worklab

Release 1.7.1

R. de Klerk, T. Rietveld, R.J.F. Janssen

Jun 03, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

The current page contains information on wheelchair propulsion research in the worklab and the correspondingly named Python package. As such, the page contains a general outline of wheelchair propulsion research, but also some practical examples on how to work with that data together with the API-reference of the worklab package. The worklab package aims to provide functions for the most common data (pre-)processing steps in wheelchair research (at our lab). It makes extensive use of pandas dataframes as those are familiar to most researchers.

CHAPTER

TWO

CONTENTS

2.1 Theory

2.1.1 Wheelchair vehicle mechanics

2.1.2 The power balance

2.1.3 References

2.2 Overground

In wheelchair propulsion the main source of energy loss is rolling friction, which depends on the wheelchair, user, and wheelchair-user interface. At high speeds aerodynamic drag rapidly increases as it is a non-linear function of speed. Wheelchair design, mechanics, and maintenance can significantly alter the resistive forces acting on the wheelchair and user. In an everyday setting these forces are to be minimised to reduce strain and, in a sports setting, they have to be minimised to optimize performance. The forces can either be identified through a drag-test or with the coasting deceleration method. This document will give a short overview of rolling resistance, air drag, and how to determine them. It will also provide some reference values where possible that can be used for simulation on a treadmill or wheelchair ergometer. It aims to be as concise as possible.

2.2.1 Determining power output

In a coast-down test you let the wheelchair coast-down without the user applying force on the handrims. The user should assume a position that is representative of the position during wheelchair propulsion. The deceleration is reflective of the frictional forces acting on the wheelchair-user combination (Figure 2). Usually, this is done in a back-and-forth manner to compensate for factors such as uneven ground. The coast-down test is a common procedure to assess the dissipative forces and their coefficients. A number of instrumentation options is available (slope, time-gates, velocity), but here we will only discuss the method where velocity data is available as many affordable technologies for gathering velocity data are currently available. This data could, for example, be gathered with a tachometer, measurement wheel, or an IMU.

Protocol

For a regular handrim wheelchair the following protocol can be used:

- The test subject should remain upright in a neutral position
- The experimenter accelerates the wheelchair to a “high” velocity

- The wheelchair should decelerate to a complete standstill without interference
- Time and velocity data should be measured for the deceleration period

Analysis

When we assume a constant friction, it is relatively easy to determine the frictional forces. This assumption is safe to make at relatively low speeds, but does become dangerous for most sports applications. A linear regression should be fit to the linear deceleration. Knowing that there is no outside force acting on the wheels other than friction, it is relatively easy to determine the total frictional force. The coefficient of friction can then be extracted with:

$$\mu = \frac{Ma}{mg}$$

An open-source implementation for the analysis of coast-down data has been developed at the University Medical Centre Groningen (Coast-down analyzer).

When constant friction cannot be assumed due to air drag, i.e. in most sports environments, the analysis becomes a little more complex. The protocol is identical; however, initial speed should probably be higher. In this case, a non-linear differential equation needs to be solved and that equation needs to be fit with a curve fitter (e.g. Levenberg-Marquardt).

2.2.2 IMUs

2.2.3 References

2.3 Treadmill

2.3.1 Determining power output

2.3.2 Setting power output

2.3.3 Calibrations

Tachometer

Angle sensor

2.3.4 References

2.4 Ergometer

This guide will explain how the settings of the Esseda wheelchair ergometer can be used to simulate overground propulsion or set a target power output for your participants. It assumes a properly operated and calibrated system.

The Esseda is a wheelchair roller ergometer developed and produced by Lode BV (Groningen, The Netherlands). It is equipped with a set of servomotors and loadcells to simulate and measure wheelchair propulsion. With the servomotors in the ergometer you can make propulsion as light or as heavy as you would like it to be.

A simple mechanical model is used to simulate overground propulsion. This model simulates:

- The inertia of the participant + the wheelchair

- The rolling resistance of the wheelchair

Both values can be changed accordingly in LEM. Weight can only be set before every measurement. Friction can be changed during measurements to allow for ramp or step protocols. Total friction (left + right module) is calculated by multiplying the weight of the participant and the wheelchair (m) with the gravitational constant (g) and a rolling resistance constant ():

$$F_{friction} = m * g * \mu$$

2.4.1 Setting power output

Oftentimes researchers are more interested in power output (W). During steady-state propulsion, the velocity of the wheelchair oscillates around a steady point with every push. Additionally, simulated friction is assumed to be independent of velocity. As such, power output can be assumed to be the product of the frictional force (F) and mean velocity (v):

$$P_{out} = F_{friction} * v$$

As such, the expected power output on the Esseda wheelchair ergometer can be calculated with relative ease.

2.4.2 Calibrations

Drift

Noise

Calibration

Acceleration

2.4.3 References

2.5 General testing

2.5.1 Measurement wheels

2.5.2 Physiology

2.5.3 Kinematics

2.5.4 References

2.6 Examples

This page contains some commonly requested examples in Jupyter notebooks. The general approach to processing the data used in this package is usually fairly similar:

- load the data
- filter the data

- pre-process the data
- reduce the data (e.g. by slicing and/or calculating means)

Loading the data can usually be performed by using the *.com.load* function. These functions usually don't do much with the data except maybe convert some columns to the SI standard.

Filtering the data is very much up to your own preferences and needs, I do not know what the contents of your signal are, but there are some functions available in the package to make this easier.

Pre-processing is largely taken care of. It could be that you need an extra variable such as, I don't know, the third derivative of velocity. You would need to compute those yourself. If you feel like a variable is really missing you are welcome to contact me and/or send a pull request and I will include it.

Reducing the data is very much up to yourself. Given that the data is in a pandas dataframe this should be straightforward. Pandas dataframes have excellent support for slicing operations and getting the basic statistics is as easy as calling the .describe method on said dataframe.

We could go on for days, but it's usually best to show an example:

2.6.1 Minimal example for working with ergometer data

Import the worklab module:

[1]:
```
import os
import worklab as wl
```

Import the data with com.load() or the device specific load function:

[2]:
```
filename = os.getcwd()
filename = os.path.join(os.path.split(filename)[0], 'example_data', 'Esseda_example_
↪LEM.xls')
ergo_data = wl.com.load(filename)
print("Ergometer data is stored in a: ", type(ergo_data))
```

```
================================================================================
Initializing loading for C:\Users\rick_\Development\worklab2\example_data\Esseda_
↪example_LEM.xls ...
File identified as Esseda datafile. Attempting to load ...
Data loaded!
================================================================================

Ergometer data is stored in a:  <class 'dict'>
```

[3]:
```
ergo_data.keys()
```

[3]:
```
dict_keys(['left', 'right'])
```

The ergometer data is a little more simple than measurement wheel data, (but you do have two modules):

[4]:
```
ergo_data["left"].head()
```

[4]:

	time	force	speed
0	0.01	0.788487	0.0
1	0.02	0.780378	0.0
2	0.03	0.797562	0.0
3	0.04	0.792799	0.0
4	0.05	0.838543	0.0

Processing is identical however:

- filter
- process
- push-by-push

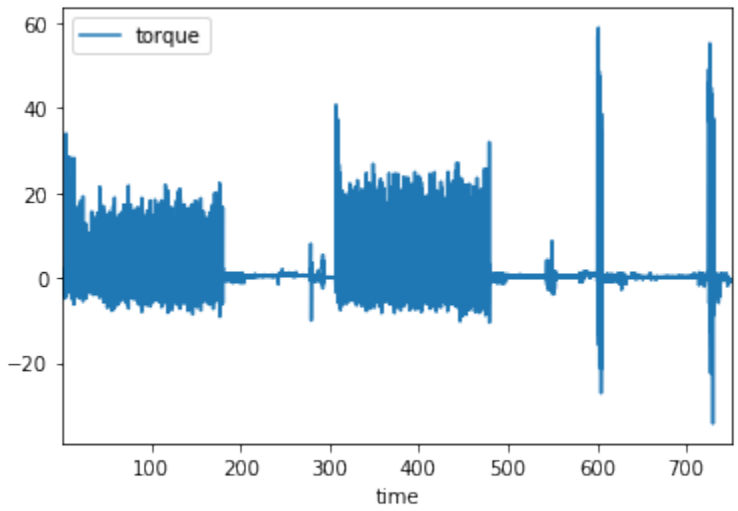
[5]:
```
ergo_data = wl.kin.filter_ergo(ergo_data)
ergo_data = wl.kin.process_ergo(ergo_data)
ergo_data["left"].head()
```

[5]:
```
   time     force          speed    torque           acc          power  \
0  0.01  0.788493 -1.163828e-14  0.244433 -2.799567e-13 -9.176693e-15
1  0.02  0.793392 -1.443784e-14  0.245951 -2.414086e-13 -1.145486e-14
2  0.03  0.802370 -1.646645e-14  0.248735 -1.179231e-13 -1.321219e-14
3  0.04  0.812440 -1.679630e-14  0.251857  1.148631e-13 -1.364600e-14
4  0.05  0.811675 -1.416918e-14  0.251619  4.896777e-13 -1.150078e-14

           dist          work    uforce        aspeed         angle
0  0.000000e+00 -9.176693e-17  0.888846 -3.754282e-14  0.000000e+00
1 -1.303806e-16 -1.145486e-16  0.894369 -4.657369e-14 -4.205825e-16
2 -2.849020e-16 -1.321219e-16  0.904490 -5.311757e-14 -9.190388e-16
3 -4.512158e-16 -1.364600e-16  0.915842 -5.418163e-14 -1.455535e-15
4 -6.060432e-16 -1.150078e-16  0.914979 -4.570705e-14 -1.954978e-15
```

Now you have almost all parameters that you will ever need:

[6]:
```
ergo_data["left"].plot("time", "torque");
```



Let's do a christmas tree for a smaller section of the data:

[7]:
```
ergo_data["left"] = ergo_data["left"].iloc[:3000, :]
ergo_data["right"] = ergo_data["right"].iloc[:3000, :]
```

Get the pushes with the push by push function:

[8]:
```
pushes = wl.kin.push_by_push_ergo(ergo_data)
print(type(pushes))
print(pushes.keys())
pushes["left"].head()
```

```
<class 'dict'>
dict_keys(['left', 'right'])
```

[8]:

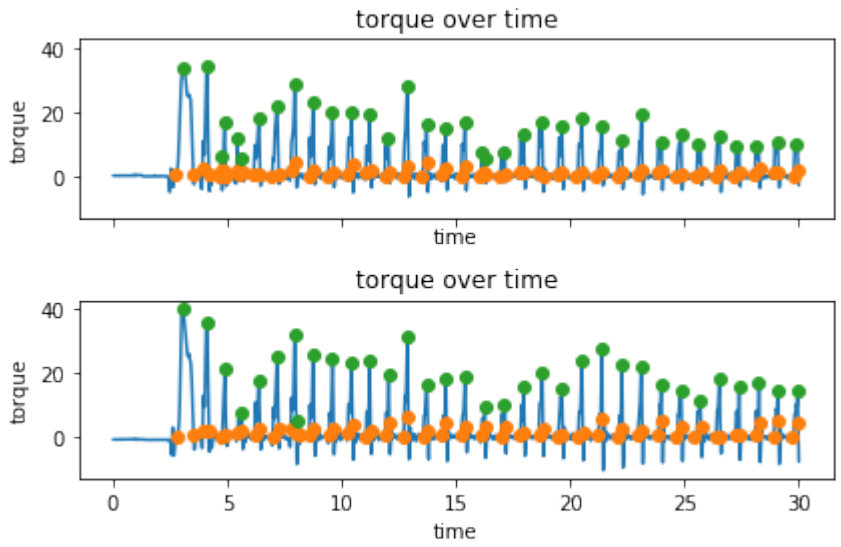
	stop	start	peak	tstart	tstop	tpeak	cangle	ptime	meanpower	\
0	351	276	311	2.77	3.52	3.12	1.180683	0.75	34.260157	
1	419	391	409	3.92	4.20	4.10	1.122666	0.28	67.705997	
2	474	468	472	4.69	4.75	4.73	0.272526	0.06	15.354861	
3	493	480	487	4.81	4.94	4.88	0.599194	0.13	44.567863	
4	547	540	543	5.41	5.48	5.44	0.319896	0.07	31.561252	

	maxpower	meantorque	maxtorque	meanforce	maxforce	work	\
0	77.243329	20.808663	33.636038	75.667865	122.312864	26.037719	
1	142.440912	16.666537	34.028533	60.605589	123.740121	19.634739	
2	27.136431	3.381357	5.975240	12.295844	21.728147	1.074840	
3	79.093543	9.648388	17.036661	35.085049	61.951494	6.239501	
4	54.087009	6.901818	11.812804	25.097520	42.955650	2.524900	

	slope	ctime	reltime
0	96.102965	1.15	65.217391
1	189.047406	0.77	36.363636
2	149.381011	0.12	50.000000
3	243.380868	0.60	21.666667
4	393.760127	0.12	58.333333

This is all very similar to the measurement wheel, except you don't have access to 3D forces. Similar to the measurement wheel data, all of the above can be achieved with the `auto_process` function.

[9]:
```
wl.plots.plot_ergometer_pushes(ergo_data, pushes);
```



Very festive!

2.6.2 Minimal example for working with measurement wheel data

Import the worklab module

[1]:
```
import os
import worklab as wl
```

Import the data with `com.load()` or the device specific load function:

[2]:
```
filename = os.getcwd()
filename = os.path.join(os.path.split(filename)[0], 'example_data', 'Optipush_example.
↪data')
mw_data = wl.com.load(filename)
print("Measurement wheel data is stored in a: ", type(mw_data))
```

```
================================================================================
Initializing loading for C:\Users\rick_\Development\worklab2\example_data\Optipush_
↪example.data ...
File identified as Optipush datafile. Attempting to load ...
Data loaded!
================================================================================

Measurement wheel data is stored in a:  <class 'pandas.core.frame.DataFrame'>
```

The DataFrame contains all the information from the file, but nothing extra:

[3]:
```
mw_data.columns
```

[3]:
```
Index(['time', 'fx', 'fy', 'fz', 'mx', 'my', 'torque', 'angle'], dtype='object')
```

Before getting more infos it would be a good idea to apply some filtering:

[4]:
```
mw_data = wl.kin.filter_mw(mw_data)
```

Now let's get more infos:

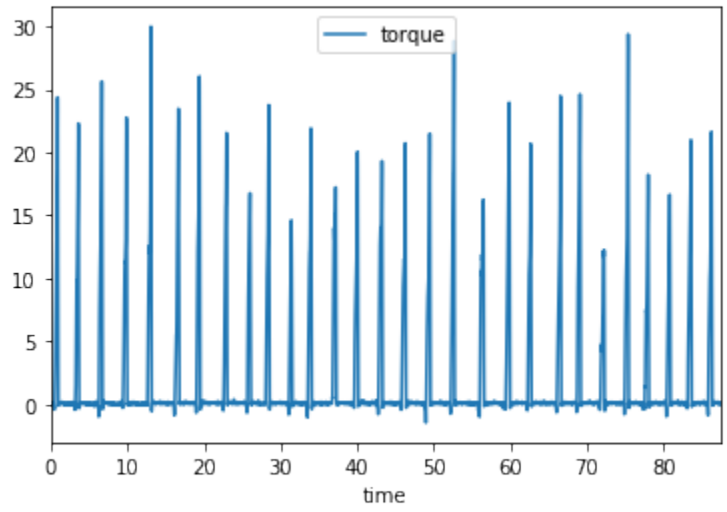
[5]:
```
mw_data = wl.kin.process_mw(mw_data)
mw_data.head()
```

[5]:

	time	fx	fy	fz	mx	my	torque	angle	aspeed	speed	dist	acc	ftot	uforce	force	power	work
0	0.000	2.838031	1.753805	-4.058034	0.004851	0.005135	-0.054472	3.673093	3.352298	1.039212	0.000000	2.535617	5.253370	-0.198079	-0.175715	-0.182605	-0.000913
1	0.005	2.401395	1.666210	-4.385999	-0.002135	0.004741	-0.058072	3.689854	3.393195	1.051890	0.005228	3.594796	5.270667	-0.211171	-0.187329	-0.197050	-0.000985
2	0.010	1.963322	1.620943	-4.642096	-0.006774	0.002821	-0.060339	3.707025	3.468259	1.075160	0.010545	4.264677	5.294445	-0.219414	-0.194642	-0.209271	-0.001046
3	0.015	1.527743	1.645536	-4.771329	-0.007243	-0.001333	-0.060383	3.724537	3.530765	1.094537	0.015970	3.537893	5.273269	-0.219573	-0.194782	-0.213197	-0.001066
4	0.020	1.104789	1.748478	-4.743367	-0.002454	-0.007348	-0.058016	3.742333	3.582384	1.110539	0.021482	2.901078	5.174675	-0.210967	-0.187148	-0.207835	-0.001039

That's more like it!

[6]:
```
mw_data.plot("time", "torque");
```



Let's see if we can find some pushes in these data:

[7]:
```
pushes = wl.kin.push_by_push_mw(mw_data)
print(f"There are {len(pushes)} complete pushes in the data")
pushes.head()
```

```
There are 29 complete pushes in the data
```

[7]:

	stop	start	peak	tstart	tstop	tpeak	cangle	ptime	meanpower	\
0	195	97	176	0.485	0.975	0.88	1.676823	0.490	36.376315	
1	735	636	718	3.180	3.675	3.59	1.630329	0.495	36.170961	
2	1349	1255	1330	6.275	6.745	6.65	1.601879	0.470	45.783597	
3	1994	1881	1972	9.405	9.970	9.86	1.781313	0.565	33.413155	
4	2629	2519	2612	12.595	13.145	13.06	1.897524	0.550	43.382667	

	maxpower	meantorque	maxtorque	meanforce	maxforce	work	\
0	94.426828	10.559981	24.357933	38.399932	88.574302	18.006276	
1	86.937413	10.449778	22.283936	37.999192	81.032495	18.085480	
2	102.833058	12.917944	25.623400	46.974343	93.175999	21.747209	
3	86.508159	10.060917	22.747502	36.585153	82.718191	19.045498	
4	118.642142	12.244082	29.976050	44.523935	109.003817	24.077380	

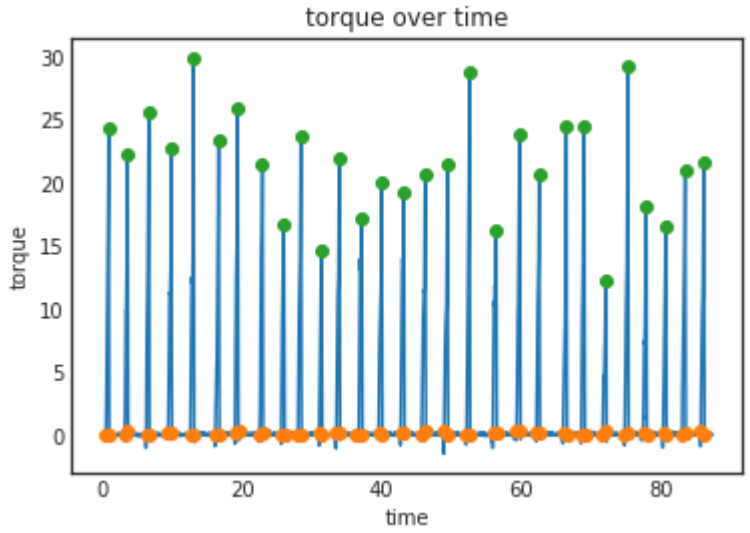
	feff	slope	ctime	reltime
0	55.197071	61.665653	2.695	18.181818
1	52.638096	54.351063	3.095	15.993538
2	63.410359	68.329066	3.130	15.015974
3	56.930325	49.994511	3.190	17.711599
4	61.715078	64.464623	3.590	15.320334

You can achieve the exact same thing using the `auto_process` function. Which chains the above operations into one monolithic function and returns the data and the pushes, as such:

```
mw_data, pushes = wl.kin.auto_process(mw_data, *args, **kwargs)
```

Now make a christmas tree!

[8]:
```
wl.plots.plot_pushes(mw_data, pushes);
```



Magnificent!

2.6.3 Minimal example for working with NGIMU data

Import the worklab module

[1]:
```
import os
import worklab as wl
```

Import the data with `com.load()`:

[2]:
```
filename = os.getcwd()
filename = os.path.join(os.path.split(filename)[0], 'examples', 'example_data', 'imu_
↪example_data')
imu_data = wl.com.load_imu(filename, filenames=["sensors"])
print("NGIMU data is stored in a: ", type(imu_data))
```

```
NGIMU data is stored in a:  <class 'dict'>
```

The structure is as follows: you have a dictionary with all devices, that contains a dictionary with all sensors. Sensordata is stored in Pandas DataFrames:

[3]:
```
print("Imu_data contains: ", imu_data.keys())  # dict
print("Frame contains: ", imu_data["frame"].keys())  # dict
print("sensors contains: ", imu_data["frame"]["sensors"].columns)  # DataFrame
```

```
Imu_data contains:  dict_keys(['right', 'left', 'frame'])
Frame contains:  dict_keys(['sensors'])
sensors contains:  Index(['time', 'gyroscope_x', 'gyroscope_y', 'gyroscope_z',
↪'accelerometer_x',
       'accelerometer_y', 'accelerometer_z', 'magnetometer_x',
       'magnetometer_y', 'magnetometer_z', 'barometer'],
      dtype='object')
```

You can resample the IMUs to a fixed frequency with `imu.resample_imu()` which takes a session data object and a sample frequency:

```
[4]: print("Freq before resampling: ", 1 / imu_data["frame"]["sensors"]["time"].diff().
     ↪mean())
     imu_data = wl.imu.resample_imu(imu_data, sfreq=400)
     print("Freq after resampling: ", 1 / imu_data["frame"]["sensors"]["time"].diff().
     ↪mean())
```

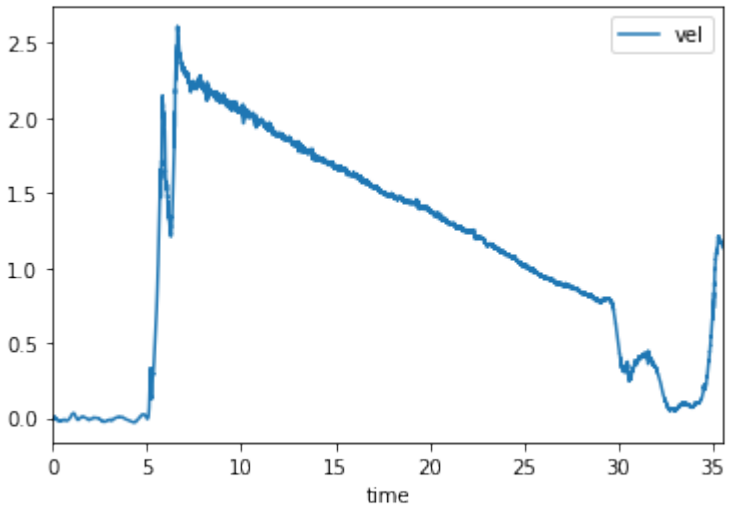
```
Freq before resampling:  397.0700650101465
Freq after resampling:  400.0
```

If you have the IMUs attached to a wheelchair you can use `imu.process_imu()` to get wheelchair performance related variables (the function needs some wheelchair specific information):

```
[5]: imu_data = wl.imu.process_imu(imu_data)
```

If you use the data structure for wheelchair related variables the different sensor keys get dropped and just the relevant DataFrames are kept. You can visualize the data using Pandas built-in plot function or with matplotlib:

```
[6]: imu_data["frame"].plot("time", "vel");
```



There you go! You can consider some filtering now and extracting the variables you need.

2.6.4 Minimal example for working with spirometer data

Import the worklab module:

```
[6]: import os
     import worklab as wl
```

Import the data with `com.load()`:

```
[7]: filename = os.getcwd()
     filename = os.path.join(os.path.split(filename)[0], 'example_data', 'COSMED_example.
     ↪xlsx')
     spiro_data = wl.com.load(filename)
     print("Spirometer data is stored in a: ", type(spiro_data))
```

```
================================================================================
```

(continues on next page)

(continued from previous page)

```
Initializing loading for C:\Users\rick_\Development\worklab2\example_data\COSMED_
↪example.xlsx ...
File identified as COSMED datafile. Attempting to load ...
Data loaded!
================================================================================

Spirometer data is stored in a:  <class 'pandas.core.frame.DataFrame'>
```

or use `wl.com.load_spiro()` which is functionally identical.

You can inspect the data with pandas functions:

[8]:
```
spiro_data.head()
```

[8]:

	time	Rf	HR	power	VO2	VCO2	weights
0	7	11.952191	62	101.842014	298.455484	268.382657	0
1	11	9.933775	62	111.407404	330.369515	281.724639	4
2	15	11.757790	63	128.974406	383.662547	322.481225	4
3	18	13.218771	64	141.802001	423.538135	349.306235	3
4	20	14.430014	64	146.172614	436.703091	359.734206	2

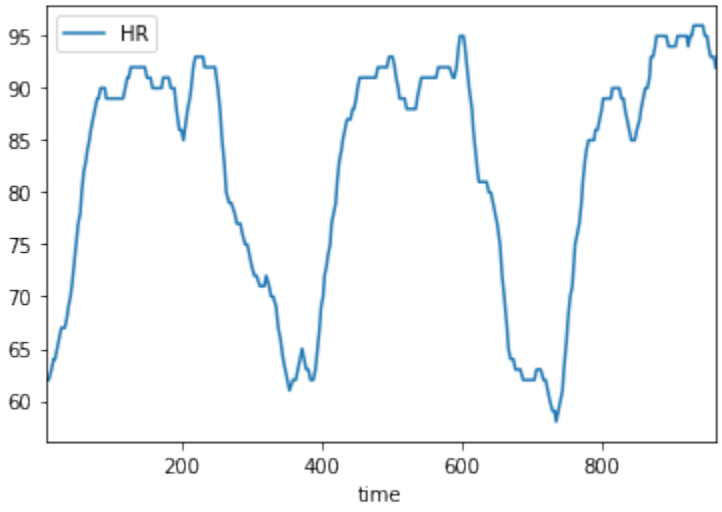
[9]:
```
spiro_data.tail()
```

[9]:

	time	Rf	HR	power	VO2	VCO2	weights
328	951	23.809524	94	338.510226	995.434028	881.662962	2
329	954	23.237800	93	338.135288	994.712143	879.522871	3
330	956	22.883295	93	348.853419	1029.271752	898.141951	2
331	959	23.382697	93	351.913720	1037.692355	907.881222	3
332	961	23.566379	92	353.994886	1043.260882	914.987154	2

Or plot with Pandas built-in functions or matplotlib:

[10]:
```
spiro_data.plot("time", "HR");
```



That's basically all there is to it. Handling spirometer data is not very exciting

2.7 Python modules

2.7.1 Rationale

This is an attempt to make analysis of wheelchair biomechanics data more accessible and transparent. Previously all analyses were performed with commercial software that is not available to everyone, especially to people not associated with a university. Having the analysis in Python makes it accessible and more readable (hopefully) for everyone. By sharing the code I hope to be transparent and to reduce the amount of times this code has to be written by other people.

2.7.2 Examples & audience

People working in our lab that want to work with data from any of our instruments. It can, of course, also be used by other people, provided that you have similar equipment. Most of the time you will only need one or two functions which you can just take from the source code or you can just install the package as it has very little overhead anyways and only uses packages that you probably already have installed. Also have a look at the examples.

2.7.3 Installation

Option 1: the package is now on pip:

```
pip install worklab
```

Option 2: download the package from this page, and run:

```
python setup.py install
```

Option 3: don't install it and just include the scripts in your working directory (why though?).

To verify if everything works simply try to import worklab:

```
python
import worklab as wl
```

That's it.

2.7.4 Breakdown

- com: Provides functions for reading and writing data, use `load` to infer filetype and automatically read it. If you use a different naming scheme you can always call the specific load functions.
- kinetics: Contains all essentials for measurement wheel and ergometer data. You only need the top-level function `auto_process` for most use-cases.
- move: Contains kinematics and movement related functions for NGIMU and some functions for 3D kinematics.
- physio: Contains physiological calculations, which for now is basically nothing as the spirometer does everything for you. Might include EMG and the likes later though.
- plots: Contains some basic plotting functionalities for plots that become repetitive, needs some TLC to become really useful.
- utils: Contains all functions that are useful for more than one application (e.g. filtering and interpolation).

The return of a function is a Pandas DataFrame in 9/10 cases. This means that you can also use all Pandas goodness.

2.7.5 Communication (.com)

Contains functions for reading data from any worklab device. If you abide by regular naming conventions you will only need the load function which will infer the correct function for you. You can also use device-specific load functions if needed.

load

worklab.com.**load**(*filename=''*)

Attempt to load a common data format.

Most important function in the module. Provides high level loading function to load common data formats. If no filename is given will try to load filename using a file dialog. Will try to infer data source from filename. Try to use a specific load function if load cannot infer the datatype.

Parameters **filename** (*str*) – name or path to file of interest

Returns **data** – raw data, format depends on source, but is usually a dict or pandas DataFrame

Return type pd.DataFrame

See also:

load_bike(), *load_esseda()*, *load_hsb()*, *load_n3d()*, *load_opti()*, *load_optitrack()*, *load_imu()*, *load_spiro()*, *load_spline()*, *load_sw()*, load_opti_offset()

load_bike

worklab.com.**load_bike**(*filename*)

Load bicycle ergometer data from LEM datafile.

Loads bicycle ergometer data from LEM to a pandas DataFrame containing time, load, rpm, and heart rate (HR).

Parameters **filename** (*str*) – full file path or file in existing path from LEM Excel sheet (.xls)

Returns **data** – DataFrame with time, load, rpm, and HR data

Return type pd.DataFrame

load_esseda

worklab.com.**load_esseda**(*filename*)

Loads HSB ergometer data from LEM datafile.

Loads ergometer data measured with LEM and returns the data in a dictionary for the left and right module with a DataFrame each that contains time, force (on wheel), and speed.

Parameters **filename** (*str*) – full file path or file in existing path from LEM Excel sheet (.xls)

Returns **data** – dictionary with DataFrame for left and right module

Return type dict

See also:

load_wheelchair() Load wheelchair information from LEM datafile.

load_spline() Load calibration splines from LEM datafile.

load_wheelchair

worklab.com.**load_wheelchair**(*filename*)
Loads wheelchair from LEM datafile.

Loads the wheelchair data from a LEM datafile. Note that LEM only recently added this to their exports. Returns:

Column	Data	Unit
name	chair name	
rimsize	radius of handrim	m
wheelsize	radius of the wheel	m
weight	weight of the chair	kg

Parameters **filename** (*str*) – full file path or file in existing path from LEM Excel sheet (.xls)

Returns **wheelchair** – dictionary with wheelchair information

Return type dict

See also:

load_esseda() Load HSB data from LEM datafile.

load_spline() Load calibration splines from LEM datafile.

load_hsb

worklab.com.**load_hsb**(*filename*)
Loads HSB ergometer data from HSB datafile.

Loads ergometer data measured with the HSBlogger2 and returns the data in a dictionary for the left and right module with a DataFrame each that contains time, force, and speed. HSB files are generally only for troubleshooting and testing that is beyond the scope of LEM.

Parameters **filename** (*str*) – full file path or file in existing path from HSB .csv file

Returns **data** – dictionary with DataFrame for left and right module

Return type dict

load_n3d

worklab.com.**load_n3d**(*filename*, *verbose=True*)
Reads NDI-Optotrak data files

Parameters

- **filename** (*str*) – Optotrak data file (.n3d)
- **verbose** (*bool*) – Print some information about the data from the file. If True (default) it prints the information.

Returns **optodata** – Multidimensional numpy array with marker positions (in m) in sample x xyz x marker dimensions.

Return type ndarray

load_opti

worklab.com.**load_opti**(*filename*, *rotate=True*)

Loads Optipush data from .data file.

Loads Optipush data to a pandas DataFrame, converts angle to radians, and flips torque (Tz). Returns a DataFrame with:

Column	Data	Unit
time	sample time	s
fx	force on local x-axis	N
fy	force on local y-axis	N
fz	force on local z-axis	N
mx	torque around x-axis	Nm
my	torque around y-axis	Nm
torque	torque around z-axis	Nm
angle	unwrapped wheel angle	rad

Note: Optipush uses a local coordinate system, option to rotate Fx and Fy available in >1.6

Parameters

- **filename** (*str*) – filename or path to Optipush .data (.csv) file
- **rotate** (*bool*) – whether or not to rotate from a local rotating axis system to a global non-rotating one, default is True

Returns **opti_df** – Raw Optipush data in a pandas DataFrame

Return type pd.DataFrame

See also:

load_sw() Load measurement wheel data from a SMARTwheel

load_optitrack

worklab.com.**load_optitrack**(*filename*, *include_header=False*)

Loads Optitrack marker data.

Parameters

- **filename** (*str*) – full path to filename or filename in current path
- **include_header** (*bool*) – whether or not to include the header in the output default is False

Returns **marker_data** – Marker data in dictionary, metadata in dictionary

Return type dict

load_imu

worklab.com.**load_imu**(*root_dir*, *filenames=None*)

Imports NGIMU session in nested dictionary with all devices and sensors.

Import NGIMU session in nested dictionary with all devices with all sensors. Translated from xio-Technologies[1].

Parameters

- **root_dir** (*str*) – directory where session is located
- **filenames** (*list, optional*) – list of sensor names or single sensor name that you would like to include, only loads sensor if not specified

Returns **session_data** – returns nested object sensordata[device][sensor][dataframe]

Return type dict

References

load_spiro

worklab.com.**load_spiro**(*filename*)

Loads COSMED spirometer data from Excel file.

Loads spirometer data to a pandas DataFrame, converts time to seconds (not datetime), computes energy expenditure, computes weights from the time difference between samples, if no heart rate data is available it fills the column with np.NaNs. Returns a DataFrame with:

Column	Data | Unit	
time	time at breath	s
HR	heart rate	bpm
EE	energy expenditure	J/s
RER	exchange ratio	VCO2/VO2
VO2	oxygen	l/min
VCO2	carbon dioxide	l/min
VE	ventilation	l/min
VE/VO2	ratio VE/VO2| -	
VE/VCO2	ratio VE/VCO2	•
O2pulse	oxygen pulse (VO2/HR)	•
PetO2	end expiratory O2 tension	mmHg
PetCO2	end expiratory CO2 tension	mmHg
VT	tidal volume	l
weights	sample weight	•

Parameters **filename** (*str*) – full file path or file in existing path from COSMED spirometer

Returns **data** – Spirometer data in pandas DataFrame

[1] https://github.com/xioTechnologies/NGIMU-MATLAB-Import-Logged-Data-Example

Return type pd.DataFrame

load_spline

worklab.com.**load_spline**(*filename*)

Load wheelchair ergometer calibration spline from LEM datafile.

Loads Esseda calibration spline from LEM which includes all forces (at the roller) at the different calibration points (1:10:1 km/h).

Parameters **filename** (*object*) – full file path or file in existing path from LEM excel file

Returns **data** – left and right calibration values

Return type dict

load_sw

worklab.com.**load_sw**(*filename*, *sfreq=200*)

Loads SMARTwheel data from .txt file.

Loads SMARTwheel data to a pandas DataFrame, converts angle to radians and unwraps it. Returns a DataFrame with:

Column	Data	Unit
time	sample time	s
fx	force on global x-axis	N
fy	force on global y-axis	N
fz	force on global z-axis	N
mx	torque around x-axis	Nm
my	torque around y-axis	Nm
torque	torque around z-axis	Nm
angle	unwrapped wheel angle	rad

Note: SMARTwheel uses a global coordinate system

Parameters

- **filename** (*str*) – filename or path to SMARTwheel .data (.csv) file
- **sfreq** (*int*) – sample frequency of SMARTwheel, default is 200Hz

Returns **sw_df** – Raw SMARTwheel data in a pandas DataFrame

Return type pd.DataFrame

See also:

load_opti() Load measurement wheel data from an Optipush wheel.

2.7.6 Kinetics (.kin)

Contains functions for working with measurement wheel (Optipush and SMARTwheel) and ergometer (Esseda) data You will usually only need the top-level function auto_process.

auto_process

worklab.kin.**auto_process**(*data*, *wheelsize=0.31*, *rimsize=0.27*, *sfreq=200*, *co_f=15*, *ord_f=2*, *co_s=6*, *ord_s=2*, *force=True*, *speed=True*, *variable='torque'*, *cutoff=0.0*, *wl=201*, *ord_a=2*, *minpeak=5.0*)

Top level processing function that performs all processing steps for mw/ergo data.

Contains all signal processing steps in fixed order. It is advised to use this function for all (pre-)processing. If needed take a look at a specific function to see how it works.

Parameters

- **data** (*pd.DataFrame, dict*) – raw ergometer or measurement wheel data
- **wheelsize** (*float*) – wheel radius [m]
- **rimsize** (*float*) – handrim radius [m]
- **sfreq** (*int*) – sample frequency [Hz]
- **co_f** (*int*) – cutoff frequency force filter [Hz]
- **ord_f** (*int*) – order force filter [..]
- **co_s** (*int*) – cutoff frequency force filter [Hz]
- **ord_s** (*int*) – order speed filter [..]
- **force** (*bool*) – force filter toggle, default is True
- **speed** (*bool*) – speed filter toggle, default is True
- **variable** (*str*) – variable name used for peak (push) detection
- **cutoff** (*float*) – noise level for peak (push) detection
- **wl** (*float*) – window length angle filter
- **ord_a** (*int*) – order angle filter [..]
- **minpeak** (*float*) – min peak height for peak (push) detection

Returns

- **data** (*pd.DataFrame, dict*)
- **pushes** (*pd.DataFrame, dict*)

See also:

filter_mw(), *process_mw()*, *push_by_push_mw()*, *filter_ergo()*, *process_ergo()*, *push_by_push_ergo()*

filter_mw

worklab.kin.**filter_mw**(*data*, *sfreq=200.0*, *co_f=15.0*, *ord_f=2*, *wl=201*, *ord_a=2*, *force=True*, *speed=True*)

Filters measurement wheel data.

Filters raw measurement wheel data. Should be used before further processing.

Parameters

- **data** (*pd.DataFrame*) – raw measurement wheel data
- **sfreq** (*float*) – sample frequency [Hz]

- **co_f** (*float*) – cutoff frequency force filter [Hz]
- **ord_f** (*int*) – order force filter [..]
- **wl** (*float*) – window length angle filter
- **ord_a** (*int*) – order angle filter [..]
- **force** (*bool*) – force filter toggle, default is True
- **speed** (*bool*) – speed filter toggle, default is True

Returns

- **data** (*pd.DataFrame*)
- *Same data but filtered.*

See also:

utils.lowpass_butter()

filter_ergo

worklab.kin.**filter_ergo**(*data*, *co_f=15.0*, *ord_f=2*, *co_s=6.0*, *ord_s=2*, *force=True*, *speed=True*)

Filters ergometer data.

Filters raw ergometer data. Should be used before further processing.

Parameters

- **data** (*dict*) – raw measurement wheel data
- **co_f** (*float*) – cutoff frequency force filter [Hz]
- **ord_f** (*int*) – order force filter [..]
- **co_s** (*float*) – cutoff frequency speed filter [Hz]
- **ord_s** (*int*) – order speed filter [..]
- **force** (*bool*) – force filter toggle, default is True
- **speed** (*bool*) – speed filter toggle, default is True

Returns **data** – Same data but filtered.

Return type dict

See also:

utils.lowpass_butter()

process_mw

worklab.kin.**process_mw**(*data*, *wheelsize=0.31*, *rimsize=0.275*, *sfreq=200*)

Basic processing for measurement wheel data.

Basic processing for measurement wheel data (e.g. speed to distance). Should be performed after filtering. Added columns:

Column	Data	Unit
aspeed	angular velocity	rad/s
speed	velocity	m/s
dist	cumulative distance	m
acc	acceleration	m/s^2
ftot	total combined force	N
uforce	effective force	N
force	force on wheel	N
power	power	W
work	instantanious work	J

Parameters

- **data** (*pd.DataFrame*) – raw measurement wheel data
- **wheelsize** (*float*) – wheel radius [m]
- **rimsize** (*float*) – handrim radius [m]
- **sfreq** (*int*) – sample frequency [Hz]

Returns data

Return type pd.DataFrame

See also:

com.load_opti(), *com.load_sw()*

process_ergo

worklab.kin.**process_ergo**(*data*, *wheelsize=0.31*, *rimsize=0.275*)

Basic processing for ergometer data.

Basic processing for ergometer data (e.g. speed to distance). Should be performed after filtering. Added columns:

Column	Data	Unit
angle	angle	rad
aspeed	angular velocity	rad/s
acc	acceleration	m/s^2
dist	cumulative distance	m
power	power	W
work	instantanious work	J
uforce	effective force	N
torque	torque around wheel	Nm

Note: the force column contains force on the wheels, uforce (user force) is force on the handrim

Parameters

- **data** (*dict*) – raw ergometer data
- **wheelsize** (*float*) – wheel radius [m]

- **rimsize** (*float*) – handrim radius [m]

Returns **data**

Return type dict

See also:

com.load_esseda()

push_by_push_mw

worklab.kin.**push_by_push_mw**(*data*, *variable='torque'*, *cutoff=0.0*, *minpeak=5.0*, *mindist=5*, *verbose=True*)

Push-by-push analysis for measurement wheel data.

Push detection and push-by-push analysis for measurement wheel data. Returns a pandas DataFrame with:

Column	Data	Unit
start/stop/peak	respective indices	
tstart/tstop/tpeak	respective samples	s
cangle	contact angle	rad
cangle_deg	contact angle	degrees
mean/maxpower	power per push	W
mean/maxtorque	torque per push	Nm
mean/maxforce	force per push	N
mean/maxuforce	(rim) force per push	N
mean/maxfeff	feffective per push	%
mean/maxftot	ftotal per push	N
work	work per push	J
cwork	work per cycle	J
negwork	negative work/cycle	J
slope	slope onset to peak	Nm/s
smoothness	mean/peak force	
ptime	push time	s
ctime	cycle time	s
reltime	relative push/cycle	%

Parameters

- **data** (*pd.DataFrame*) – measurement wheel DataFrame
- **variable** (*str*) – variable name used for peak (push) detection
- **cutoff** (*float*) – noise level for peak (push) detection
- **minpeak** (*float*) – min peak height for peak (push) detection

Returns **pbp** – push-by-push DataFrame

Return type pd.DataFrame

push_by_push_ergo

worklab.kin.**push_by_push_ergo**(*data*, *variable='power'*, *cutoff=0.0*, *minpeak=50.0*, *mindist=5*, *verbose=True*)

Push-by-push analysis for wheelchair ergometer data.

Push detection and push-by-push analysis for ergometer data. Returns a pandas DataFrame with:

Column	Data	Unit
start/stop/peak	respective indices	
tstart/tstop/tpeak	respective samples	s
cangle	contact angle	rad
cangle_deg	contact angle	degrees
mean/maxpower	power per push	W
mean/maxtorque	torque per push	Nm
mean/maxforce	force per push	N
mean/maxuforce	(rim) force per push	N
work	work per push	J
cwork	work per cycle	J
negwork	negative work/cycle	J
slope	slope onset to peak	Nm/s
smoothness	mean/peak force	
ptime	push time	s
ctime	cycle time	s
reltime	relative push/cycle	%

Parameters

- **data** (*dict*) – wheelchair ergometer dictionary
- **variable** (*str*) – variable name used for peak (push) detection, default = power
- **cutoff** (*float*) – noise level for peak (push) detection, default = 0
- **minpeak** (*float*) – min peak height for peak (push) detection, default = 50.0
- **mindist** (*int*) – minimum sample distance between peak candidates, can be used to speed up algorithm

Returns **pbp** – dictionary with left, right and mean push-by-push DataFrame

Return type dict

2.7.7 Kinematics (.move)

Basic functions for movement related data from optical tracking systems. If I have the time I will make a vector3d class. Most functions assume an [n, 3] or [1, 3] array or dataframe.

get_perp_vector

worklab.move.**get_perp_vector**(*vector2d*, *clockwise=True*, *normalized=True*)

Get the vector perpendicular to the input vector. Only works in 2D as 3D has infinite solutions.

Parameters

- **vector2d** (*np.array*) – [n, 3] vector data, only uses x and y
- **clockwise** (*bool*) – clockwise or counterclockwise rotation
- **normalized** (*bool*) – whether or not to normalize the result, default is True

Returns **perp_vector2d** – rotated vector

Return type np.array

get_rotation_matrix

worklab.move.**get_rotation_matrix**(*new_frame*, *local_to_world=True*)

Get the rotation matrix between a new reference frame and the global reference frame or the other way around.

Parameters

- **new_frame** (*np.array*) – [3, 3] array specifying the new reference frame
- **local_to_world** (*bool*) – global to local or local to global

Returns **rotation_matrix** – rotation matrix that can be used to rotate marker data, e.g.: rotation_matrix @ marker

Return type np.array

get_orthonormal_frame

worklab.move.**get_orthonormal_frame**(*point1*, *point2*, *point3*, *mean=False*)

Returns an orthonormal frame from three reference points. For example, a local coordinate system from three marker points.

Parameters

- **point1** (*np.array*) – first marker point, used as origin if mean=False
- **point2** (*np.array*) – second marker point, used as x-axis
- **point3** (*np.array*) – third marker point
- **mean** (*bool*) – whether or not the mean should be used as origin, default is False

Returns

- **origin** (*np.array*) – xyz column vector with coordinates of the origin which is point1 or the mean of all points
- **orthonormal** (*np.array*) – 3x3 array with orthonormal coordinates [x, y, z] of the new axis system

mirror

worklab.move.**mirror**(*vector3d*, *axis='xyz'*)

Simply mirror one or multiple axes.

Parameters

- **vector3d** (*np.array*) – vector to be mirrored, also works on dataframes
- **axis** (*str*) – string with axes to be mirrored

Returns **vector3d** – mirrored vector

Return type np.array

rotate

worklab.move.**rotate**(*vector3d*, *angle*, *deg=False*, *axis='z'*)

Rotate a vector around a single given axis, specify rotation angle in radians or degrees.

Parameters

- **vector3d** (*np.array*) – vector to be rotated, also works on dataframes, assumes [n, xyz] data
- **angle** (*float*) – angle to rotate over
- **deg** (*bool*) – True if angle is specified in degrees, False for radians
- **axis** (*str*) – axis to rotate over, default = "z"

Returns **vector3d** – rotated vector

Return type np.array

scale

worklab.move.**scale**(*vector3d*, *x=1.0*, *y=1.0*, *z=1.0*)

Scale a vector in different directions.

Parameters

- **vector3d** (*np.array*) – array to be scaled, also works on dataframes, assumes [n, xyz] data
- **x** (*float*) – x-axis scaling
- **y** (*float*) – y-axis scaling
- **z** (*floag*) – z-axis scaling

Returns **vector3d** – scaled array

Return type np.array

magnitude

worklab.move.**magnitude**(*vector3d*)

Calculates the vector magnitude using an l2 norm. Works with [1, 3] or [n, 3] vectors.

Parameters **vector3d** (*np.array*) – a [1, 3] or [n, 3] vector

Returns **vector3d** – scalar value or column vector

Return type np.array

normalize

worklab.move.**normalize**(*vector3d*)

Normalizes [n, 3] marker data using an l2 norm. Works with [1, 3] and [n, 3] vectors, both arrays and dataframes.

Parameters **vector3d** (*np.array*) – marker data to be normalized

Returns **vector3d** – normalized marker data

Return type np.array

distance

worklab.move.**distance**(*point1*, *point2*)

Compute Euclidean distance between two points, this is the distance if you were to draw a straight line.

Parameters

- **point1** (*np.array*) – a [1, 3] or [n, 3] array with point coordinates
- **point2** (*np.array*) – a [1, 3] or [n, 3] array with point coordinates

Returns **distance** – distance from point1 to point2 in a [1, 3] or [n, 3] array

Return type np.array

marker_angles

worklab.move.**marker_angles**(*v_1*, *v_2*, *deg=False*)

Calculates n angles between two [n, 3] markers, two [1, 3] markers, or one [n, 3] and one [1, 3] marker.

Parameters

- **v_1** (*np.array*) – [n, 3] array or DataFrame for marker 1
- **v_2** (*np.array*) – [n, 3] array or DataFrame for marker 2
- **deg** (*bool*) – return radians or degrees, default is radians

Returns **x** – returns [n, 1] array with the angle for each sample or scalar value

Return type np.array

is_unit_length

worklab.move.**is_unit_length**(*vector3d*, *atol=1e-08*)

Checks whether an array ([1, 3] or [n, 3]) is equal to unit length given a tolerance

2.7.8 IMU (.imu)

Basic functions for movement related data from IMUs. IMU functions are specifically made for the NGIMUs we use in the worklab.

resample_imu

worklab.imu.**resample_imu**(*sessiondata*, *sfreq=400.0*)

Resample all devices and sensors to new sample frequency.

Resamples all devices and sensors to new sample frequency. Sample intervals are not fixed with NGIMU's so resampling before further analysis is recommended. Translated from xio-Technologies[2].

Parameters

- **sessiondata** (*dict*) – original session data structure to be resampled
- **sfreq** (*float*) – new intended sample frequency

Returns **sessiondata** – resampled session data structure

[2] https://github.com/xioTechnologies/NGIMU-MATLAB-Import-Logged-Data-Example

Return type dict

References

calc_wheelspeed

worklab.imu.**process_imu**(*sessiondata*, *camber=15*, *wsize=0.31*, *wbase=0.6*, *inplace=False*)

Calculate wheelchair kinematic variables based on NGIMU data

Parameters

- **sessiondata** (*dict*) – original sessiondata structure
- **camber** (*float*) – camber angle in degrees
- **wsize** (*float*) – radius of the wheels
- **wbase** (*float*) – width of wheelbase
- **inplace** (*bool*) – performs operation inplace

Returns **sessiondata** – sessiondata structure with processed data

Return type dict

change_imu_orientation

worklab.imu.**change_imu_orientation**(*sessiondata*, *inplace=False*)

Changes IMU orientation from in-wheel to on-wheel

Parameters

- **sessiondata** (*dict*) – original sessiondata structure
- **inplace** (*bool*) – perform operation inplace

Returns **sessiondata** – sessiondata with reoriented gyroscope data

Return type dict

push_imu

worklab.imu.**push_imu**(*acceleration*, *sfreq=400.0*)

Push detection based on velocity signal of IMU on a wheelchair[3].

Parameters

- **acceleration** (*np.array, pd.Series*) – acceleration data structure
- **sfreq** (*float*) – sampling frequency

Returns

Return type push_idx, acc_filt, n_pushes, cycle_time, push_freq

[3] van der Slikke, R., Berger, M., Bregman, D., & Veeger, D. (2016). Push characteristics in wheelchair court sport sprinting. Procedia engineering, 147, 730-734.

References

butterfly

spider

sprint_10m

sprint_20m

vel_zones

worklab.imu.**vel_zones**(*velocity*, *time*)

Calculate wheelchair velocity zones

Parameters

- **velocity** (*np.array, pd.Series*) – velocity data structure
- **time** (*np.array, pd.Series*) – time data structure

Returns **velocity_zones** – velocity zones (m/s), 1-2, 2-3, 3-4, 4-5, 5 and above

Return type dict

2.7.9 Physiology (.physio)

Basics for working with physiological data. We only have a spirometer in the lab at the moment and this involves very little processing. Might expand with EMG related function at some point in the future.

get_spirometer_units

2.7.10 Plotting (.plots)

Most variables can easily be plotted with matplotlib or pandas as most data in this package is contained in dataframes. Some plotting is tedious however and these are functions for those plots.

plot_pushes

worklab.plots.**plot_pushes**(*data*, *pushes*, *var='torque'*, *start=True*, *stop=True*, *peak=True*, *ax=None*)

Plot pushes from measurement wheel or ergometer data.

Parameters

- **data** (*pd.DataFrame*) –
- **pushes** (*pd.DataFrame*) –
- **var** (*str*) – variable to plot, default is torque
- **start** (*bool*) – plot push starts, default is True
- **stop** (*bool*) – plot push stops, default is True
- **peak** (*bool*) – plot push peaks, default is True
- **ax** (*axis object*) – Axis to plot on, you can add your own or it will make a new one.

Returns ax

Return type axis object

plot_pushes_ergo

worklab.plots.**plot_pushes_ergo**(*data*, *pushes*, *title=None*, *var='power'*, *start=True*, *stop=True*, *peak=True*)

Plot left, right and mean side ergometer push data

Parameters

- **data** (*dict*) – processed ergometer data dictionary with dataframes
- **pushes** (*dict*) – processed push_by_push ergometer data dictionary with dataframes
- **title** (*str*) – title of the plot, optional
- **var** (*str*) – variable to plot, default is power
- **start** (*bool*) – plot push starts, default is True
- **stop** (*bool*) – plot push stops, default is True
- **peak** (*bool*) – plot push peaks, default is True

Returns **axes** – an array containing an axis for the left, right and mean side

Return type np.array

plot_power_speed_dist

worklab.plots.**plot_power_speed_dist**(*data*, *title=''*, *ylim_power=None*, *ylim_speed=None*, *ylim_distance=None*)

Plot power, speed and distance versus time for left (solid line) and right (dotted line) seperately

Figure scales automatically, unless you specify it manually with the ylim_* arguments

Parameters

- **data** (*dict*) – processed ergometer data dictionary with dataframes
- **title** (*str*) – a title for the plot
- **ylim_power** (*list [min, max] of float or int, optional*) – list of the minimal and maximal ylim for power in W
- **ylim_speed** (*list [min, max] of floats or int, optional*) – list of the minimal and maximal ylim for speed in km/h
- **ylim_distance** (*list [min, max] of floats or int, optional*) – list of the minimal and maximal ylim for distance in m

Returns

- **fig** (*matplotlib.figure.Figure*)
- **axes** (*tuple*) – the three axes objects

acc_peak_dist_plot

worklab.plots.**acc_peak_dist_plot**(*time*, *acc*, *dist*, *name=''*)
Plot acceleration and distance versus time, with acc_peak

Parameters

- **time** (*np.array, pd.Series*) – time structure
- **acc** (*np.array, pd.Series*) – acceleration structure
- **dist** (*np.array, pd.Series*) – distance structure
- **name** (*str*) – name of a session

Returns ax

Return type axis object

acc_peak_plot

worklab.plots.**acc_peak_plot**(*time*, *acc*, *name=''*)
Plot acceleration versus time, with acc_peak

Parameters

- **time** (*np.array, pd.Series*) – time structure
- **acc** (*np.array, pd.Series*) – acceleration structure
- **name** (*str*) – name of a session

Returns ax

Return type axis object

acc_plot

worklab.plots.**acc_plot**(*time*, *acc*, *name=''*)
Plot acceleration versus time

Parameters

- **time** (*np.array, pd.Series*) – time structure
- **acc** (*np.array, pd.Series*) – acceleration structure
- **name** (*str*) – name of a session

Returns ax

Return type axis object

imu_push_plot

worklab.plots.**imu_push_plot**(*time*, *vel*, *acc_raw*, *name=''*)
Plot push detection with IMUs

Parameters

- **time** (*dict*) – time structure
- **vel** (*dict*) – velocity structure

- **acc_raw** (*dict*) – raw acceleration structure
- **name** (*str*) – name of a session

Returns ax

Return type axis object

rot_vel_plot

worklab.plots.**rot_vel_plot**(*time*, *rot_vel*, *name=''*)

Plot rotational velocity versus time

Parameters

- **time** (*np.array, pd.Series*) – time structure
- **rot_vel** (*np.array, pd.Series*) – rotational velocity structure
- **name** (*str*) – name of a session

Returns ax

Return type axis object

set_axes_equal_3d

worklab.plots.**set_axes_equal_3d**(*axes*)

Set 3D plot axes to equal scale and size

Parameters **axes** (*matplotlib.axes._subplots.Axes3DSubplot*) – axes containing 3D plotted data

vel_peak_dist_plot

worklab.plots.**vel_peak_dist_plot**(*time*, *vel*, *dist*, *name=''*)

Plot velocity and distance against time

Parameters

- **time** (*np.array, pd.Series*) – time structure
- **vel** (*np.array, pd.Series*) – velocity structure
- **dist** (*np.array, pd.Series*) – distance structure
- **name** (*str*) – name of a session

Returns ax

Return type axis object

vel_peak_plot

worklab.plots.**vel_peak_plot**(*time*, *vel*, *name=''*)

Plot velocity versus time, with vel_peak

Parameters

- **time** (*np.array, pd.Series*) – time structure

- **vel** (*np.array, pd.Series*) – velocity structure
- **name** (*str*) – name of a session

Returns **ax**

Return type axis object

vel_plot

worklab.plots.**vel_plot**(*time*, *vel*, *name=''*)

Plot velocity versus time

Parameters

- **time** (*np.array, pd.Series*) – time structure
- **vel** (*np.array, pd.Series*) – velocity structure
- **name** (*str*) – name of a session

Returns **ax**

Return type axis object

2.7.11 Utilities (.utils)

This module contains utility functions used by all modules or functions that have multiple applications such as filtering, finding zero-crossings, finding the nearest value in a signal.

pick_file

worklab.utils.**pick_file**(*initialdir=None*)

Open a window to select a single file

Parameters **initialdir** (*str*) – directory to start from

Returns **filename** – full path to picked file

Return type str

pick_files

worklab.utils.**pick_files**(*initialdir=None*)

Open a window to select multiple files

Parameters **initialdir** (*str*) – directory to start from

Returns **filename** – full path to picked file

Return type list

pick_directory

worklab.utils.**pick_directory**(*initialdir=None*)

Open a window to select a directory

Parameters **initialdir** (*str*) – directory to start from

Returns **directory** – full path to selected directory

Return type str

pick_save_file

worklab.utils.**pick_save_file**(*initialdir=None*)
Open a window to select a savefile

Parameters **initialdir** (*str*) – directory to start from

Returns **directory** – full path to selected savefile

Return type str

calc_weighted_average

make_calibration_spline

worklab.utils.**make_calibration_spline**(*calibration_points*)
Makes a pre-1.0.4 calibration spline for the Esseda wheelchair ergometer.

Parameters **calibration_points** (*dict*) – dict with left: np.array, right: np.array

Returns **spl_line** – dict with left: np.array, right: np.array containing the interpolated splines

Return type dict

make_linear_calibration_spline

worklab.utils.**make_linear_calibration_spline**(*calibration_points*)
Makes a post-1.0.4 calibration spline for the Esseda wheelchair ergometer.

Parameters **calibration_points** (*dict*) – dict with left: np.array, right: np.array

Returns **spl_line** – dict with left: np.array, right: np.array containing the interpolated splines

Return type dict

pd_dt_to_s

worklab.utils.**pd_dt_to_s**(*dt*)
Calculates time in seconds from datetime or string.

Parameters **dt** (*pd.Series*) – datetime instance or a string with H:m:s data

Returns **time** – time in seconds

Return type pd.Series

lowpass_butter

worklab.utils.**lowpass_butter**(*array*, *sfreq=100.0*, *cutoff=20.0*, *order=2*)
Apply a simple zero-phase low-pass Butterworth filter on an array.

Parameters

- **array** (*np.array*) – input array to be filtered

- **sfreq** (*float*) – sample frequency of the signal, default is 100
- **cutoff** (*float*) – cutoff frequency for the filter, default is 20
- **order** (*int*) – order of the filter, default is 2

Returns **array** – filtered array

Return type np.array

interpolate_array

worklab.utils.**interpolate_array**(*x*, *y*, *kind='linear'*, *fill_value='extrapolate'*, *assume=True*)

Simple function to interpolate an array with Scipy's interp1d. Also extrapolates NaNs.

Parameters

- **x** (*np.array*) – time array (without NaNs)
- **y** (*np.array*) – array with potential NaNs
- **kind** (*str*) – kind of filter, default is "linear"
- **fill_value** (*str*) – fill value, default is "extrapolate"
- **assume** (*bool*) – assume that the array is sorted (performance), default is True

Returns **y** – interpolated y-array

Return type np.array

pd_interp

worklab.utils.**pd_interp**(*df*, *interp_column*, *at*)

Resamples (and extrapolates) DataFrame with Scipy's interp1d, this was more performant than the pandas one for some reason.

Parameters

- **df** (*pd.DataFrame*) – target DataFrame
- **interp_column** (*str*) – column to interpolate on, e.g. "time"
- **at** (*np.array*) – column to interpolate to

Returns **interp_df** – interpolated DataFrame

Return type pd.DataFrame

merge_chars

worklab.utils.**merge_chars**(*chars*)

Merges list or tuple of binary characters to single string

Parameters **chars** (*list, tuple*) – list or tuple of binary characters

Returns concatenated characters

Return type str

find_peaks

worklab.utils.**find_peaks**(*data*, *cutoff=1.0*, *minpeak=5.0*, *min_dist=5*)

Finds positive peaks in signal and returns indices of start and stop.

Parameters

- **data** (*pd.Series, np.array*) – any signal that contains peaks above minpeak that dip below cutoff
- **cutoff** (*float*) – where the peak gets cut off at the bottom, basically a hysteresis band
- **minpeak** (*float*) – minimum peak height of wave
- **min_dist** (*int*) – minimum sample distance between peak candidates, can be used to speed up algorithm

Returns **peaks** – dictionary with start, end, and peak **index** of each peak

Return type dict

coast_down_velocity

worklab.utils.**coast_down_velocity**(*t*, *v0*, *c1*, *c2*, *m*)

Solution for the non-linear differential equation M(dv/dt) + c1*v**2 + c2 = 0. Returns the instantaneous velocity decreasing with time (t) for the friction coefficients c1 and c2 for an object with a fixed mass (M)

Parameters

- **t** (*np.array*) –
- **v0** (*float*) –
- **c1** (*float*) –
- **c2** (*float*) –
- **m** (*float*) –

Returns

Return type np.array

nonlinear_fit_coast_down

worklab.utils.**nonlinear_fit_coast_down**(*time*, *vel*, *total_weight*)

Performs a nonlinear fit on coast-down data, returning c1 and c2.

Parameters

- **time** (*np.array*) –
- **vel** (*np.array*) –
- **total_weight** (*float*) –

Returns c1, c2

Return type tuple

mask_from_iterable

worklab.utils.**mask_from_iterable**(*array*, *floor_values*, *ceil_values*)

Combines multiple masks from iterable into one mask (e.g. can be used to select multiple time slices).

Parameters

- **array** (*np.array*) – array to apply mask on
- **floor_values** (*list*) – minimum values in array
- **ceil_values** (*list*) – maximum values in array

Returns **mask**

Return type np.array

calc_inertia

worklab.utils.**calc_inertia**(*weight=0.8*, *radius=0.295*, *length=0.675*, *period=1.0*)

Calculate the inertia of an object based on the trifilar pendulum equation.

Parameters

- **weight** (*float*) – total mass of the object, default is 0.8
- **radius** (*float*) – radius of the object, default is 0.295
- **length** (*float*) – length of the trifilar pendulum
- **period** (*float*) – observed oscillation period

Returns **inertia** – inertia [kgm2]

Return type float

zerocross1d

worklab.utils.**zerocross1d**(*x*, *y*, *indices=False*)

Find the zero crossing points in 1d data.

Find the zero crossing events in a discrete data set. Linear interpolation is used to determine the actual locations of the zero crossing between two data points showing a change in sign. Data point which are zero are counted in as zero crossings if a sign change occurs across them. Note that the first and last data point will not be considered whether or not they are zero.

Parameters

- **x** (*np.array, pd.Series*) – time/sample variable
- **y** (*np.array, pd.Series*) – y variable
- **indices** (*bool*) – return indices or not, default is False

Returns position in time and optionally the index of the sample before the zero-crossing

Return type np.array

camel_to_snake

worklab.utils.**camel_to_snake**(*name: str*)
Turns CamelCased text into snake_cased text.

Parameters **name** (*str*) – StringToConvert

Returns converted_string

Return type str

find_nearest

worklab.utils.**find_nearest**(*array*, *value*, *index=False*)
Find the nearest value in an array or the index thereof.

Parameters

- **array** (*np.array*) – array which has to be searched
- **value** (*float*) – value that you are looking for
- **index** (*bool*) – whether or not you want the index

Returns value or index of nearest value

Return type np.array

binned_stats

worklab.utils.**binned_stats**(*array*, *bins=10*, *pad=True*, *func=<function mean>*, *nan_func=<function nanmean>*)
Apply a compatible Numpy function to every bins samples (e.g. mean or std).

Parameters

- **array** (*np.array*) – array which has to be searched
- **bins** (*int*) – number of samples to be averaged
- **pad** (*bool*) – whether or not to pad the array with NaNs if needed
- **func** – function that is used when no padding is applied
- **nan_func** – function that is used when padding is applied

Returns **means** – array with the mean for every bins samples.

Return type np.array

Timer

class worklab.utils.**Timer**(*name=''*, *text='Elapsed time: {:0.4f} seconds'*, *start=True*)
Simple timer for timing code(blocks).

Parameters

- **name** (*str*) – name of timer, gets saved in Timer.timers optional
- **text** (*str*) – custom text, optional
- **start** (*bool*) – automatically start the timer when it's initialized, default is True

start()
start the timer

stop()
stop the timer, prints and returns the time

lap()
print the time between this lap and the previous one

CHAPTER

THREE

SOURCE

The source is on this GitLab page.

CHAPTER

FOUR

ABOUT

4.1 Authors

- **Rick de Klerk** - *Initial work* - GitLab - UMCG
- **Thomas Rietveld** - GitLab - UMCG
- **Rowie Janssen** - UMCG

4.2 Citing

If you want to refer to this package please use this DOI: 10.5281/zenodo.3268671, or cite:

R.de Klerk. (2019, July 4). Worklab: a wheelchair biomechanics mini-package (Version 1.0.0). Zenodo. http://doi.org/10.5281/zenodo.3268671

4.3 Acknowledgments

- Thanks to R.J.K. Vegter for providing information on the Optipush and SMARTwheel systems.
- Thanks to R.M.A. van der Slikke for providing information on skid correction.
- Thanks to the people at Umaco for answering my questions however dumb they may be.

4.4 References

Specifically for the Python module:

- Vegter, R. J., Lamoth, C. J., De Groot, S., Veeger, D. H., & Van der Woude, L. H. (2013). Variability in bimanual wheelchair propulsion: consistency of two instrumented wheels during handrim wheelchair propulsion on a motor driven treadmill. Journal of neuroengineering and rehabilitation, 10(1), 9.
- Van der Slikke, R. M. A., Berger, M. A. M., Bregman, D. J. J., & Veeger, H. E. J. (2015). Wheel skid correction is a prerequisite to reliably measure wheelchair sports kinematics based on inertial sensors. Procedia Engineering, 112, 207-212.
- van der Slikke, R., Berger, M., Bregman, D., & Veeger, D. (2016). Push characteristics in wheelchair court sport sprinting. Procedia engineering, 147, 730-734.

4.5 Disclaimer

THE GUIDE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE GUIDE OR THE USE OR OTHER DEALINGS IN THE GUIDE.

INDEX

R

S

T

V

Z