This Python package includes data analysis tools for Bluelake HDF5 data.

The source code repository is located on Github where you can also post any questions, comments or issues that you might have.
• Loading scans and kymographs is now much faster
• Improved performance of slicing continuous channels
• Fixed Unknown channel kind error with the v2 file format
• Fixed deprecation warnings with h5py v2.9
• TIFF scan exports now default to using 32bit floats

• Support for photon time tag data has been added (Bluelake version 1.5.0 and higher, HDF5 file format version 2)

• File properties like File.kymos, File.scans now return empty dict objects instead of an error if there are no such items in the file
• Channel slices can be downsampled: `lf_force = hf_force.downsampled_by(factor=20)`

• FDCurves now support subtraction, e.g. `fd = f.fdcures["measured"] - f.fdcures["baseline"]`

• Scans and kymos now have a `.timestamps` property with per-pixel timestamps with the same shape as the image arrays

• Added Matlab compatibility examples to the repository in `examples/matlab`

• `h5py` >= v2.8 is now required
• Initial release
This section contains sample scripts to load .h5df files using different analysis softwares.
We recommend Python/Jupyter Notebook, but h5df files can generally be opened with any data analysis software that’s out there.

5.1 Python

The script for this page with sample data can be downloaded from here.
This page explains how to open .h5df files in Python (Jupyter Notebook) and what you can do with each variable.
You would need to use Pylake to easily access data in the exported files. You only have to run this script once to make sure Pylake is installed:

!pip install lumicks.pylake

If it says “Requirement already satisfied”, then no need to run this each time you run a script.

Load libraries so that you can analyze the exported data:

# Pylake package that LUMICKS provides
import lumicks.pylake as pylake

# standard python toolkit for more elaborate mathematical operations
import numpy as np

# plotting library
import matplotlib.pyplot as plt

5.1.1 Load Files

To load one file, we need to run the following lines of code:

filename = r'20181121-174038 Marker Single tether.h5'
f = pylake.File(filename)
In Python, we can also get a list of all the files in one folder, and we can store this list and load the files one by one:

```python
# load a library to look for files in a folder
import glob

# select the folder, here we search for .h5 files in the folder where you're running the script in
files = glob.glob('*.h5')
print("found {} files in folder".format(len(files)))

for file in files:
    print(file)
    f = pylake.File(file)
    # do the operation on each file here
```

5.1.2 List the Content of a File

We can view the structure of a loaded file like this:

```python
>> filename = 'Data/20181121-174038 Marker Single tether.h5'
>> f = pylake.File(filename)
>> print(f)
```

File root metadata:
- Bluelake version: 1.5.0-fix-timeout.1
- Description:
- Experiment:
- GUID: {45D771E7-E9BA-4255-B4A2-AE00B8F1715E}
- Export time (ns): 1542818438986002600
- File format version: 1

Calibration:
120:
- Force 1x
- Force 1y
- Force 2x
- Force 2y

JSON:
- Data type: object
- Size: 1

Distance:
Distance 1:
- Data type: [('Timestamp', '<i8'), ('Value', '<f8')]
- Size: 172

Force HF:
Force 1x:
- Data type: float64
- Size: 1668751
Force 1y:
- Data type: float64
- Size: 1668751
Force 2x:
- Data type: float64
- Size: 1668751
Force 2y:
- Data type: float64
- Size: 1668751

There are two types of variables in these files in general:
1. Time-traces, e.g. force, distance recordings, confocal recordings, photon count

2. Events, e.g. trap calibration, zeroing the force

Let’s look at the first type.

5.1.3 Access Time-Traces / Channels

- “f” is the file that you previously selected
- the first bracket is the type of variable you’re interested in. These are the leftmost items in the list (e.g. Distance or Force)
- the second bracket is the secondary item in that list. E.g. “Force 1x” is located inside “Force HF”

So a sample code would be like this:

```python
force1x = f["Force HF"]['Force 1x']
```

# or

distance = f['Distance']['Distance 1']

Then you can use different commands on these files:

- `.plot()` to plot these items
- `['0s':'10s']` to slice them
- `.sample_rate` to obtain the sampling rate of this variable
- `.downsampled_by(100)` to downsample the by 100 fold
- `.data` to obtain the raw data from these files
- `.timestamps` to obtain the time data points from this data (note that time is in nanoseconds)

AND you can also combine these command

```python
.plot()` plots the different measured parameters, e.g. force:
```

```python
plt.figure()
force1x.plot()
```
['1s'::'10s'] slices them to look at only a certain section of the data:

```
force1x['1s'::'10s'].plot()
```

sampling_rate gives you the sampling rate in Hz:

```
>>> sampling_rate = force1x.sample_rate
>>> print(sampling_rate)
78125
```
.downsampled_by(100) downsamples the force to a certain sampling frequency:

```python
final_sampling_rate = 100 #Hz
force1x_100Hz = force1x.downsampled_by(int( sampling_rate / final_sampling_rate ))
```

Now we can plot the downsampled force with the original:

```python
force1x.plot()
force1x_100Hz.plot()
```

We can also get the RAW data out and plot them ourselves:

```python
force_data = force1x_100Hz.data
force_time = force1x_100Hz.timestamps

# Please NOTE that the time data are in NANOSECONDS
# We can convert them to seconds in this way, you subtract the first (zeroth) value, then divide by 1e9 (ten to the power of nine)
force_time = (force_time - force_time[0]) * 1e-9

plt.plot(force_time, force_data)
```
5.1.4 Plot Force-Distance Curve

We can make the same plot as before with the obtained data:

```python
force_data = f["Force LF"]['Force 1x'].data
distance_data = f["Distance"]['Distance 1'].data
plt.plot(distance_data, force_data)

"""Now you have to label the axis yourselves""

plt.xlabel("Distance (um)")
plt.ylabel("Force (pN)")
```
5.1.5 Access Events (e.g. calibration)

These are the second type of variables, that are recorded at one point in time and not continuously.

In terms of calibration events, we have 1 in the dataset that is named “123”. We can access it the same way:

```python
params = f['Calibration']['120']['Force 1x']
```

We can also apply certain functions to these files:

- `.h5.attrs.items()` to get a list of what was recorded during this event
- `.attrs.get()` to obtain one of the parameters in the marker

`.h5.attrs.items()` can print the list of calibration parameters:
>>> list(params.h5.attrs.items())

[('Kind', 'Reset offset to zero'),
 ('Offset (pN)', -83.95937128462808),
 ('Response (pN/V)', 817.3620512725377),
 ('Sign', 1.0),
 ('Start time (ns)', 1542807812426015400),
 ('Stop time (ns)', 1542807822426015400),
 ('Bead diameter (um)', 4.4),
 ('Fit range (max.) (Hz)', 23000.0),
 ('Fit range (min.) (Hz)', 10.0),
 ('Fit tolerance', 1e-07),
 ('Max iterations', 10000.0),
 ('Number of samples', 781250.0),
 ('Points per block', 2000.0),
 ('Sample rate (Hz)', 78125.0),
 ('Temperature (C)', 20.0),
 ('Viscosity (Pa*s)', 0.001002),
 ('D (V^2/s)', 0.0013035237229152086),
 ('Rd (um/V)', 8.644328902663627),
 ('Rf (pN/V)', 817.3620512725377),
 ('alpha', 0.6348574679502846),
 ('backing (%)', 100.0),
 ('chi_squared_per_deg', 3.437949600748519),
 ('err_D', 9.46357965045672e-06),
 ('err_alpha', 0.0026906380810170494),
 ('err_f_diode', 143.89781216952397),
 ('err_fc', 4.06541348229771),
 ('f_diode (Hz)', 7140.409945934314),
 ('fc (Hz)', 362.1694334888449),
 ('kappa (pN/nm)', 0.09455471448115298),
 ('ps_fitted', 0.0),
 ('ps_model_fit', 0.0)]

.attrs.get() can grab the parameter of interest, e.g. stiffness:

>>> stiffness = params.h5.attrs.get("kappa (pN/nm)")

>>> print(stiffness)
0.09455471448115298

Chapter 5. Loading H5DF Files
Pylake can be installed on Windows, Linux or Mac, with the following prerequisites:

- Python 3.6 or newer (Python 2.x is not supported)
- The SciPy stack of scientific packages

If you’re already familiar with Python and have the above prerequisites, installing Pylake is just a simple case of using pip, Python’s usual package manager:

```
pip install lumicks.pylake
```

Alternatively, if you’re using Anaconda:

```
conda install lumicks.pylake --c conda-forge
```

If you are new to Python/SciPy, more detailed installation instructions are available below.

### 6.1 Anaconda

The easiest way to install Python and SciPy is with Anaconda, a free scientific Python distribution for Windows, Linux and Mac.

**Windows**

1. Go to the Anaconda website and download the Python 3.6 installer.
2. Run it and accept the default options during the installation.
3. Open Anaconda Prompt from the Start menu. Enter the following command a press enter. This will enable the conda-forge:

   ```
   conda config --add channels conda-forge
   ```

4. Finally, enter the following command to install Pylake:

   ```
   conda install lumicks.pylake
   ```

That’s it, all done. Check out the Tutorial for some example code and Jupyter notebooks to get started.
Linux

1. Go to the Anaconda website and download the Python 3.6 installer.
2. Open a terminal window and run:
   
   ```bash
   bash Anaconda3-x.x.x-Linux-x86_64.sh
   ```
   
   Follow the installation steps. You can accept most of the default values, but make sure that you type `yes` to add Anaconda to PATH:
   
   Do you wish the installer to prepend the Anaconda3 install location to PATH in your `/home/<user_name>/bashrc`? [yes|no] [no] >>> yes
   
   Now, close your terminal window and open a new one for the changes to take effect.
3. Next, enable conda-forge:
   
   ```bash
   conda config --add channels conda-forge
   ```
   
4. Finally, install Pylake with the following command:
   
   ```bash
   conda install lumicks.pylake
   ```

That’s it, all done. Check out the Tutorial for some example code and Jupyter notebooks to get started.

macOS

1. Go to the Anaconda website and download the Python 3.6 installer.
2. Run it and accept the default options during the installation.
3. Open Terminal and run the following command to enable conda-forge:
   
   ```bash
   conda config --add channels conda-forge
   ```
   
4. Finally, install Pylake with the following command:
   
   ```bash
   conda install lumicks.pylake
   ```

That’s it, all done. Check out the Tutorial for some example code and Jupyter notebooks to get started.

6.2 Updating

If you already have Pylake installed and you want to update to the latest version, just run:

```bash
conda update lumicks.pylake
```

6.3 Troubleshooting

If you run into any errors after installation, try updating all conda packages to the latest versions using the following command:

```bash
conda update --all
```
This section will present the essential features of Pylake with example code to get you started quickly. Most of the tutorial pages are also available for download as Jupyter notebooks.

Code snippets are included directly within the tutorial text to illustrate features, thus they omit some common and repetitive code (like import statements) in order to save space and avoid distractions. It is assumed that the following lines precede any other code:

```python
import numpy as np
import matplotlib.pyplot as plt
from lumicks import pylake
```

This uses the common scientific package aliases: `np` and `plt`. These import conventions are used consistently in the tutorial.

### 7.1 Files and channels

Opening a Bluelake HDF5 file is very simple:

```python
from lumicks import pylake
file = pylake.File("example.h5")
```

#### 7.1.1 Contents

To see a textual representation of the contents of a file:

```python
>>> print(file)
File root metadata:
- Bluelake version: 1.3.1
- Experiment: Example
- Description: Collecting example data for Pylake
- GUID: {1A8024D2-C49B-48FF-B183-2FDF0065F26D}
- Export time (ns): 1531162366497820300
- File format version: 1
```

(continues on next page)
Calibration:
1:
    Force 1x
    Force 1y
    Force 2x
    Force 2y
JSON:
    - Data type: object
    - Size: 1
Force HF:
    Force 1x:
    - Data type: float64
    - Size: 706251
    Force 1y:
    - Data type: float64
    - Size: 706251
    Force 2x:
    - Data type: float64
    - Size: 706251
    Force 2y:
    - Data type: float64
    - Size: 706251
Info wave:
    Info wave:
    - Data type: uint8
    - Size: 706251
Marker:
    FRAP 3:
    - Data type: object
    - Size: 1
Photon count:
    Blue:
    - Data type: uint32
    - Size: 706251
    Green:
    - Data type: uint32
    - Size: 706251
    Red:
    - Data type: uint32
    - Size: 706251
Scan:
    reference:
    - Data type: object
    - Size: 1
    bleach:
    - Data type: object
    - Size: 1
    imaging:
    - Data type: object
    - Size: 1

For a listing of more specific timeline items:

    >>> list(file.fdcurves)
    ['baseline', '1', '2']

    >>> list(file.scans)
    ['reference', 'bleach', 'imaging']

    >>> list(file.kymos)

(continues on next page)
The above examples use the `force1x` channel. A full list of available channels can be found on the *File* reference page.

By default, entire channels are returned from a file:

```python
everything = file.force1x
everything.plot()
```

But channels can easily be sliced:

```python
# Get the data between 1 and 1.5 seconds
part = file.force1x['1s':'1.5s']
part.plot()
# Or manually
fix_data = part.data
fix_timestamps = part.timestamps
plt.plot(fix_timestamps, fix_data)
```

# More slicing examples
```python
a = file.force1x[:'-5s']  # everything except the last 5 seconds
b = file.force1x['-1m']   # take the last minute
c = file.force1x['-1m':'-500ms']  # last minute except the last 0.5 seconds
d = file.force1x['1.2s':'-4s']  # between 1.2 seconds and 4 seconds from the end
e = file.force1x['5.7m':'1h 40m']  # 5.7 minutes to an hour and 40 minutes
```

# Subslicing is also possible
```python
a = file.force1x['1s']   # from 1 second to the end of the file
b = a['1s']             # 1 second relative to the start of slice `a`  # -- > `b` starts at 2 seconds relative to the beginning of the file
```

Note that channels are indexed in time units using numbers with suffixes. The possible suffixes are d, h, m, s, ms, us, ns, corresponding to day, hour, minute, second, millisecond, microsecond and nanosecond. This indexing only applies to channels slices. Once you access the raw data, those are regular arrays which use regular array indexing:

---

[7.1.2 Channels](#)

Just like the Bluelake timeline, exported HDF5 files contain multiple channels of data. They can be easily accessed as shown below:

```python
file.force1x.plot()
plt.savefig("force1x.png")
```

The channels have a few convenient methods, like `.plot()` which make it easy to preview the contents, but you can also always access the raw data directly:

```python
fix_data = file.force1x.data
fix_timestamps = file.force1x.timestamps
plt.plot(fix_timestamps, fix_data)
```

---

[7.1. Files and channels](#)
channel_slice = file.force1x['1.5s':'20s']  # timestamps
data_slice = file.force1x.data[20:40]  # indices into the array

7.2 FD curves

The following code loads an HDF5 file and lists all of the FD curves inside of it:

```python
from lumicks import pylake

file = pylake.File("example.h5")
list(file.fdcurves)  # e.g. shows: "['baseline', '1', '2']"
```

To visualize an FD curve, you can use the built-in `.plot_scatter()` function:

```python
# Pick a single FD curve
fd = file.fdcurves["baseline"]
fd.plot_scatter()
```

Here, `.fdcurves` is a standard Python dictionary, so we can do standard `dict` thing with it. For example, we can iterate over all the FD curve in a file and plot them:

```python
for name, fd in file.fdcurves.items():
    fd.plot_scatter()
    plt.savefig(name)
```

By default, the FD channel pair is `downsampled_force2` and `distance1`. This assumes that the force extension was done by moving trap 1, which is the most common. In that situation the force measured by trap 2 is more precise because that trap is static. The channels can be switched with the following code:

```python
alt_fd = fd.with_channels(force='1x', distance='2')
alt_fd.plot_scatter()
```

The raw data can be accessed as well:

```python
# Access the raw data: default force and distance channels
force = fd.f
distance = fd.d
```

```python
# Access the raw data: specific channels
force = fd.downsampled_force1y
distance = fd.distance2
```

```python
# Plot manually: FD curve
plt.scatter(distance.data, force.data)
```

```python
# Plot manually: force timetrace
plt.plot(force.timestamps, force.data)
```

7.3 Confocal images

The following code uses scans as an example. Kymographs work the same way – just substitute `file.scans` with `file.kymos`. To load an HDF5 file and lists all of the scans inside of it, run:

```python
...```
from lumicks import pylake

file = pylake.File("example.h5")
list(file.scans)  # e.g. shows: "['reference', 'bleach', 'imaging']"

Once again, .scans is a regular Python dictionary so we can easily iterate over it:

```python
# Plot all scans in a file
for name, scan in file.scans.items():
    scan.plot_rgb()
    plt.savefig(name)
```

Or just pick a single one:

```python
scan = file.scans["name"]
scan.plot_red()
```

Access the raw image data:

```python
rgb = scan.rgb_image  # matrix with `shape == (h, w, 3)`
blue = scan.blue_image  # single color so `shape == (h, w)`

# Plot manually
plt.imshow(rgb)
```

The images contain pixel data where each pixel is made up a multiple photon count samples collected by the scanner. For an even lower-level look at data, the raw photon count samples can be accessed:

```python
photons = scan.red_photons
plt.plot(photons.timestamps, photons.data)
```

The images can also be exported in the TIFF format:

```python
scan.save_tiff("image.tiff")
```

Multi-frame scans are also supported:

```python
print(scan.num_frames)
print(scan.blue_image.shape)  # (self.num_frames, h, w) -> single color channel
print(scan.rgb_image.shape)  # (self.num_frames, h, w, 3) -> three color channels
```

```python
scan.plot(frame=3)  # plot the third frame -- defaults to the first frame if no argument is given
```
This section contains sample scripts to analyze data from different applications.

You can find application notes for all these segments here.

8.1 Cellular Structure and Transport

You can find our application notes for this segment here.

8.1.1 Kinesin Attached to a Bead Walking on Microtubule

In this assay we had microtubules on the surface. We trapped beads with Kinesin (molecular motor) and had ATP inside the assay. As we lowered the kinesin-coated beads on top of a microtubule, it attached to it and started stepping on them. Kinesins were pulling the bead out of the center of the trap and thus increasing the force on the bead. At one point the kinesins couldn’t keep up with this increased force and the bead construct snapped back to its original position.

After this, the cycles starts again with Kinesins pulling the bead out of the center of the trap.

With the IRM, you can see unlabeled microtubules and the kinesin-coated bead on top of one of them.
To see and characterize this behaviour, we need to have the following plots:

- Plot force versus time
- Plot displacement of the bead from the center of the trap over time

Load all the needed libraries:

```python
import numpy as np
import matplotlib.pyplot as plt
from lumicks import pylake
```

Install Pylake, in case it's not installed:

```
!pip install lumicks.pylake
```

Open the file:

```python
filename = r'20190215-170635 Marker Kinesin stepping open loop.h5';
data = pylake.File(filename)
```

Look at the contents of the file:

```python
>>> print(data)
File root metadata:
- Bluelake version: Unknown
- Description:
- Experiment:
- GUID: {568888F2-223B-40E8-B8FD-87582EF4D20A}
- Export time (ns): 1550246795359570747
- File format version: 2

Calibration:
9:
  Force 1x
  Force 1y
JSON:
  - Data type: object
  - Size: 1
Force HF:
Force 1x:
  - Data type: float64
  - Size: 184297
Force 1y:
  - Data type: float64
  - Size: 184297
Force LF:
Force 1x:
  - Data type: [('Timestamp', '<i8'), ('Value', '<f8')]
  - Size: 93
Force 1y:
  - Data type: [('Timestamp', '<i8'), ('Value', '<f8')]
  - Size: 93
Marker:
single stepping 1:
  - Data type: object
  - Size: 1
```

**Force versus Time**

Get the raw data out:
forcey = data['Force HF']['Force 1y']
forcex = data['Force HF']['Force 1x']

# We need to convert the time from nanoseconds to milliseconds and make sure that
# time=0 at 0 seconds

time = (forcey.timestamps - forcey.timestamps[0]) / 1e9
forcex_data = forcex.data
forcey_data = forcey.data

Plot force in x and y:

plt.figure(figsize=(13, 5))
plt.subplot(2,1,1)
plt.plot(time, forcex_data)
plt.xlabel('Time (s)')
plt.ylabel('Force X (pN)')
plt.subplot(2,1,2)
plt.plot(time, forcey_data)
plt.xlabel('Time (s)')
plt.ylabel('Force Y (pN)')

We can clearly see that the bead was moving in the y direction, so for now we’re just going to work with that. Later I have an example of how to deal with a bead moving at an angle, like at 45 degrees.

But for now, let’s also downsample the force data to 100 Hz and plot the two together.

Downsample the y force data:

downsampled_rate = 100 # Hz

sample_rate = forcey.sample_rate

forcey_downsamp = forcey.downsampled_by(int(sample_rate/downsampled_rate))
forcex_downsamp = forcex.downsampled_by(int(sample_rate/downsampled_rate))
time_downsampled = (forcey_downsamp.timestamps - forcey_downsamp.timestamps[0]) / 1e9

forcey_downsamp_data = forcey_downsamp.data

The two sampling rates are:

>>> print('Original sampling rate is ' + str(sample_rate) + ' Hz')
>>> print('Downsampled rate is ' + str(downsampled_rate) + ' Hz')
Original sampling rate is 30000 Hz
Downsampled rate is 100 Hz

Plot the original force and the downsampled rate:

```python
plt.figure(figsize=(13, 5))
plt.plot(time, forcey_data, label='Original, 30 kHz')
plt.plot(time_downsampled, forcey_downsamp_data, 'r', label='Downsampled, 100 Hz')
plt.xlabel('Time (s)')
plt.ylabel('Force X (pN)')
plt.legend()
plt.grid()
```

Displacement versus Time

We need to convert the force to displacement, which we can do with the following formula:

$$\Delta x = \frac{F}{k}$$

where $F$ is the force and $k$ is the trap stiffness. Force we already have, we need to get stiffness.

Get stiffness from force calibration:

```python
params = data['Calibration']['9']['Force 1y'].h5
ky = params.attrs.get("kappa (pN/nm)")

params = data['Calibration']['9']['Force 1x'].h5
kx = params.attrs.get("kappa (pN/nm)")
```

The stiffness values are:

```python
>>> print(ky) # this is in pN/nm
0.02648593456747345
>>> print(kx) # this is in pN/nm
0.019126295617530483
```

Calculate and plot displacement versus time:

```python
displacement = forcey_data / ky
displacement_downsampled = forcey_downsamp_data / ky
```
plt.figure(figsize=(13, 5))
plt.plot(time, displacement, label='Original, 30 kHz')
plt.plot(time_downsampled, displacement_downsampled, 'r', label='Downsampled, 100 Hz')
plt.xlabel('Time (s)')
plt.ylabel('Displacement (nm)')
plt.legend()
plt.grid()

Distance and Force versus Time on Same Graph

Plot:
fig, ax1 = plt.subplots(figsize=(13, 5))
plt.plot(time, displacement, label='Original, 30 kHz')
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Displacement (nm)')
ax1.set_yticks([-60, -50, -40, -30, -20, -10, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
ax1.grid()

# create another axis
ax2 = ax1.twinx()
# ax2.plot(time_downsampled, fy_downsamp.data+5*ky, 'r-')
ax2.plot(time_downsampled, forcey_downsamp_data, 'r', label='Downsampled, 100 Hz')
ax2.set_ylabel('Force (pN)', color='r')
ax2.tick_params('y', colors='r')

# Here we just make sure that both the displacement and the force axis have the same limits
ylimits = [-60, 100]
ylim2 = []
for i in ylimits:
    ylim2.append(i+ky)

(continues on next page)
ax1.set_ylim(ylimits)
ax2.set_ylim(ylim2)
ax1.set_xlim([0, 5])

32 Chapter 8. Examples by Applications

**X vs Y Position of the Bead**

To get an idea in which direction the microtubule was oriented, which direction the force was applied, we plot the 
(x,y) position of the bead:

```python
plt.plot(forcex_downsamp.data / kx , forcey_downsamp_data / ky,'.')
plt.xlim([-60, 80])
plt.ylim([-60, 80])

plt.ylabel('y-position (nm)')
plt.xlabel('x-position (nm)')
plt.grid()
```
8.1.2 Force Clamp - Kinesin Attached to a Bead Walking on Microtubule

In this assay we had microtubules on the surface. We trapped beads with Kinesin (molecular motor) and had ATP inside the assay. As we lowered the kinesin-coated beads on top of a microtubule, it attached to it and started stepping on the microtubule. Kinesins were pulling the bead out of the center of the trap and thus increasing the force on the bead.

At a certain (set) force, we turn the force clamp on and the nanostage moves towards the motion of the bead. Now the force stays constant, and we get an idea of how the motor steps by looking at the motion of the nanostage.

With the IRM, you can see unlabeled microtubules and the kinesin-coated bead on top of one of them.

Install Pylake, in case it’s not installed:

!pip install lumicks.pylake

Load the relevant libraries:

```python
import numpy as np
import matplotlib.pyplot as plt
from lumicks import pylake
```

Open the file:

```python
filename = r'20190215-142512 Marker force clamp.h5';
data = pylake.File(filename)
```

Look at the contents of the file:

```python
>>> print(data)
```

File root metadata:
- Bluelake version: Unknown
- Description:
- Experiment:
- GUID: {E77F9E67-182F-4BB5-ABB0-14CDB9068600}
- Export time (ns): 1550237112593852547
- File format version: 2

Diagnostics:
Nano X:
- Data type: float64
- Size: 1943676
Nano Y:

(continues on next page)
- Data type: float64
- Size: 1943676

Nano Z:
- Data type: float64
- Size: 1943676

Force HF:
Force 1x:
- Data type: float64
- Size: 1943676
Force 1y:
- Data type: float64
- Size: 1943676

Load the data:

```python
# Force in the y direction (pN)
forcey = data['Force HF']['Force 1y']['6s':'8.5s']
# Nanostage position in the y direction (V)
nanoy = data['Diagnostics']['Nano Y']['6s':'8.5s']
# time traces (seconds)
time = forcey.timestamps/1e9
time = time - time[0]
sample_rate = data['Diagnostics']['Nano Y'].sample_rate
```

Downsample the data:

```python
downsampled_rate = 100 # Hz

# downsample the force, nanostage position and time
forcey_downsamp = forcey.downsampled_by(int(sample_rate/downsampled_rate))
nanoy_downsamp = nanoy.downsampled_by(int(sample_rate/downsampled_rate))
time_downsamp = forcey_downsamp.timestamps/1e9
time_downsamp = time_downsamp - time_downsamp[0]
```

Conversion factor for the nanostage:

```python
# this is determined for each nanostage and it has 3 different conversion factors
# for the 3 directions (x,y,z)
conv_fact = 50000/(1.849-0.04933) #nm/V
```

**Force versus Time**

Plot it:

```python
fig = plt.figure(figsize=(10,5))

forcey.plot()
forcey_downsamp.plot(color='r')
plt.ylabel('Force 1y (pN)')
plt.savefig('../../docs/examples_by_segments/cytoskeletal_kinesin_bead_closed_loop_...
```
Determine force fluctuations:

```python
>>> print('Mean force is: ' + str(np.mean(forcey_downsamp.data)) + ' pN')
>>> print('Variation in the force is: ' + str(np.std(forcey_downsamp.data)) + ' pN')
```

Mean force is: 1.6587699919874592 pN
Variation in the force is: 0.17120278599815678 pN

Here we see that the force stay at 1.7 pN and stays relatively constant

**Nanostage Position versus Time**

Plot it:

```python
fig = plt.figure(figsize=(5,5))
# plot position versus time
ax = plt.subplot(1,1,1)
plt.plot(time_downsamp, nanoy_downsamp.data*conv_fact-2000)
plt.xlim([0, 2])
plt.ylim([60, 160])
# create y-ticks for axis
lims2=[]
for i in range(14):
    lims2.append(i*8+60)
ax.set_yticks(lims2)
# add grid to the graph
ax.yaxis.grid()
# label axis
ax.set_xlabel('Time (s)')
plt.ylabel('Nanostage position (nm)')
```

8.1. Cellular Structure and Transport
8.2 DNA/RNA-Protein Interactions

You can find our application notes for this segment here.

8.2.1 Kymograph and Force

We have two beads trapped and a DNA attached to both of them at either end. We made sure that we have a single tether of DNA by pulling on them before and doing the FD curve.

We then moved into the channel that contains Sytox-Green. It binds to DNA if the DNA is under tension. We can the scan along the DNA and create kymographs using the confocal part of the system.

As we start the kymographs, we can change the force on the DNA and observe the force dependent binding of Sytox to DNA.

This experiments perfectly demonstrates the correlative capabilities of the C-trap.

Install Pylake, in case it’s not installed:

```bash
!pip install lumicks.pylake
```

Load the relevant libraries:

```python
import numpy as np
import matplotlib.pyplot as plt
from lumicks import pylake
```
Open the file:

```python
# Sytox binding, unbinding, with decreased, than increased force
filename = "20181107-152940 Sytox kymograph 7.h5"

# load file
data = pylake.File(filename)
```

Look at the contents of the file:

```python
>>> print(file)
File root metadata:
- Bluelake version: 1.5.0-alpha.35
- Description:
- Experiment:
- GUID: {30C83182-2488-4B27-9EBE-60530CC12D0C}
- Export time (ns): 1541600980363255800
- File format version: 1

Force HF:
Force 1x:
- Data type: float64
- Size: 5059376
Force 1y:
- Data type: float64
- Size: 5059376
Force 2x:
- Data type: float64
- Size: 5059376
Force 2y:
- Data type: float64
- Size: 5059376

Kymograph:
7:
- Data type: object
- Size: 1
Photon count:
Blue:
- Data type: uint32
- Size: 5059376
Green:
- Data type: uint32
- Size: 5059376
Red:
- Data type: uint32
- Size: 5059376
```

Make Kymographs

List all the kymographs in the file:

```python
>>> for kymo_name in file.kymos:
    >>> print(kymo_name)
7
```

Load the kymograph in the file:

```python
# you can either do this and then you have to change which kymo you load for every
--file:
kymo_data = file.kymos["7"] # as this file contains kymograph #7
```

(continues on next page)
# ALTERNATIVELY you can either do this and then you don't have to worry about which file you open

```python
kymos = list(file.kymos)
kymo_data = file.kymos[kymos[0]]
```

Plot the red channel:

```python
fig = plt.figure(figsize=(15,10))

# here you select the kymo
kymo = file.kymos["7"]
kymo.plot_green()
plt.tight_layout()
```

Note that we can also scale the colorbar of the image.

This is not so straightforward, here we just show a very simple way of doing it.

Get the raw data out of the kymographs:

```python
blue_date = kymo.blue_image
green_date = kymo.green_image
red_date = kymo.red_image
```

```python
# this gives you the timestamps if you want to produce the kymos yourself
timestamps = kymo.timestamps
```

Get a sense of the pixel values in the kymos

```python
>>> max_px = np.max(green_date)
35
>>> min_px = np.min(green_date)
0
```
Scale the colorbar and make the kymograph look better:

```python
fig = plt.figure(figsize=(15,10))
file.kymos["7"].plot_green(vmax=10)
plt.tight_layout()
```

---

**Force versus Time**

Load the data:

```python
# Force in the x direction (pN)
forcex = data['Force HF']['Force 1x']

# time traces (seconds)
time = forcex.timestamps/1e9
    time = time - time[0]

sample_rate = forcex.sample_rate
```

Downsample the data:

```python
downsampled_rate = 100 # Hz

# downsample the force, nanostage position and time
forcex_downsamp = forcex.downsampled_by(int(sample_rate/downsampled_rate))
time_downsamp = forcex_downsamp.timestamps/1e9
    time_downsamp = time_downsamp - time_downsamp[0]
```

Plot Force:

```python
fig = plt.figure(figsize=(10,5))
```
 forcex.plot(label="Original")
 forcex_downsamp.plot(color='r', label="Downsampled")
 plt.ylabel('Force 1x (pN)')
 plt.xlim([0, max(time)])
 plt.legend()
 plt.tight_layout()

 Correlated Force and Confocal

 Plot the final figure:

 fig = plt.figure(figsize=(15,10))

 plt.subplot(2,1,1)
 file.kymos["7"].plot_green(vmax=10)

 plt.subplot(2,1,2)
 forcex.plot(label="Original")
 forcex_downsamp.plot(color='r', label="Downsampled")
 plt.xlim([0, max(time)])
 plt.ylabel('Force 1x (pN)')
 plt.tight_layout()
We see when we decreased the force on the DNA the Sytox unbound. As soon as we increase the tension back, we see Sytox binding again. At around 52 seconds, the DNA tether broke, which is why the force went back to its original position.
CHAPTER 9

API Reference

This detailed reference lists all the classes and functions contained in the package. If you are just looking to get started, read the Tutorial first.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>File</code></td>
<td>A convenient HDF5 file wrapper for reading data exported from Bluelake</td>
</tr>
<tr>
<td><code>channel.Slice</code></td>
<td>A lazily evaluated slice of a timeline/HDF5 channel</td>
</tr>
<tr>
<td><code>fdcurve.FDCurve</code></td>
<td>An FD curve exported from Bluelake</td>
</tr>
<tr>
<td><code>kymo.Kymo</code></td>
<td>A Kymograph exported from Bluelake</td>
</tr>
<tr>
<td><code>scan.Scan</code></td>
<td>A confocal scan exported from Bluelake</td>
</tr>
<tr>
<td><code>point_scan.PointScan</code></td>
<td>A point scan exported from Bluelake</td>
</tr>
</tbody>
</table>

### 9.1 lumicks.pylake.File

**class File (filename)**

A convenient HDF5 file wrapper for reading data exported from Bluelake

**Parameters**

- **filename** [str] The HDF5 file to open in read-only mode

**Examples**

```python
from lumicks import pylake

file = pylake.File("example.h5")
file.force1x.plot()
file.kymos["name"].plot()
```

**Methods**

- **__init__ (filename)**
  Initialize self. See help(type(self)) for accurate signature.
__init__ (filename)
  Initialize self.

from_h5py (h5py_file)
  Directly load an existing h5py.File

**Attributes**

SUPPORTED_FILE_FORMAT_VERSIONS
blue_photon_count
blue_photon_time_tags
bluelake_version
  The version of Bluelake which exported this file
description
  The description of the measurement as entered by the user in Bluelake
distance1
distance2
downsampling_force1
downsampling_force1x
downsampling_force1y
downsampling_force2
downsampling_force2x
downsampling_force2y
downsampling_force3
downsampling_force3x
downsampling_force3y
downsampling_force4
downsampling_force4x
downsampling_force4y
experiment
  The name of the experiment as entered by the user in Bluelake
export_time
  The moment this file was exported
fdcurves
force1x
force1y
force2x
force2y
force3x
force3y
force4x
force4y
format_version
  The version of the Bluelake-specific HDF5 file structure
green_photon_count
green_photon_time_tags
guid
  An ID which uniquely identifies each exported file
kymos
point_scans
red_photon_count
red_photon_time_tags
scans

__getitem__ (item)
  Return a subgroup or a bluelake timeline channel

classmethod from_h5py (h5py_file)
  Directly load an existing h5py.File
bluelake_version
  The version of Bluelake which exported this file
description
The description of the measurement as entered by the user in Bluelake

experiment
The name of the experiment as entered by the user in Bluelake

export_time
The moment this file was exported

format_version
The version of the Bluelake-specific HDF5 file structure

guid
An ID which uniquely identifies each exported file

9.2 lumicks.pylake.channel.Slice

class Slice (data_source, labels=None)
A lazily evaluated slice of a timeline/HDF5 channel

Users will only ever get these as a result of slicing a timeline/HDF5 channel or slicing another slice (via this class' __getitem__), i.e. the __init__ method will never be invoked by users.

Parameters

data_source [Any] A slice data source. Can be Continuous, TimeSeries, 'TimeTags', or any other source which conforms to the same interface.

labels [Dict[str, str]] Plot labels: “x”, “y”, “title”.

__init__ (data_source, labels=None)
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__ (data_source[, labels]) Initialize self.
downsampled_by (factor[, reduce]) Return a copy of this slice which is downsampled by factor
plot(**kwargs) A simple line plot to visualize the data over time

Attributes

data The primary values of this channel slice

sample_rate The data frequency for continuous data sources or None if it’s variable

timestamps Absolute timestamps (since epoch) which correspond to the channel data

__getitem__ (item)
All indexing is in timestamp units (ns)
downsampled_by (factor, reduce=<function mean>) Return a copy of this slice which is downsampled by factor

Parameters

factor [int] The size and sample rate of the data will be divided by this factor.

reduce [callable] The numpy function which is going to reduce multiple samples into one. The default is np.mean, but np.sum could also be appropriate for some cases,
```
e.g. photon counts.

plot(**kwargs)
A simple line plot to visualize the data over time

Parameters
**kwargs Forwarded to matplotlib.pyplot.plot().

data
The primary values of this channel slice

sample_rate
The data frequency for continuous data sources or None if it’s variable

timestamps
Absolute timestamps (since epoch) which correspond to the channel data

9.3 lumicks.pylake.fdcurve.FDCurve

class FDCurve(file, start, stop, name, force='2', distance='1')
An FD curve exported from Bluelake

By default, the primary force and distance channels are downsampled_force2 and distance1. Alternatives can be selected using FDCurve.with_channels(). Note that it does not modify the FD curve in place but returns a copy.

Attributes

- start, stop [int] The time range (ns) of this FD curve within the file.
- name [str] The name of this FD curve as it appeared in the timeline.

__init__(file, start, stop, name, force='2', distance='1')
Initialize self. See help(type(self)) for accurate signature.

Methods

__init__(file, start, stop, name[, force, ...]) Initialize self.
from_dset(h5py_dset, file, **kwargs)
plot_scatter(**kwargs) Plot the FD curve points
with_channels(**kwargs) Return a copy of this FD curve with difference primary force and distance channels

Attributes

- d The primary distance channel associated with this FD curve
- distance1
- distance2
- downsampled_force1
- downsampled_force1x
- downsampled_force1y
- downsampled_force2
- downsampled_force2x
- downsampled_force2y
- downsampled_force3

Continued on next page
```
Table 7 – continued from previous page

<table>
<thead>
<tr>
<th>downsampled_force3x</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>downsampled_force3y</td>
<td></td>
</tr>
<tr>
<td>downsampled_force4</td>
<td></td>
</tr>
<tr>
<td>downsampled_force4x</td>
<td></td>
</tr>
<tr>
<td>downsampled_force4y</td>
<td></td>
</tr>
</tbody>
</table>

\( f \)  
The primary force channel associated with this FD curve

\( \text{plot\_scatter(**kwargs)} \)  
Plot the FD curve points

**Parameters**

**kwargs  
Forwarded to \text{~matplotlib.pyplot.scatter}.

\( \text{with\_channels(force, distance)} \)  
Return a copy of this FD curve with difference primary force and distance channels

\( d \)  
The primary distance channel associated with this FD curve

\( f \)  
The primary force channel associated with this FD curve

9.4 lumicks.pylake.kymo.Kymo

\texttt{class Kymo(h5py\_dset, file)}  
A Kymograph exported from Bluelake

**Parameters**

\( \text{h5py\_dset} \) [h5py.Dataset] The original HDF5 dataset containing kymo information

\( \text{file} \) [lumicks.pylake.File] The parent file. Used to loop up channel data

\texttt{__init__(h5py\_dset, file)}  
Initialize self. See help(type(self)) for accurate signature.

**Methods**

\texttt{__init__(h5py\_dset, file)}  
Initialize self.

\texttt{plot\_blue(**kwargs)}  
Plot an image of the blue photon channel

\texttt{plot\_green(**kwargs)}  
Plot an image of the green photon channel

\texttt{plot\_red(**kwargs)}  
Plot an image of the red photon channel

\texttt{plot\_rgb(**kwargs)}  
Plot a full rgb kymograph image

\texttt{save\_tiff(filename[, dtype, clip])}  
Save the RGB photon counts to a TIFF image

**Attributes**

\texttt{blue\_image}  

\texttt{blue\_photon\_count}  

\texttt{green\_image}  

\texttt{green\_photon\_count}  

\texttt{has\_fluorescence}  

\texttt{has\_force}  

\texttt{infowave}  

Continued on next page
### Table 9 – continued from previous page

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pixels_per_line</td>
<td></td>
</tr>
<tr>
<td>red_image</td>
<td></td>
</tr>
<tr>
<td>red_photon_count</td>
<td></td>
</tr>
<tr>
<td>rgb_image</td>
<td></td>
</tr>
<tr>
<td>timestamps</td>
<td>Timestamps for image pixels, not for samples</td>
</tr>
</tbody>
</table>

#### plot_blue(**kwargs)**

Plot an image of the blue photon channel

**Parameters**

**kwargs Forwarded to ~matplotlib.pyplot.imshow.

#### plot_green(**kwargs)**

Plot an image of the green photon channel

**Parameters**

**kwargs Forwarded to ~matplotlib.pyplot.imshow.

#### plot_red(**kwargs)**

Plot an image of the red photon channel

**Parameters**

**kwargs Forwarded to ~matplotlib.pyplot.imshow.

#### plot_rgb(**kwargs)**

Plot a full rgb kymograph image

**Parameters**

**kwargs Forwarded to ~matplotlib.pyplot.imshow.

#### save_tiff(filename, dtype=<class 'numpy.float32'>, clip=False)

Save the RGB photon counts to a TIFF image

**Parameters**

filename [str] The name of the TIFF file where the image will be saved.

dtype [np.dtype] The data type of a single color channel in the resulting image.

clip [bool] If enabled, the photon count data will be clipped to fit into the desired dtype. This option is disabled by default: an error will be raise if the data does not fit.

#### timestamps

Timestamps for image pixels, not for samples

The returned array has the same shape as the *_image arrays.

## 9.5 lumicks.pylake.scan.Scan

**class Scan(h5py_dset, file)**

A confocal scan exported from Bluelake

**Parameters**

h5py_dset [h5py.Dataset] The original HDF5 dataset containing kymo information

file [lumicks.pylake.File] The parent file. Used to loop up channel data

#### __init__(h5py_dset, file)

Initialize self. See help(type(self)) for accurate signature.
Methods

```python
__init__(h5py_dset, file) Initialize self.
plot_blue(**kwargs) Plot an image of the blue photon channel
plot_green(**kwargs) Plot an image of the green photon channel
plot_red(**kwargs) Plot an image of the red photon channel
plot_rgb(**kwargs) Plot a full rgb kymograph image
save_tiff(filename, dtype=<class 'numpy.float32'>, clip=False) Save the RGB photon counts to a TIFF image
```

Attributes

```python
blue_image
blue_photon_count
green_image
green_photon_count
has_fluorescence
has_force
infowave
lines_per_frame
num_frames
pixels_per_line
red_image
red_photon_count
rgb_image
timestamps
```

plot_blue(**kwargs)
Plot an image of the blue photon channel

**Parameters**

**kwargs Forwarded to ~matplotlib.pyplot.imshow.

plot_green(**kwargs)
Plot an image of the green photon channel

**Parameters**

**kwargs Forwarded to ~matplotlib.pyplot.imshow.

plot_red(**kwargs)
Plot an image of the red photon channel

**Parameters**

**kwargs Forwarded to ~matplotlib.pyplot.imshow.

plot_rgb(**kwargs)
Plot a full rgb kymograph image

**Parameters**

**kwargs Forwarded to ~matplotlib.pyplot.imshow.

save_tiff(filename, dtype=<class 'numpy.float32'>, clip=False) Save the RGB photon counts to a TIFF image

**Parameters**

filename [str] The name of the TIFF file where the image will be saved.

dtype [np.dtype] The data type of a single color channel in the resulting image.
**clip** [bool] If enabled, the photon count data will be clipped to fit into the desired dtype. This option is disabled by default: an error will be raise if the data does not fit.

**timestamps**

Timestamps for image pixels, not for samples

The returned array has the same shape as the *_image arrays.

### 9.6 lumicks.pylake.point_scan.PointScan

#### class PointScan(h5py_dset, file)

A point scan exported from Bluelake

**Parameters**

- **h5py_dset** [h5py.Dataset] The original HDF5 dataset containing the point scan
- **file** [lumicks.pylake.File] The parent file. Used to look up channel data.

**__init__**(h5py_dset, file)

Initialize self. See help(type(self)) for accurate signature.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong></td>
<td>Initialize self.</td>
</tr>
<tr>
<td>plot_blue(<strong>kwargs</strong>)</td>
<td>Plot the red photon channel</td>
</tr>
<tr>
<td>plot_green(<strong>kwargs</strong>)</td>
<td>Plot the red photon channel</td>
</tr>
<tr>
<td>plot_red(<strong>kwargs</strong>)</td>
<td>Plot the red photon channel</td>
</tr>
<tr>
<td>plot_rgb(<strong>kwargs</strong>)</td>
<td>Plot all color channels</td>
</tr>
</tbody>
</table>

**Attributes**

- **blue_photon_count**
- **green_photon_count**
- **has_fluorescence**
- **has_force**
- **red_photon_count**

**plot_blue(**kwargs**)**

Plot the red photon channel

**Parameters**

- **kwargs** Forwarded to `matplotlib.pyplot.plot`.

**plot_green(**kwargs**)**

Plot the red photon channel

**Parameters**

- **kwargs** Forwarded to `matplotlib.pyplot.plot`.

**plot_red(**kwargs**)**

Plot the red photon channel

**Parameters**

- **kwargs** Forwarded to `matplotlib.pyplot.plot`.
plot_rgb(**kwargs)

Plot all color channels

**Parameters**

**kwargs Forwarded to ~matplotlib.pyplot.plot.

- genindex