Analyzing XL-Calibur Data with Python

Fabian Kislat

1. Preparation

In order to use the XL-Calibur flight software for Python analysis on the cluster a few setup steps are necessary. Log in to a Wustl cluster machine. Currently, the software works on

`adrastea, callisto, cassini, europa, ganymede, jupiter`

To set up the required environment, log in to one of the cluster machines and run the following commands (I'm using my account as an example):

```
fabian@gluon ~ % ssh fkislat@jupiter.wustl.edu -Y
[fkislat@jupiter ~]$ scl enable devtoolset-11 rh-python38 bash
[fkislat@jupiter ~]$ python3 -m venv ~/xlcalibur
[fkislat@jupiter ~]$ source ~/xlcalibur/bin/activate
(xlcalibur) [fkislat@jupiter ~]$ pip3 install --upgrade pip
(xlcalibur) [fkislat@jupiter ~]$ pip3 install ipython numpy scipy matplotlib
```

Next, verify that your installation works by running `ipython` and importing some XL-Calibur software modules (note that I'm using a \ to indicate line continuation):

```
(xlcalibur) [fkislat@jupiter ~]$ \
    /data/xlcal/software/flightsoftware/env-shell.sh ipython
Entering env-shell environment
Python 3.8.11 (default, Jul 23 2021, 14:55:16)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.0.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from xlcalibur import dataclasses, housekeeping, dataaccess
In [2]: from xlcalibur.core import logging
In [3]: logging.log_info("Success!")
<ipython-input-3-8ef2cbb946b0>:1: INFO: Success!
```

In particular, check that this is using Python 3.8.11 and that the import statements work. If this doesn't work: check for typos. Did you remember to `source ~/xlcalibur/bin/activate`? Did you prefix `ipython` with the XL-Calibur `env-shell.sh`?

The steps up to this point only need to be done once. However, each time after logging in, you have to set up the environment: enable RedHat Software Collections `devtoolset-11` and `rh-python28`, then activate your Python environment:

```
fabian@gluon ~ % ssh fkislat@jupiter.wustl.edu -Y
[fkislat@jupiter ~]$ scl enable devtoolset-11 rh-python38 bash
[fkislat@jupiter ~]$ source ~/xlcalibur/bin/activate
(xlcalibur) [fkislat@jupiter ~]$
```

The changed prompt indicates that you're in the Python virtual environment.

2. Reading Data

2.1. Starting a Python Session

Start ipython. Prefix with XL-Calibur `env-shell.sh`:

```
(xlcalibur) [fkislat@jupiter ~]$ \
    /data/xlcal/software/flightsoftware/env-shell.sh ipython
Entering env-shell environment
Python 3.8.11 (default, Jul 23 2021, 14:55:16)
Type 'copyright', 'credits' or 'license' for more information
IPython 8.0.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Note: I'm using a backslash (\) to indicate line continuation. Omit it and just continue your line.

Import XL-Calibur python modules:

```
In [1]: from xlcalibur import dataclasses, housekeeping, dataaccess
In [2]: from xlcalibur.xcom import Packets
In [3]: from xlcalibur.systems import Systems
```

The modules are:

- `dataclasses`: Python wrappers for classes representing XL-Calibur data files.
- `housekeeping`: Special data classes representing housekeeping data.
- `dataaccess`: Simplified access to event data.
- `xcom`: Low-level data format definitions. The `Packets` structure has constants identifying types of data packets.
- `xlcalibur.systems.Systems`: Constants identifying XL-Calibur flight systems (X_SYSTEM_GSE_CLIENT, X_SYSTEM_GSE, X_SYSTEM_WASP_GSE, X_SYSTEM_GONDOLA, X_SYSTEM_TRUSS, X_SYSTEM_POLARIMETER). Handy for selecting data based on their origin.

Of course, all of this can also be scripted in a Python script run from the command line.

2.2. Getting Data Packets

2.2.1. Basic File Information

XL-Calibur data are stored in binary files.

Data are transmitted in packets with the following structure:

- Header containing meta-data about the contents, 10 bytes in total: Packet start word `0xF00D`; packet size; origin; type of data; payload version; sequence number.
- Packet payload. The encoded binary data. If necessary padded to an even number of bytes.
- Packet footer with 16-bit CRC checksum followed by `0xD0D0` and `0xCAFE`.

The `xcom.Packets` structure has constants identifying types of data packets:

- X_PKT_COMMAND = 0: A command (usually sent to the payload).
- X_PKT_EXP_DATA = 1: Polarimeter event data.
- X_PKT_PING = 2: Ping from the payload (to tell us the system is alive).
- X_PKT_HOUSEKEEPING = 3: Housekeeping data.
- X_PKT_DAQ_STATUS = 4: Status update from the data acquisition. Events collected, rate, …
- X_PKT_ALIGNMENT_DATA = 5: Alignment fit results.
- X_PKT_RUNHEADER = 6: Run header containing run number and detector configuration in use.
- X_PKT_MESSAGE = 7: A text message.
- X_PKT_QUERY_REPLY = 8: A container packet that can have different formats, sent in response to a command. For example, asking for the current HV settings.
- X_PKT_FILE_TRANSFER = 9: Part of a file. Not written to data files. Instead, filewriter picks up the pieces and puts files together.
- X_PKT_SHIELD_THRESH_SCAN_DATA = 0xA: Shield threshold scan data.
- X_PKT_POINTING_DATA = 0xB: Pointing data from the WASP.
- X_PKT_TEST_DATA = 0xC: Connection testing data with meaningless contents.
- X_PKT_GSE_REPLY = 0xE: Response from the CSBF GSE to a command.

Python bindings for these classes are implemented in the ground-base package. The C++ classes are documented at

https://xcalibur.physics.wustl.edu/flightsoftware/docs/trunk/

The best way to find out what the Python interface is, is to look at the Python bindings source code

https://gitlab.com/xl-calibur-flight/ground-base/-/tree/master/dataclasses-pybindings

2.2.2. Reading All Data From A File

Create a file object and extract all packets:

```
In [4]: infile = \
    dataclasses.XDataFile("/data/xlcal/datafromxcbe/Run010248.dat")
In [5]: all_packets = infile.Scan()
In [6]: print(len(all_packets))
51808
```

Data are stored in `/data/xlcal` and data received by the automatic processing is in `/data/xlcal/datafromxcbe`.

The result of `infile.Scan()` is a Python list of `XDataPacket` objects that hold the binary data and provide access to the packet header fields.

To make the data accessible to a Python program they need to be decoded ("deserialized" as in "converted from a serial stream of bytes into a Python [or C++] structure"). Python's list comprehension allows a concise and easily readable syntax:

```
In [8]: all_data = [p.Deserialize() for p in all_packets]

In [9]: all_data[:5]
Out[9]:
[<xlcalibur.dataclasses.XRunHeader at 0x7f244e73c6d0>,
<xlcalibur.dataclasses.XMessage at 0x7f244e73c900>,
<xlcalibur.dataclasses.XMessage at 0x7f244e73c510>,
<xlcalibur.dataclasses.XDataRate at 0x7f244e73c430>,
<xlcalibur.dataclasses.XDataRate at 0x7f244e73c7b0>]
```

As you can see, the result is a list of objects representing the different types of packets, starting with the run header.

You can combine the two operations:

```
In [10]: all_data = [p.Deserialize() for p in infile.Scan()]
```

2.2.3. Reading Specific Data

Often you don't want all data, but only a certain kind of data.

You can pass a constant from `xcom.Packets` to `XDataFile.Scan()` to select a type of packet you want. For example:

```
In [12]: all_hskp = [
   ...:     p.Deserialize()
   ...:     for p in infile.Scan(Packets.X_PKT_HOUSEKEEPING)
   ...: ]
```

It's also possible to select a list of packet types:

```
In [13]: all_dacq = [
   ...:     p.Deserialize()
   ...:     for p in infile.Scan([
   ...:         Packets.X_PKT_RUNHEADER, Packets.X_PKT_DAQ_STATUS
   ...:     ])
   ...: ]
```

2.2.4. Reading Large Amounts Of Data

Packets can use much more memory than they use on disk. It can be better to read a file one packet at a time:

```
In [17]: count = 0
In [18]: infile.Rewind()
In [19]: p = infile.NextPacket(Packets.X_PKT_EXP_DATA)
In [20]: while p is not None:
   ...:     p = p.Deserialize()
   ...:     count += p.count
   ...:     p = infile.NextPacket(Packets.X_PKT_EXP_DATA)
In [21]: count
Out[21]: 30688
```

Note: It's necessary to manually reset the read pointer to the beginning of the file using `Rewind()` if you use `NextPacket()`.

Like `Scan()`, `NextPacket()` accepts an optional packet type or list of packet types.

After deserialization, `p` is an object of type `XEventPacket`. Its `count` property equals the number of stored events.

2.3. Exercise 1

Print the severity and text of the 100th text message sent by the polarimeter system stored in the file `/data/xlcal/datafromxcbe/Run010248.dat`.

Hints:

- Messages are represented by `XMessage` objects. Severity and message text are represented by the object properties `severity` and `message`, respectively.
- The origin of a data packet is stored in its `origin` property.
- The polarimeter system is identified by `Systems.X_SYSTEM_POLARIMETER`.

2.3.1. Solution

```
In [1]: from xlcalibur import dataclasses
In [2]: from xlcalibur.xcom import Packets
In [3]: from xlcalibur.systems import Systems
In [4]: infile = \
    dataclasses.XDataFile("/data/xlcal/datafromxcbe/Run010248.dat")
In [5]: all_polarimeter_messages = [
   ...:     p.Deserialize()
   ...:     for p in infile.Scan(Packets.X_PKT_MESSAGE)
   ...:     if p.origin == Systems.X_SYSTEM_POLARIMETER
   ...: ]
In [6]: msg = all_polarimeter_messages[99]
In [7]: print(msg.severity, msg.message)
2 DACQ: Encountered 10 timeouts. Resetting ASICs.
```

3. Housekeeping Data

3.1. Basic Structure

Housekeeping data are stored in packets of type `XHousekeepingData` identified by `Packets.X_PKT_HOUSEKEEPING`.

Each housekeeping packet can contain data from multiple subsystems of a flight system.

Within a housekeeping packet, data are organized in frames. The structure of frames depends on the data stored and each type of frame is represented by a Python data class.

```
In [29]: all_hskp = [
   ...:     p.Deserialize() for p in infile.Scan(Packets.X_PKT_HOUSEKEEPING)
   ...: ]
In [30]: list(all_hskp[0].frames)
Out[30]:
[<xlcalibur.housekeeping.ClockFrame at 0x7f244dbe1660>,
<xlcalibur.housekeeping.PolarimeterHousekeepingFrame at 0x7f244e660dd0>,
<xlcalibur.housekeeping.CPUMonitoringFrame at 0x7f244e576890>,
<xlcalibur.housekeeping.DiskMonitoringFrame at 0x7f244e576120>,
<xlcalibur.housekeeping.TransmitterFrame at 0x7f244e576350>]
```

Note: in the last line I converted to a Python list only to print the list of frames. `XHousekeepingData.frames` is indexable and iterable:

```
In [35]: all_hskp[0].frames[2]
Out[35]: <xlcalibur.housekeeping.CPUMonitoringFrame at 0x7f244dd17040>
In [36]: for frame in all_hskp[0].frames:
   ...:     print(frame)
<xlcalibur.housekeeping.ClockFrame object at 0x7f244e5766d0>
<xlcalibur.housekeeping.PolarimeterHousekeepingFrame object at 0x7f244e64ca50>
<xlcalibur.housekeeping.CPUMonitoringFrame object at 0x7f244e5766d0>
<xlcalibur.housekeeping.DiskMonitoringFrame object at 0x7f244e64ca50>
<xlcalibur.housekeeping.TransmitterFrame object at 0x7f244e5766d0>
```

Here’s a quick way to get all housekeeping frames:

```
In [44]: all_frames = [
   ...:     fr
   ...:     for p in infile.Scan(Packets.X_PKT_HOUSEKEEPING)
   ...:     for fr in p.Deserialize().frames
   ...: ]
```

3.2. Housekeeping Frames

Most housekeeping frame classes are created from an XML description (to reduce C++ boilerplate code):

https://gitlab.com/xl-calibur-flight/xlcalibur-core/-/tree/master/housekeeping/frames/xml

The resulting classes are subclasses of `KVFrame` documented here:

https://xcalibur.physics.wustl.edu/flightsoftware/docs/trunk/classKVFrame.html

There are a few exceptions. All frames are subclasses of `XHousekeepingDataFrame`:

https://xcalibur.physics.wustl.edu/flightsoftware/docs/trunk/classXHousekeepingDataFrame.html

Each frame has an associated acquisition time:

```
In [39]: t = all_hskp[0].frames[0].time
In [40]: t
Out[40]: <xlcalibur.core.GPSTime at 0x7f244f3b3ba0>
In [41]: print(t.week, t.sec_in_week)
2196 164246.67975
In [42]: t.to_mjd()
Out[42]: 59617.90100323784
```

Time is stored in `GPSTime` objects. This **does not** mean that the time was acquired from the GPS.

`GPSTime` objects store the GPS week (counted since January 6, 1980) and second within that week.

The `GPSTime.to_mjd()` function provides a convenient way to convert to MJD.

3.3. Key-Value Access

Frame classes derived from `KVFrame` provide data access via a key-value dictionary.

This dictionary provides basic introspection allowing access to the stored data without knowledge of the structure.

The `iteritems()` member function allows iteration over the key-value pairs. For example a clock frame:

```
In [50]: for kv in frame.iteritems():
   ...:     print(kv.first, kv.second)
deltaT 0.0009958399459719658
gpssecond 164235.0208
gpsweek 2196
```

Here's a quick way to get all frames of this type:

```
In [45]: kvframes = [
   ...:     fr
   ...:     for p in infile.Scan(Packets.X_PKT_HOUSEKEEPING)
   ...:     for fr in p.Deserialize().frames
   ...:     if isinstance(fr, housekeeping.KVFrame)
   ...: ]
```

3.4. Exercise 2

Create a text file named `test-hskp.txt` that contains a table of housekeeping data (KVFrame items) in the file `/data/xlcal/datafromxcbe/Run010248.dat` with the following columns:

- Key;
- Origin of the data;
- Time;
- Value.

Hints:

- You can open an output file using `outf = open(“test-hskp.txt”, ‘w’)` or use a `with open(“test-hskp.txt”, ‘w’) as outf:` block.
- To write a line to the file use `outf.write(text)` where `text` is the text to be written including a newline character (\n) at the end.
- Use `isinstance(frame, housekeeping.KVFrame)` to determine if a frame is a `KVFrame` object.

3.4.1. Solution

```
In [52]: with open("test-hskp.txt", "w") as outf:
    ...:     for p in all_hskp:
    ...:         for fr in p.frames:
    ...:             if isinstance(fr, housekeeping.KVFrame):
    ...:                 for kv in fr.iteritems():
    ...:                     outf.write(
    ...:                         "%s %d %f %s\n"
    ...:                         % (
    ...:                             kv.first,
    ...:                             p.origin,
    ...:                             fr.time.to_mjd(),
    ...:                             str(kv.second)
    ...:                         )
    ...:                     )
```

You can use Ctrl+Z to temporarily suspend your ipython session. Then check the file using `less test-hskp.txt`.

4. Alignment Data

4.1. Basic Structure

Alignment data are stored in data packets identified by `Packets.X_PKT_ALIGNMENT_DATA` represented by the class `XAlignmentData`.

Alignment data contains the fit results x_{center}, y_{center}, $\alpha_{rotation}$, scale with errors:

```
In [56]: all_alignment = [
   ...:     p.Deserialize()
   ...:     for p in infile.Scan(Packets.X_PKT_ALIGNMENT_DATA)
   ...: ]

In [57]: a = all_alignment[0]

In [58]: print(
   ...:     a.center_x,
   ...:     a.center_x_error,
   ...:     a.center_y,
   ...:     a.center_y_error,
   ...:     a.angle,
   ...:     a.angle_error,
   ...:     a.scale,
   ...:     a.scale_error,
   ...:     a.chi2,
   ...:     a.fit_valid
   ...: )
724.4375 0.16015625 483.4375 0.16015625 4.540037631988525 0.0032043154351413251
1.378173828125 0.0041961669921875 1.9335829019546509 True
```

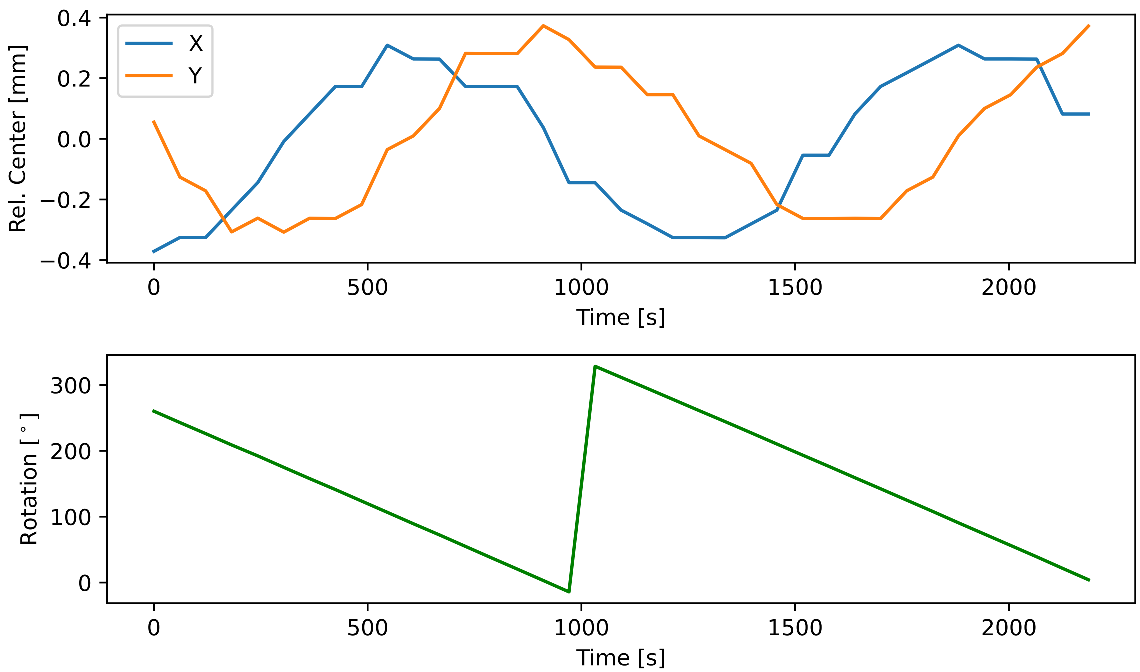
Note: x_{center} and y_{center} are stored in pixels. Scale is in pixels/mm. Rotation angle is in rad.

The property `fit_valid` is a boolean that indicates whether the fit was successful.

Additional information about the individual LEDs and the clusters identified in the image is available via the `clusters` property and the `cluster_assignments` array.

4.2. Exercise 3

Use matplotlib to plot the deviation of the center x and y position in millimeters from its average as a function of time (in seconds since the start of the data taking) on one panel, and the rotation angle (in degrees) on a separate panel below. Your result should look like this:



Hints:

- The numpy library comes in handy:

```
In [67]: import numpy as np
```

- For plotting, I recommend:

```
In [68]: from matplotlib import pyplot as plt
In [69]: %matplotlib
```

- Use `fig, axes = plt.subplots(2, 1)` to create a plot with two panels. Then `axes` is a list of two `Axis` objects that can be used to plot, one for each panel.
- If ax is one of the two axis objects, you can plot using `ax.plot(xdata, ydata)` where `xdata` and `ydata` are arrays representing x and y coordinates of data points. Add the optional argument `label="X"` to label a graph in the legend. Add the optional argument `'g'` to select a green line color.
- Use `ax.legend()` to create a legend.
- Use `ax.set_xlabel(title)` and `ax.set_ylabel(title)` to set the axis titles for x and y axis.
- Use `fig.tight_layout()` to clean up margins around the figure.
- Use `fig.savefig(filename)` to save the figure as a pdf, eps, svg, or png file.

4.2.1. Solution

```
In [1]: from xlcalibur import dataclasses, housekeeping
In [2]: from xlcalibur.xcom import Packets
In [3]: import numpy as np

In [4]: from matplotlib import pyplot as plt
In [5]: %matplotlib
Using matplotlib backend: TkAgg

In [6]: infile = \
    dataclasses.XDataFile("/data/xlcal/datafromxcbe/Run010248.dat")

In [7]: all_alignment = [
   ...:     p.Deserialize()
   ...:     for p in infile.Scan(Packets.X_PKT_ALIGNMENT_DATA)
   ...: ]
In [8]: good_alignment = [p for p in all_alignment if p.fit_valid]

In [9]: meanx = np.average([a.center_x for a in good_alignment])
In [10]: meany = np.average([a.center_y for a in good_alignment])
In [11]: t0 = good_alignment[0].time.to_mjd()

In [12]: tdata = [86400 * (a.time.to_mjd() - t0) for a in good_alignment]
In [13]: xdata = [(a.center_x - meanx) / a.scale for a in good_alignment]
In [14]: ydata = [(a.center_y - meany) / a.scale for a in good_alignment]

In [15]: fig, axes = plt.subplots(2, 1)

In [16]: ax = axes[0]
In [17]: ax.plot(tdata, xdata, label="X")
Out[17]: [<matplotlib.lines.Line2D at 0x7f23ebb62fa0>]
In [18]: ax.plot(tdata, ydata, label="Y")
Out[18]: [<matplotlib.lines.Line2D at 0x7f23ebd7c5b0>]

In [19]: ax.legend()
Out[19]: <matplotlib.legend.Legend at 0x7f23ebc94430>
In [20]: ax.set_xlabel("Time [s]")
Out[20]: Text(0.5, 450.24444444444447, 'Time [s]')
In [21]: ax.set_ylabel("Rel. Center [mm]")
Out[21]: Text(61.94444444444443, 0.5, 'Rel. Center [mm]')

In [22]: ax = axes[1]
In [23]: ax.plot(tdata, [a.angle * 180 / np.pi for a in good_alignment], "g")
Out[23]: [<matplotlib.lines.Line2D at 0x7f23ebc94100>]

In [24]: ax.set_xlabel("Time [s]")
Out[24]: Text(0.5, 47.04444444444444, 'Time [s]')
In [25]: ax.set_ylabel(r"Rotation [$^\circ$]")
Out[25]: Text(76.19444444444443, 0.5, 'Rotation [$^\\circ$]')

In [26]: fig.tight_layout()
In [27]: fig.savefig("alignment.pdf")
```