jpkfile Documentation

Release 1.3

Ilyas Kuhlemann

Apr 29, 2019

Contents

jpkfile is a module for reading of data archives recorded with JPK Instruments. It is a very early stage of the project, so expect to encounter some problems. This project is licensed under MIT License, and everyone is invited to contribute. If you encounter any problems, feel free to notify me via email, open a new issue on github or fix it and send a pull request!

To see how to use this module, I recommend to take a look at the files in the *examples* folder. The source code is documented on *this page*. I also refer to *a page* several times where I describe the structure of JPK archives.

CHAPTER 1

Install

You have the following three options to install jpkfile.

1.1 via pip

The simplest way, if you have internet access on the target device, is to install via pip. In a command line type:

```
$ pip install --user jpkfile
```

1.2 via source

You can also clone/download the source files from jpkfile's project page. Navigate to your copy of the project folder in a command line and type:

```
$ python setup.py install --user
```

1.3 add module to your `PYTHONPATH`

Currently `jpkfile` has only one relevant python file `jpkfile.py`, which makes "installing" straight-forward: you simply need to add the folder in which the .py file lies to your python path. One way to do so is to add a *.pth* file containing the path to the folder as text to your user's *site-packages* folder: * On my Ubuntu 18.04 machine, the site-packages folder is at */home/<username>/.local/lib/python3.7/site-packages* * If yours isn't there, you can figure it out by starting python in the command line and typing

```
>>> import site
>>> site.USER_SITE
```

- Open a new file in a text editor, enter the path to the folder containing the jpkfile module.

- Save the file into the site-packages folder.

That’s it. You should now be able to import jpkfile anywhere on your system.

CHAPTER 2

First Steps

You can have a look at the `read_data_from_jpk_archive.py` file, or, if you have jupyter/ipython notebook installed, at the `read_data_from_jpk_archive.ipynb` file in the examples folder. Those examples show you first steps how to use the jpkfile module.

CHAPTER 3

Additional Resources

3.1 Source code documentation

The whole jpkfile project/package consists of only one relevant file for normal use: jpkfile.py. It is composed of three classes: `JPKFile`, `JPKSegment`, and `JPKMap`. The user's interface is mostly covered by the `JPKFile` class, which is used for regular force and tweezer recordings. For force maps, you need to use the `JPKMap` class. `JPKSegment` is usually created and populated internally from within the other two classes.

3.1.1 JPKFile

class `jpkfile.`**`JPKFile`**(*fname*)

Class to unzip a JPK archive and handle access to its headers and data.

Parameters **`fname`** (`str`) – Filename of archive to read data from.

`get_array`(*channels=[], decode=True*)

Returns channel data from all segments in a numpy array; in addition, reads physical units as specified by header files.

Parameters

- **`channels`** – List of channels (channel names, i.e. strings) of which to return data.
- **`decode`** (`bool`) – Determines whether data is to be decoded, i.e. transformed according to transformation parameters defined in header files.

Returns Tuple with two items: (1) Numpy array with labeled columns, one column per requested channel; (2) dictionary assigning units to channels.

`get_info`(*issue='general'*)

Request a string on a certain issue. Currently only one *issue* keyword is possible: 'segments'. This returns a summary/overview on segment properties.

Parameters **`issue`** (`str`) – Keyword/issue on which to request information.

Returns String on requested issue.

`read_files`(*list_of_filenames*)
Crawls through list of files in archive and processes them automatically by name and extension. It populates `parameters` and `segments` with content. For different file types present in JPK archives, have a look at the *structure of JPK archives*.

`has_shared_header = None`
Will be set to `True` if archive has a shared header, `False` otherwise

`num_segments = None`
Number of segments in archive.

`parameters = None`
Dictionary containing parameters read from the top level `header.properties` file.

`segments = None`
Dictionary containing one JPKSegment instance per segment.

`shared_parameters = None`
`None` if no shared header is present, dictionary containing parameters otherwise.

3.1.2 JPKSegment

class `jpkfile.`**`JPKSegment`**(*parent_has_shared_header=False*, *shared_properties=None*)
Class to hold data and parameters of a single segment in a JPK archive. It is usually created internally when handling a JPK archive with the JPKFile class.

Parameters

- **`parent_has_shared_header`** (*`bool`*) – True, if JPKFile finds shared header in JPK archive, False otherwise.
- **`shared_properties`** – If parent_has_shared_header is True, this parameter needs to hold the dictionary containing the header's contents. Otherwise it is None.

`get_array`(*channels=[]*, *decode=True*)
Constructs a numpy array containing data of given channels. If *decode* is True (default), data is converted following conversions defined in segment's header (or shared header).

Parameters

- **`channels`** – List of channels (channel names, i.e. strings) of which to return data.
- **`decode`** (*`bool`*) – Determines whether data is to be decoded, i.e. transformed according to transformation parameters defined in header files.

Returns Tuple with two items: (1) Numpy array with labeled columns, one column per requested channel; (2) dictionary assigning units to channels.

`get_decoded_data`(*channel*, *conversions_to_be_applied='auto'*)
Get decoded data of one channel. 'decoded' here means the raw, digital data gets converted (to physical data) following certain conversion steps. These steps should be defined in the JPK archive's header files. This routine tries to read those conversion steps from those header files by default (*conversions_to_be_applied='auto'*). Alternatively, you can pass a list of conversion keywords as *conversions_to_be_applied* manually (see documentation on *JPK archive structures* for an overview of what I think how conversion rules are stored in the header files . . .).

Parameters

- **`channel`** (*`str`*) – Name of channel to convert data of.
- **`conversions_to_be_applied`** – Specifying what conversions to apply, see description above.

Returns Tuple with 2 items; (1) Single-column numpy array containing converted data; (2) Unit as read for last conversion step from header file.

get_info(*issue='general'*)

Request information (string) on some issue. This is basically just a more user-friendly assignment of parameters of interest to single keywords. For the *issue* parameter the following strings are valid:

- 'general' (default)
- 'channels'
- 'num-points'
- 'duration'
- 'type'

Parameters **issue** (*str*) – Keyword specifying what kind of information is asked for.

Returns String or list of strings containing requested information.

get_time(*offset=0*)

Returns time-stamps, increased by possible offset.

data = None

Dictionary assigning numpy arrays containing data and definitions on how to convert raw data to physical data to all channels present in this segment.

parameters = None

Dictionary holding parameters read from segment header.

3.1.3 JPKMap

class `jpkfile.`**JPKMap**(*fname*)

Loads a JPK map file (ending on '.jpk-force-map') to the buffer. This map consists of multiple 'pixels', each of which is a single force recording at one position. This makes maps a collection of recordings. To get the single pixels to work in a similar way as the JPKFile objects and to make the whole module more modular (!?) , I created a class based on *JPKFile* called *_JPKFileForJPKMap*, which requires a *_VirtualZipFile* object instead of the path to a zip file. This makes the whole concept a bit harder to follow, but it makes the pixels behave as JPKFile objects. This class is not yet outfitted with many helpful functions to retrieve or analyse data of the map. The problem is, I don't really know what is useful and what isn't, since I never worked with maps. If you want a feature added, feel free to send me a message or open an issue on github or implement it yourself and send a pull request.

Parameters **fname** (*str*) – Path to force map (zip archive, usually ending on '.jpk-force-map').

get_single_pixel(*index*)

Returns JPKFile instance of a single pixel.

Parameters **index** – Integer for flat indices or tuple/list of two integers for grid coordinates pointing to desired pixel.

read_files()

Crawls through list of files in archive and processes them automatically by name and extension. It populates *parameters* and *flat_indices* with content. For different file types present in JPK archives, have a look at the *structure of JPK archives*.

flat_indices = None

Dictionary containing JPKFile instances, one per pixel, indexed with flat indices.

parameters = None
Dictionary holding parameters stored in top level header file.

3.1.4 Helper functions, attributes and classes

`jpkfile.`**`extract_data`**(*content*, *dtype*, *num_points*)
Converts data from contents of .dat files in the JPKArchive to python-understandable formats. This function requires the binary *content*, the *dtype* of the binary content as read from the appropiate header file, and the number of points as specified in the header file to double check the conversion.

Parameters

- **content** (*str*) – Binary content of a .dat file.
- **dtype** (*str*) – Data type as read from heade file.
- **num_points** (*int*) – Expected number of points encoded in binary content.

Returns Numpy array containing digital (non-physical, unconverted) data.

`jpkfile.`**`determine_conversions_automatically`**(*conversion_set_dictionary*)
Takes all parameters on how to convert some channel's data read from a header file to determine the chain of conversion steps automatically.

Parameters **conversion_set_dictionary** (*dict*) – Dictionary of 'conversion-set' parameters as parsed from header file with function *parse_header_file*.

Returns List of conversion keywords.

class `jpkfile.jpkfile.`**`_VirtualZipFile`**(*parent_zip*, *excerpt_list_of_filenames*, *prefix*)
THIS CLASS SHOULD NEVER BE USED DIRECTLY. IT IS USED INDIRECTLY VIA *JPKMap*. *Virtual* ZipFile class, to make the functionality of the real ZipFile class available for a subfolder of real zip archives. I implemented this to be able to use the familiar JPKFile class for each pixel of a force map. This way, only few things have to be adjusted in JPKFileForJPKMap, which inherits JPKFile, to make every pixel available as a JPKFile instance.

Parameters

- **parent_zip** (*ZipFile*) – ZipFile instance holding the subfolder that is to be governed by this _VirtualZipFile.
- **excerpt_list_of_filenames** – List of filenames (strings) containing only files of the subfolder; path has to be relative as if looking from within the subfolder.
- **prefix** (*str*) – Path prefix, i.e., path to the subfoler. This is used to construct the complete path to each file for the real ZipFile instance.

list_of_filenames = None
List of filenames (strings) containing only files of the subfolder; Paths have to be relative as if looking from within the subfolder. For example, if your complete zip archive (see files in *parent_zip*) has a folder called 'A', and it contains a file named 'bla.txt', its path will be 'A/bla.txt' in the real ZipFile. In a *_VirtualZipFile* supposed to govern the contents of folder 'A', the path has to be only 'bla.txt', however.

parent_zip = None
(Pointer to) Real ZipFile instance, containing this _VirtualZipFile's folder.

prefix = None
Prefix to the folder governed by this _VirtualZipFile. Referring to the example above, this needs to be 'A/'. It will be used to pass the complete path to the *parent_zip*.

class jpkfile.jpkfile.**_JPKFileForJPKMap**(*virtual_zip*, *has_shared_header*, *shared_parameters*)

THIS CLASS SHOULD NEVER BE USED DIRECTLY. IT IS USED INDIRECTLY VIA *JPKMap*. This class is derived from JPKFile; its purpose is to make the JPKFile class, which is designed to provide a user interface to data of a single measurement or recording, available for use with JPKMap (force maps) which is a collection of multiple recordings.

Parameters

- **virtual_zip** – (Pointer to) _VirtualZipFile for the subdirectory.
- **has_shared_header** – *True* if parent *JPKMap* has a shared header, *False* otherwise.
- **shared_parameters** – *None* if no shared header present, a dictionary containing parameters read from shared header otherwise.

3.2 Structure of JPK Archives.

JPK files are zipped archives, containing data and header files, arranged into a top level folder containing one header filer (named `header.properties`) and several subfolders. To my knowledge, there are 3 different types of archives:

- **simple force scan**, with default file extension `.jpk-force`. This type contains a single force spectroscopy recording.
- **nt force scan**, with default file extension `.jpk-nt-force`. This contains data recorded with JPK Instruments' NanoTracker system (optical tweezers).
- **force scan map**, default extension `.jpk-force-map`. This type contains a 'force map', a collection of simple force scans.

The following is a closer look at the archive structure. Common features are described first, followed by a list of features unique to the different archive types.

3.2.1 Common features of JPK archives

- There is a header file at the top level named `header.properties`. This file appears to contain parameters all subfolders have in common.
- There can be a folder named `shared-data`. It contains another header file with the same name as the top level header, by full path it should be `shared-data/header.properties`. It contains parameters that sometimes is referred to by header files in subfolders. It uses a system of indices to discriminate between blocks of parameters. This indices are then referred to by the header files in subfolders. **WARNING** I do not know if I covered all cases of links from local headers to the shared header. This might lead to errors in conversion from raw data to physical units. I list the cases that I implemented at the bottom under 'Links to shared header'.

3.2.2 Subfolders in simple force scans and nt force scans

These two archive types seem to have the same structure.

- There is a folder named `segments`. It contains subfolders with integers as names. Each subfolder stands for one segment of the recording.
  - Each subfolder contains a header file named `segment-header.properties`. The parameters of this header file are only valid for one single segment. Among the parameters is one specifying the type of

segment: whether it's an extension, a retraction or a pause. It also contains information on what channels are present in this segment, and how to convert the raw data to physical data.

  – Each subfolder contains another folder named `channels`.

    * The `channels` folder contains pure data files, no headers or meta-data of any form. The files' names are identical to the channels' names, followed by the extension `.dat`. The data type is specified in the `segment-header.properties` file above. The data type has to be known to read the binary data in the files and convert it into a python data type. All data appears to be stored in C structs and can be converted using the python module `struct`'s function `unpack`.

3.2.3 Additional files in force scan maps

This archive type has some additional files at the top level:

- `thumbnail.jpg`, `data-image.force`. Both seem to be preview images, maybe with different file types for different systems, not sure.
- A folder named `index`. It contains subfolders with integers as names. Each of these folders holds the data of one pixel of the force map and parameters valid only for one pixel.
  – The contents of each of those folders are identical to that of a simple force scan. They contain another `header.properties` and a `segments` folder.

3.2.4 Links to shared header

General rule

The header files contain many parameters (one per line), they have the form of keywords separated by dots, followed by an equal sign and a value. Here is an example from a header file:

```
channel.ySignal1.data.num-points=16384
```

A parameter line can also contain a link to parameters in the shared header. They take the following form:

```
<keyword_1>.<keyword_2>...<keyword_n>.<link_label>-info.*=i
```

where `i` is an integer.This links to parameters starting with `<link_label>-info.i` in the shared header, like so:

```
<link_label>-info.i.<keyword_a1>.<keyword_a2> = value_a
<link_label>-info.i.<keyword_b1> = value_b
```

and so on. These would then be considered in the local header as

```
<keyword_1>.<keyword_2>...<keyword_n>.<keyword_a1>.<keyword_a2> = value_a
<keyword_1>.<keyword_2>...<keyword_n>.<keyword_b1> = value_b
```

In other words, you leave out the label with the '-info' suffix and the integer and append the keywords and values that follow in the shared header.

Links from segments to shared header

The following are lists of where links to a shared header occured in the JPK archives that I used to test the `jpkfile` module.

1. For data conversion

For conversion of raw data to physical data, the headers contain a key called *conversion-set*, under which there can be multiple factors that need to be applied for the conversion. Instead of storing the *conversion-set* locally, it can be stored in the shared header. This seems to be the case if there is a keyword chain looking like this in the local header:

```
channel.<channel_label>.lcd-info.*=i
```

This integer links to a number in the shared header, occuring in the following style:

```
lcd-info.i
```

There can be multiple parameters starting with this keyword chain.

2. General segment information

Some of the segment's general parameters can be stored in a shared header. There needs to be the following chain of keywords:

```
force-segment-header.force-segment-header-info.*=i
```

This links to parameters in the shared header starting with:

```
force-segment-header-info.i
```

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

CHAPTER 5

License

Python Module Index

Index