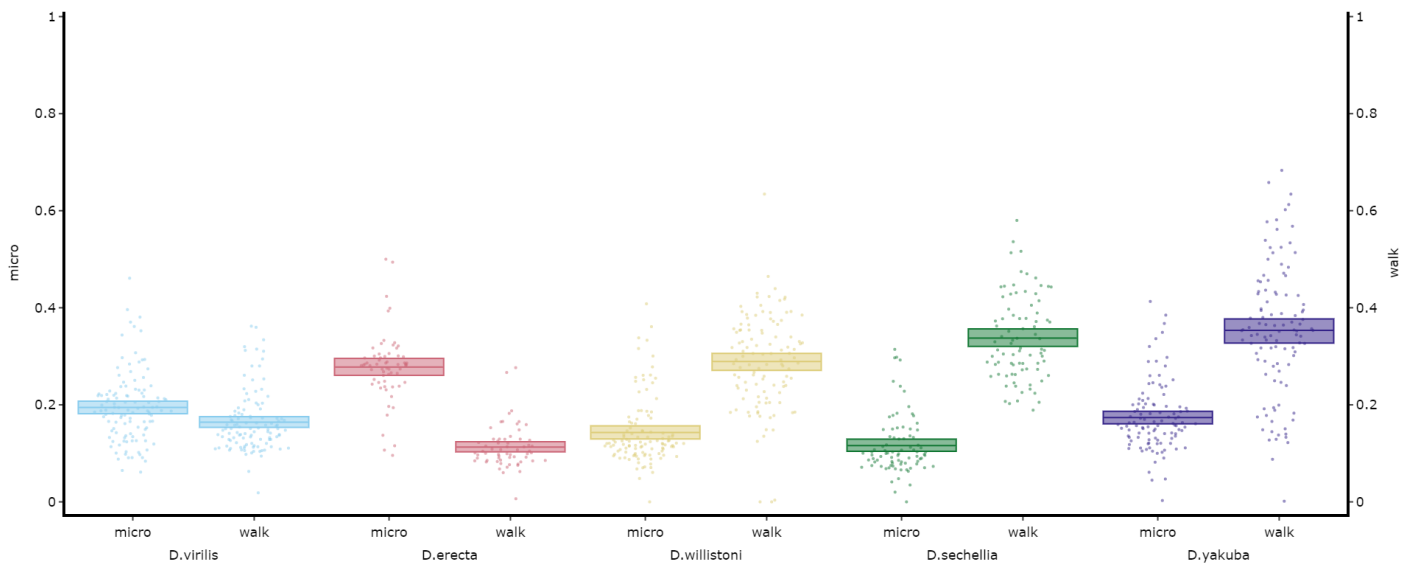
Compare variables

You may sometimes want to compare different but similar variables from your experiment, often comparing them across different experimental groups. Use `.plot_compare_variables()` to compare up to 24 different variables (we run out of colours after that...).

```
# Like the others it takes the regular arguments. However, rather than the variable parameter it's variables,
which takes a list of strings of the different columns you want to compare. The final variable in the list will have
it's own y-axis on the right-hand side, so save this one for different scales.
```

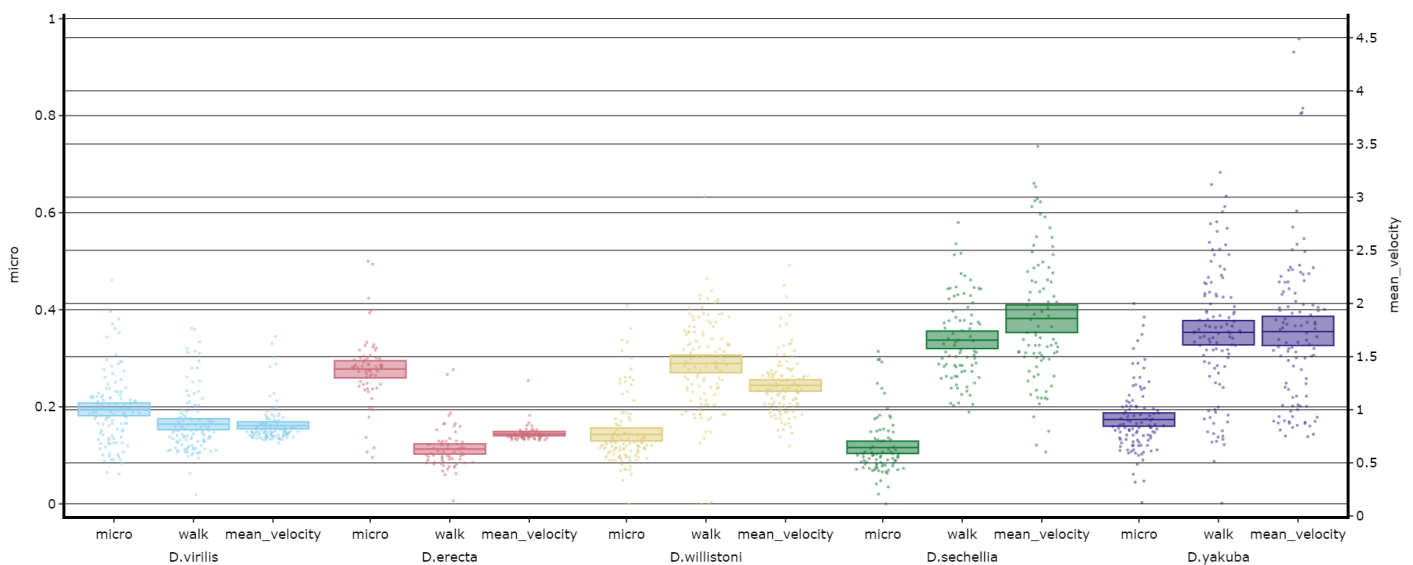
```
# The most often use of this with the ethoscope data is to compare micro movements to walking movements to
better understand the locomotion of the specimen, so lets look at that.
```

```
fig, stats = df.plot_compare_variables(
    variables = ['mirco', 'walk']
    facet_col = 'species',
    facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak'],
    facet_labels = ['D.virilis', 'D.erecta', 'D.willistoni', 'D.sechellia', 'D.yakuba']
)
fig.show()
```



Lets add the velocity of the specimen to see how it looks with different scales.

```
fig, stats = df.plot_compare_variables(
    variables = ['micro', 'walk', 'mean_velocity']
    facet_col = 'species',
    facet_arg = ['D.vir', 'D.ere', 'D.wil', 'D.sec', 'D.yak', 'D.sims'],
    facet_labels = ['D.virilis', 'D.erecta', 'D.willistoni', 'D.sechellia', 'D.yakuba', 'D.Simulans'],
    # Add grids to the plot so we can better see how the plots align
    grids = True
)
fig.show()
```



Head to the Overview Tutorial for interactive examples of some of these plots. It also shows you how to edit the fig object after it's been generated if you wish to change things like axis titles and axis range.

Saving plots

As seen above all plot methods will produce a figure structure that you use `.show()` to generate the plot. To save these plots all we need to do is place the fig object inside a behavpy function called `etho.save_figure()`

You can save your plots as PDFs, SVGs, JPEGs or PNGS. It's recommended you save it as either a pdf or svg for the best quality and then the ability to alter then later.

You can also save the plot as a html file. Html files retain the interactive nature of the plotly plots. This comes in most useful when using jupyter notebooks and you want to see your plot full screen, rather than just plotted under the cell.

```
# simply have you returned fig as the argument alongside its save path and name to save

# remember to change the file type at the end of path as you want it
# you can set the width and height of the saved image with the parameters width and height (this has no effect
for .html files)
etho.save_figure(fig, './tutorial_fig.pdf', width = 1500, height = 1000)

# or to get a better view of it and keep the interactability save it as a html file
etho.save_figure(fig, './tutorial_fig.html') # you can't change the height and width wehn saved as a html
```

Visualising mAGO data

Within the Gilestro lab we have special adaptations to the ethoscope which includes the mAGO, a module that can sleep deprive flies manually and also deliver a puff of an odour of choice after periods of rest. See the documentation here: [ethoscope_mAGO](#).

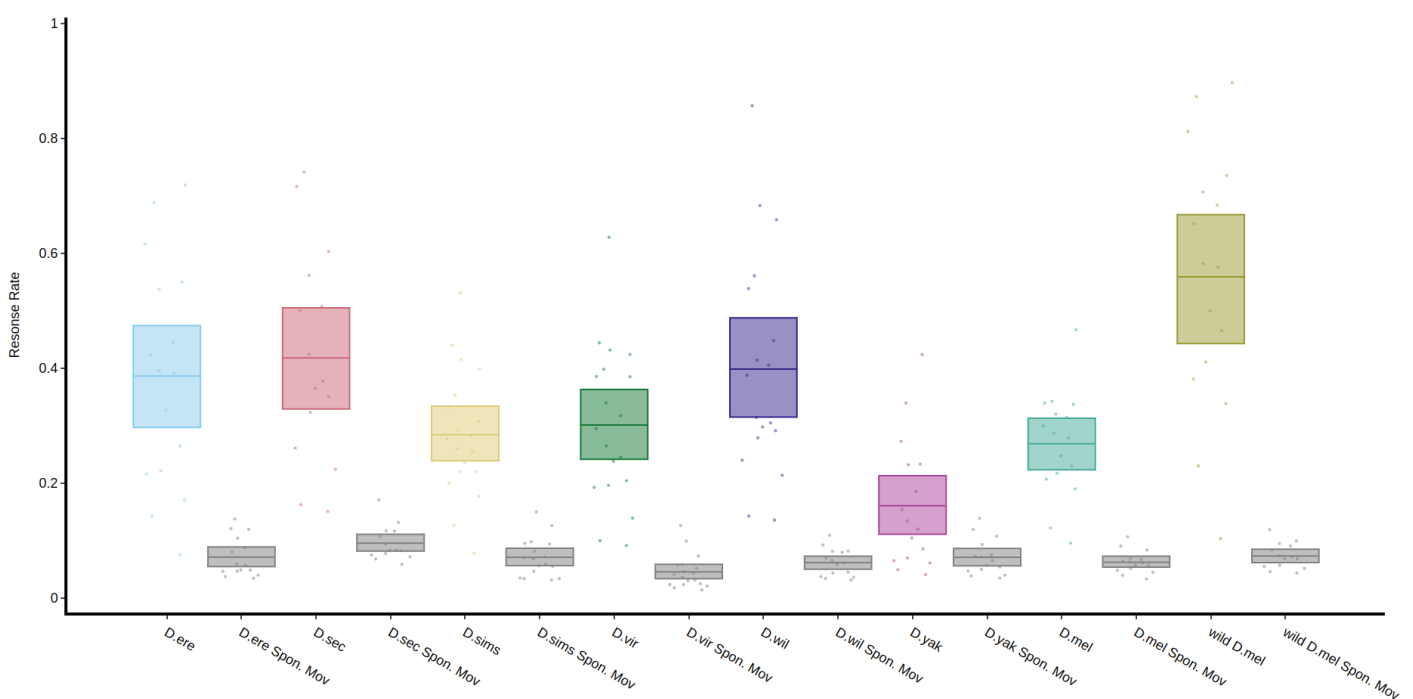
If you've performed a mAGO experiment then the data needs to be loaded in with the function `puff_mago()` which decides if a fly has responded if it's moved with the time limit post puff (default is 10 seconds).

Quantify response

The method `.plot_response_quantify()` will produce a plot of the mean response rate per group. Within the software for delivering a puff it can be set to have a random chance of occurring. I.e. if it's set to 50%, and the immobility criteria is met then the chance a puff will be delivered is 50/50. This gives us two sets of data, true response rate and the underlying likelihood a fly will just randomly move, which is called here Spontaneous Movement (Spon. Mov).

```
# The parameters other than response_col (which is the column of the response per puff) are the same as other  
quantify methods
```

```
fig, stats = df.plot_response_quantify(  
    response_col = 'has_responded',  
    facet_col = 'species',  
)  
fig.show()
```



Quantify response overtime

You can also view how the response rate changes over time with `.plot_response_overtime()`. For this method you'll need to load in the normal dataset, i.e. with `motion_detector` or `sleep_annotation` as the method needs to know at what point the puff occurred. The plot is per minute, so it's best to load it in with a time window of 60. If you have it higher the method won't work.

```
# Call the method on the normal behavpy table with the first argument the puff behavpy table
# The seconds argument decides if you're looking at runs of inactivty or activity,
# if you set the puff chance low enough you can probe activity too

fig, stats = df.plot_response_overtime(
    response_df = puff_df,
    activity = 'inactive',
    facet_col = 'species'
)
fig.show()
```