

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 =====
```

id	t	x	y	w	h	phi	...	beam_crosses	moving	micro	walk	is_interpolated	asleep
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

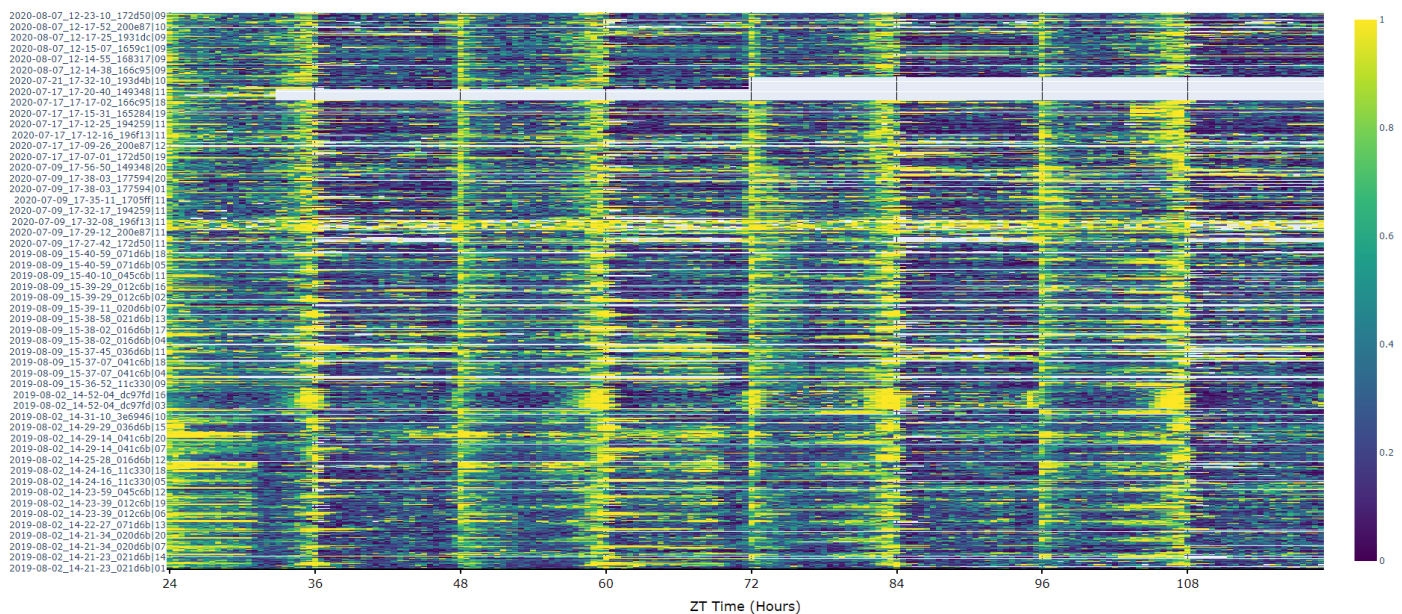
The first port of call when looking at time series data is to create a heatmap to see if there are any obvious irregularities in your experiments.

```
# 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()
```

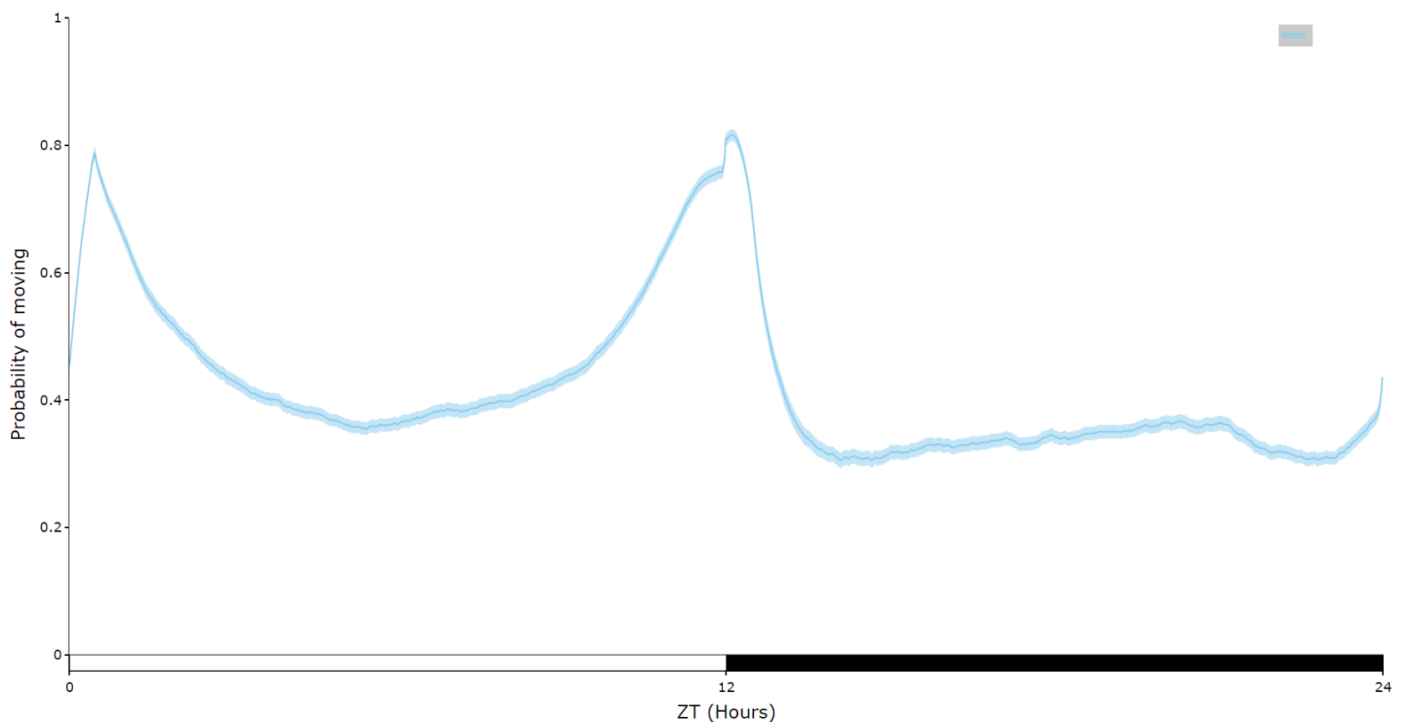


Plots over time

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
```

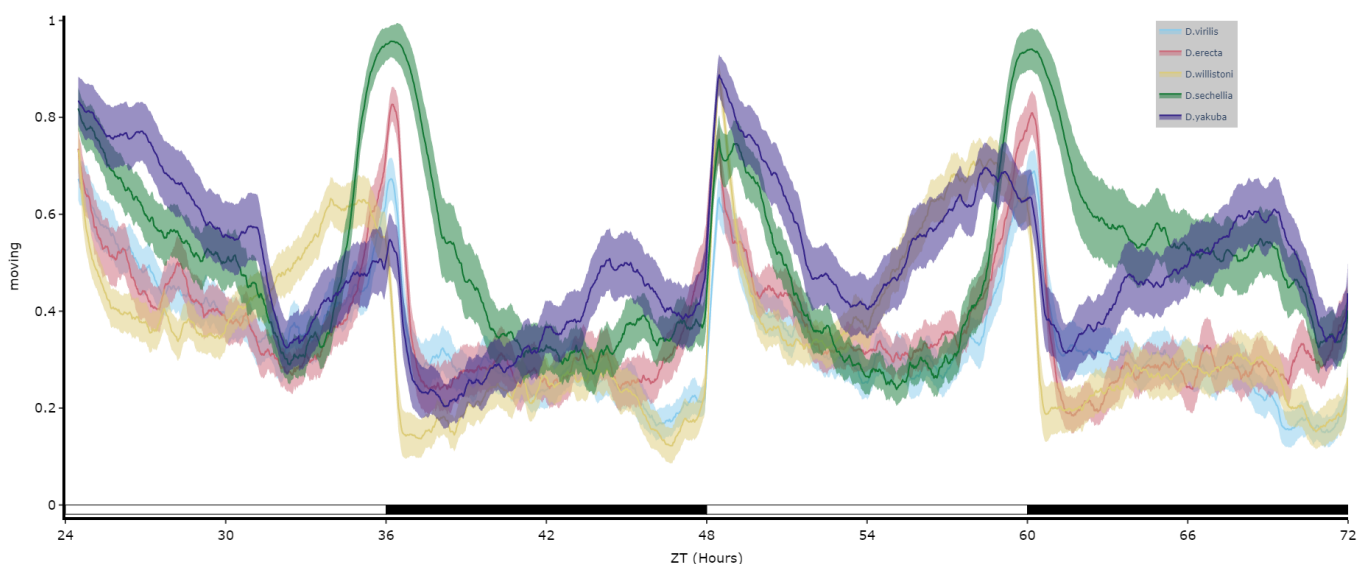


```
# You can separate out your plots by your specimen labels in the metadata. Specify which
column you want fromn the metadata with facet_col and then specify which groups you want with
facet_args
# What you enter for facet_args must be in a list and be exactly what is in that column in the
metadata
```

```
# 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 sill 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
```

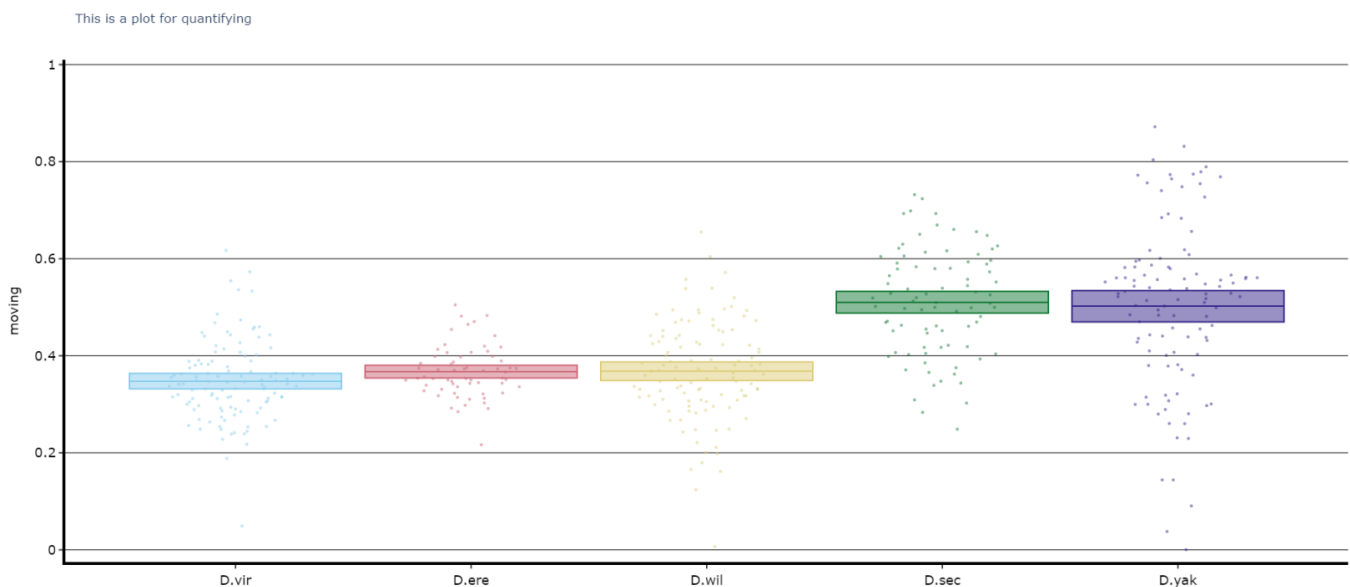
the common statistical package scipy or

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 ecosytem 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
```

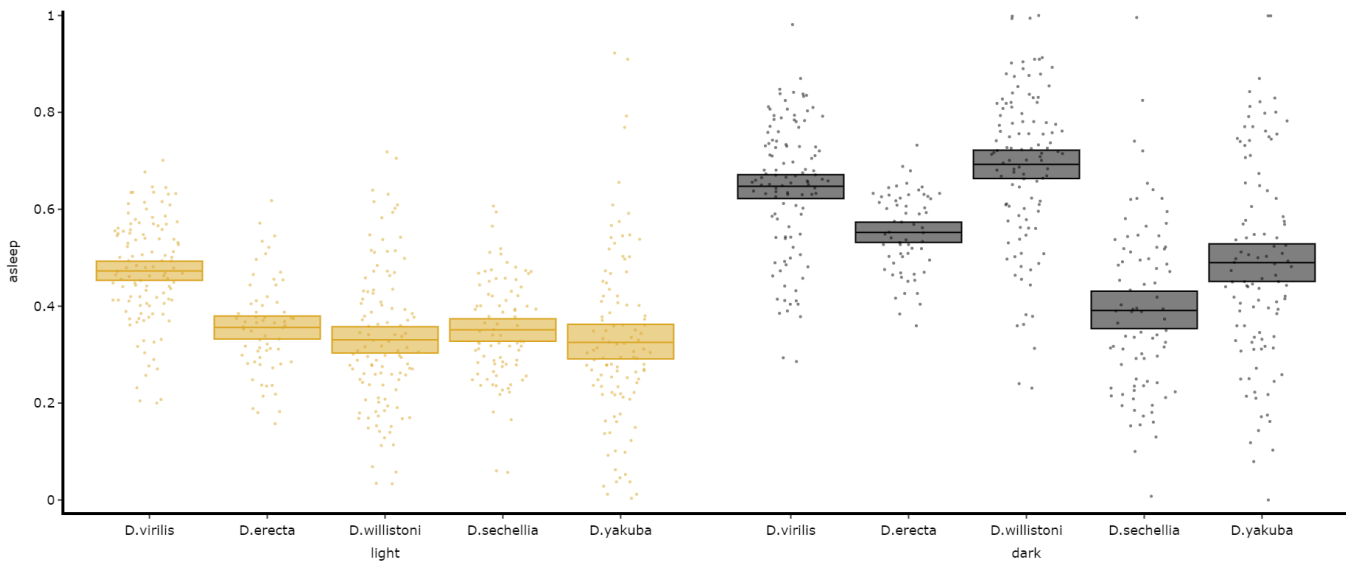
Quantify day and night

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()
```



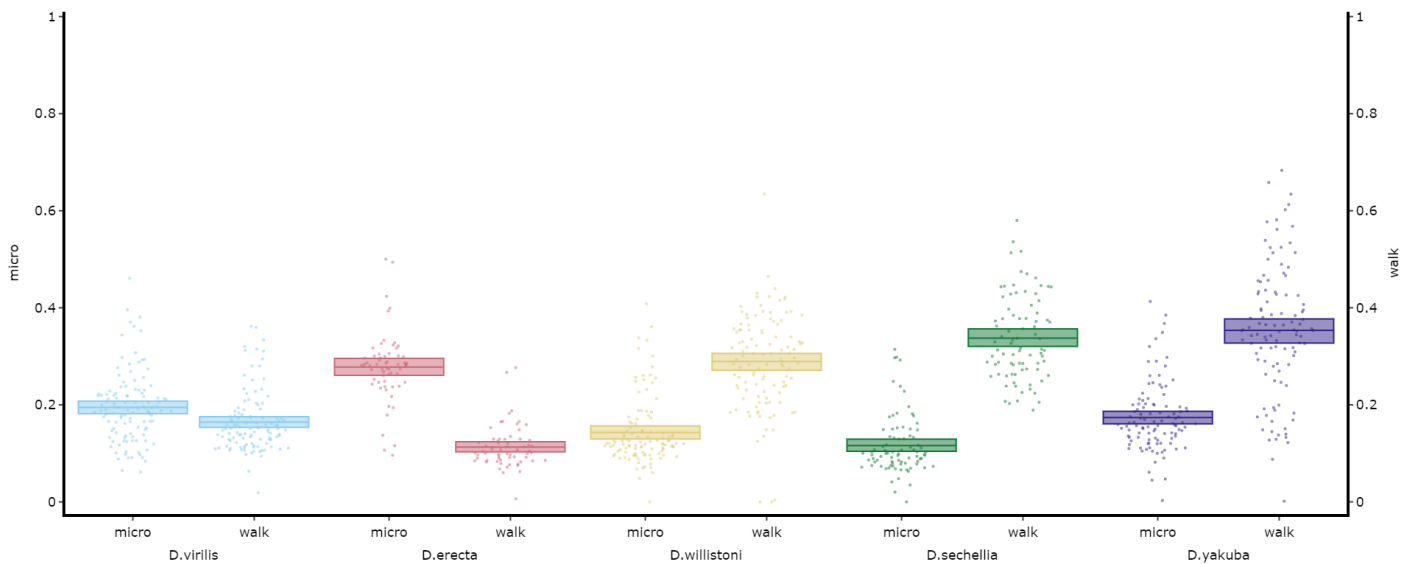
Compare variables

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.
```

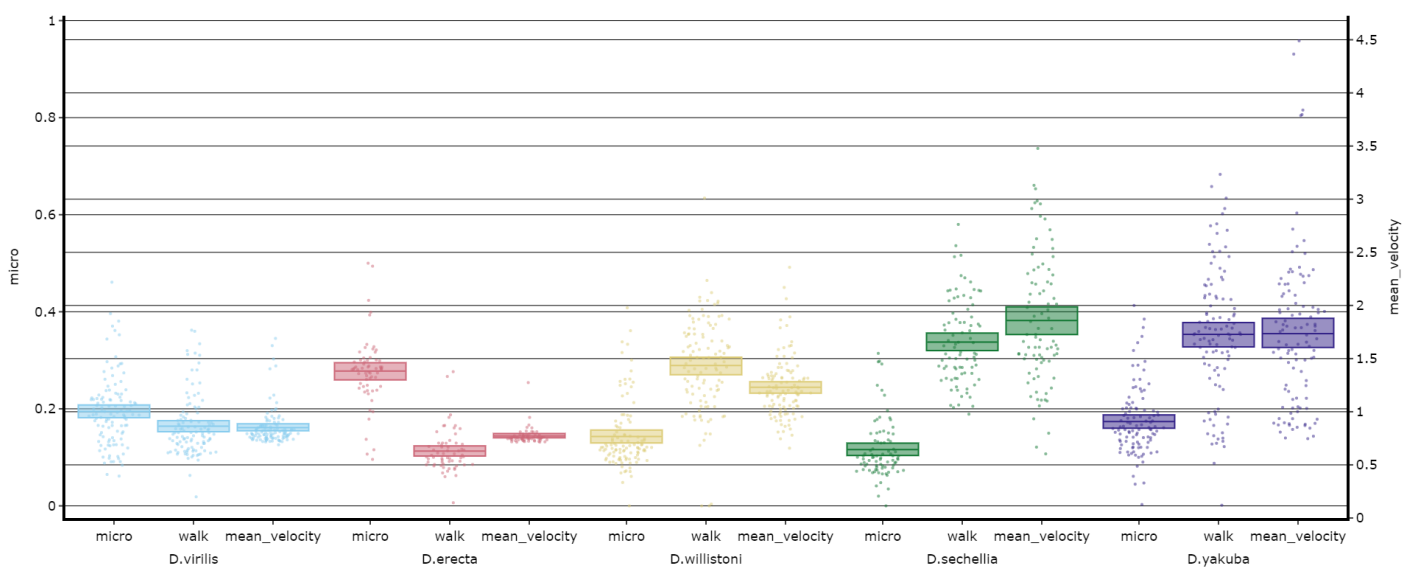
```
# The most often use of this with the ethoscope data is to compare micro movements to walking
movements to better understand the locomotion of the specimen, so lets look at that.
```

```
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
```

Revision #6

Created 23 November 2022 08:46:46 by Giorgio Gilestro

Updated 13 June 2023 11:58:45 by Laurence Blackhurst