Convenient Fringe-Fit Data Storage in Python Dictionaries for VO2187 Experiment

Leonid Benkevitch, May 19, 2025

Table of Contents

Motivation

In Very Long Baseline Interferometry (VLBI), the correlated and fringe-fit data are stored in the Mark4 format, a 2-level directory tree. The top directory name is a 4-digit number, below it contains directories named `<doy>-<time>[<letter>]`. Each of these directories holds the data files for a scan: the cross- and auto-correlated data, and the fringe-fit data. For example:

```
2187
  ├── 187-1803b
      ├── 0458-020.3HJQAB
      ├── E..3HJQAB
      ├── EE..3HJQAB
      ├── G..3HJQAB
      ├── GE..3HJQAB
      ├── GE.X.6.3HJQAB
```

```
      ├── HE..3HJQAB
      ├── HE.X.2.3HJQAB
    .  .  .  .  .  .
```

The file names only have the baseline letters. The fringe-fit file names also have ".X.". In order to access the information stored in the files the package HOPS is used. For example, here is how to access data items in a single file `GE.X.6.3HJQAB` located in the `"/home/benkev/Work/2187/scratch/Lin_I/2187/187-1803b/"` directory:

```
$ cd /home/benkev/Work/2187/scratch/Lin_I/2187/187-1803b/
```

In Python or IPython:

```
from vpal import fringe_file_manipulation as ffm
f_obj = ffm.FringeFileHandle()
f_obj.load("GE.X.6.3HJQAB")
src = f_obj.source            # Celestial source
phase = f_obj.resid_phas      # Residual phase
dtec = f_obj.dtec             # Differential Total Electron Content
ttag = f_obj.time_tag         # Time or measurement, seconds
mbdelay = f_obj.mbdelay*1e6   # Multiband delay, us
sbdelay = f_obj.sbdelay*1e6   # Single-band delay, us
snr = f_obj.snr               # Signal to noise ratio
```

The data in a single file are for a specific polarization correlation product (for linear polarization, one of XX, XY, YX, YY, or I for pseudo Stokes I (or, further, pI) parameter). It can be found with the following code:

```
import hopstestb as ht
pp_list = \
    ht.get_file_polarization_product_provisional("GE.X.6.3HJQAB")
pp = pp_list[0]          # Polarization Product
```

With such an organization of information, retrieving multiple data items that meet several criteria (for example, the time interval of scanning a particular source for baselines that make up a triangular closure) becomes quite a non-trivial task. Directly using the Mark4 files is also inefficient because a lot of time is spent opening and reading multiple files.

The data needed for a particular analysis can be extracted from all of the fringe-fit file Mark4 database only once. The extracted data can be stored in data structures that provide convenient access. For example, using a good data structure would make it possible to access the whole data cluster related to a celestial source, or a time tag, or a baseline, or a baseline closure triangle.

Python has a built-in dictionary type, currently implemented as a hash table. "Multi-dimensional" dictionaries (or dictionaries of sub-dictionaries of sub-sub-dictionaries...) are ideal containers for storing Mark4 fringe-fit data and for easy access to it. Several Python dictionaries have been developed

that are created once and written to disk using the Python `pickle` module. To access the fringe-fit data, one or more of the appropriate dictionaries need to be “unpickled” into memory as Python `dict` (i.e. “dictionary”) type variables.

Dictionaries to Study Effects of PolConversion

I was given the task of statistically studying the effects of PolConversion by comparing the original experiment VO2187 data with those transformed by PolConvert software. The original VO2187 data are obtained from various receivers with mixed polarization. I. Martí-Vidal *et al* (2016) wrote:
“*As we have already noted, ALMA uses receivers that record the signal on a linear (X/Y) basis, whereas VLBI stations mostly record the signals on a circular (R/L) basis.*”.
“*The [PolConvert] program applies the calibration and conversion equations … for a phased array with linear-feed receivers. It … identifies the antenna(s) with linear feeds used in the observations; and converts the visibilities to a pure circular basis*”
Thus, the data resulted from the PolConversion are totally circularly-polarized. To distinguish the data before and after the PolConversion I (quite provisionally) call the former *linear* and the latter *circular.*

Since I have two Mark4 fringe-fit datasets, the linear and the PolConverted circular, I store their data in the pairs of dictionaries with slightly different names. The linear files and dictionaries are ended with the letter “l”, while the circular files and dictionaries are ended with the letter “c”. The files have also “I” letter to indicate that they contain pure pseudo-Stokes I data. Here are the pickle file lists for both polarizations:

Linear	**Circular**
`idx2187lI.pkl`	`idx2187cI.pkl`
`idxs2187lI.pkl`	`idxs2187cI.pkl`
`idxf2187lI.pkl`	`idxf2187cI.pkl`
`clos2187lI.pkl`	`clos2187cI.pkl`
`clot2187lI.pkl`	`clot2187cI.pkl`

The following two pickle files are common for both linear and circular datasets.

`bls_2187.pkl`	List of the common baselines, used in both linear and circular datasets
`tribl_2187.pkl`	Dictionary of closure triangles pointing at the baseline triplets

The dictionaries, all or only some of them, can be unpickled into memory and assigned Python variable names using the commands:

```
import pickle
```

```
with open('idx2187lI.pkl', 'rb') as finp: idxl = pickle.load(finp)
with open('idx2187cI.pkl', 'rb') as finp: idxc = pickle.load(finp)

with open('idxs2187lI.pkl', 'rb') as finp: idxsl = pickle.load(finp)
with open('idxs2187cI.pkl', 'rb') as finp: idxsc = pickle.load(finp)

with open('idxf2187lI.pkl', 'rb') as finp: idxfl = pickle.load(finp)
with open('idxf2187cI.pkl', 'rb') as finp: idxfc = pickle.load(finp)

with open('clos2187lI.pkl', 'rb') as finp: closl = pickle.load(finp)
with open('clos2187cI.pkl', 'rb') as finp: closc = pickle.load(finp)

with open('clot2187lI.pkl', 'rb') as finp: clotl = pickle.load(finp)
with open('clot2187cI.pkl', 'rb') as finp: clotc = pickle.load(finp)

with open('bls_2187.pkl', 'rb') as finp: bls = pickle.load(finp)
with open('tribl_2187.pkl', 'rb') as finp: tribl = pickle.load(finp)
```

As can be seen, for myself I use the following names: `bls, tribl`, `idxl, idxc` and so on. The dictionaries created from the linear and the circular Mark4 databases have very similar (or the same) structures, so further I will use their “generic” names without the “l” or “c” endings:

Linear	Circular	Generic
`idxl`	`idxc`	`idx`
`idxsl`	`idxsc`	`idxs`
`idxfl`	`idxfc`	`idxf`
`closl`	`closc`	`clos`
`clotl`	`clotc`	`clot`

`bls`: List of common baselines

When I ran PolConvert, it by some reason omitted the 'Y' (i.e. 'Yj', Yebes) station, so it was not present in the baseline list of the PolConverted database. I have not examined the issue yet. Anyway, I have to use only the baselines common for both linear and circular databases. Also, the 'ST' baseline is excluded because the S (Oe, Onsala, north-east) and T (Ow, Onsala, south-west) stations are too close to each other. To avoid selecting the right baselines each time I use them, I prefer keeping the list on disk and read it into the `bls` variable. Here they are:

```
print(bls)
['GE', 'GH', 'GI', 'GM', 'GS', 'GT', 'HE', 'HM', 'HS', 'HT', 'IE',
 'IH', 'IM', 'IS', 'IT', 'ME', 'MS', 'MT', 'SE', 'TE']
```

`tribl:` Dictionary of Closure triangles

The `tribl` dictionary is indexed with 3-letter keys of the closure triangles composed of only the baselines from the `bls` list. Each triangle key has its value a tuple of the three baselines that make up the triangle:

```
tribl['HMS'] → ('HM', 'MS', 'HS')
```

Here is the whole dictionary:

```
tribl →
{'EGH': ('GH', 'HE', 'GE'),
 'EGI': ('GI', 'IE', 'GE'),
 'EGM': ('GM', 'ME', 'GE'),
 'EGS': ('GS', 'SE', 'GE'),
 'EGT': ('GT', 'TE', 'GE'),
 'GHI': ('GI', 'IH', 'GH'),
 'GHM': ('GH', 'HM', 'GM'),
 'GHS': ('GH', 'HS', 'GS'),
 'GHT': ('GH', 'HT', 'GT'),
 'GIM': ('GI', 'IM', 'GM'),
 'GIS': ('GI', 'IS', 'GS'),
 'GIT': ('GI', 'IT', 'GT'),
 'GMS': ('GM', 'MS', 'GS'),
 'GMT': ('GM', 'MT', 'GT'),
 'EHM': ('HM', 'ME', 'HE'),
 'EHS': ('HS', 'SE', 'HE'),
 'EHT': ('HT', 'TE', 'HE'),
 'HMS': ('HM', 'MS', 'HS'),
 'HMT': ('HM', 'MT', 'HT'),
 'HIM': ('IH', 'HM', 'IM'),
 'HIS': ('IH', 'HS', 'IS'),
 'HIT': ('IH', 'HT', 'IT'),
 'EIM': ('IM', 'ME', 'IE'),
 'EIS': ('IS', 'SE', 'IE'),
 'EIT': ('IT', 'TE', 'IE'),
 'IMS': ('IM', 'MS', 'IS'),
 'IMT': ('IM', 'MT', 'IT'),
 'EMS': ('MS', 'SE', 'ME'),
```

```
 'EMT': ('MT', 'TE', 'ME')}
```

Dictionaries of closure triangles are generated with the function `find_baseline_triangles()` from the libvp.py module from a list of baselines. For example:

```
import libvp
libvp.find_baseline_triangles(['IE', 'GI', 'MS', 'SE', 'GT', 'IT', \
                               'TE', 'IH', 'HT', 'IM', 'MT', 'ME']) →
{'GIT': ('GI', 'IT', 'GT'),
 'HIT': ('IH', 'HT', 'IT'),
 'EIM': ('IM', 'ME', 'IE'),
 'EIT': ('IT', 'TE', 'IE'),
 'IMT': ('IM', 'MT', 'IT'),
 'EMS': ('MS', 'SE', 'ME'),
 'EMT': ('MT', 'TE', 'ME')}
```

idx: 3D Dictionary `idx[baseline][polprod][data_item]`

For any of the available baselines and any of the available polarization products, `idx` contains a sub-sub-dictionary of the data items, sorted in time-ascending order. I work with the pseudo-Stokes I data only, so the only `polprod` key is `'I'`. Let's see which baselines are available in linear `idxl` and circular `idxc`:

```
idxl.keys() →
 dict_keys(['HE', 'MT', 'IM', 'ST', 'IT', 'MS', 'IS', 'GH', 'GM',
'IH', 'HM', 'GI', 'MY', 'GT', 'IE', 'EY', 'GE', 'TY', 'GY', 'SY',
'TE', 'SE', 'ME', 'IY', 'GS', 'HY', 'HS', 'HT'])

idxc.keys() →
 dict_keys(['HE', 'MS', 'IM', 'MT', 'IS', 'ST', 'IT', 'HM', 'GH',
'GI', 'GM', 'IH', 'GT', 'GS', 'TE', 'SE', 'ME', 'IE', 'GE', 'HS',
'HT'])
```

Polproducts for an arbitrary bl, say, `'HE'`:

```
idxl['HE'].keys() →
dict_keys(['I'])
```

The possible data items (the same set for any baseline and any polprod) are:

```
idxl['MS']['I'].keys() →
dict_keys(['time', 'source', 'dir', 'file', 'full_fname', 'mbdelay',
'sbdelay', 'snr', 'tot_mbd', 'tot_sbd', 'phase', 'dtec', 'time_tag',
'thour'])
```

All the numerical data items are Numpy float arrays; others are lists of the same size. The size is determined by the number of times the baseline was used in the course of observation.

`'time_tag':` time in seconds from an epoch (just the value from the fringe-fit file).
`'time':` time in seconds from the session start.
`'thour':` time in hours from the session start (just time/3600).
`'source':` celestial source, like '1803+784' or 'OJ287'.
`'mbdelay':` multiband delay in picoseconds.
`'sbdelay':` single-band delay in picoseconds.
`'phase':` residual phase (resid_phas value from the fringe-fit file).
`'tot_mbd':` total multiband delay
`'tot_sbd':` total single-band delay
`'dtec':` differential total electron content, dTEC
`'snr':` signal-to-noise ratio
`'full_fname':` absolute path to the fringe-fit file with these data items
`'dir':` Mark4 directory name <doy>-<time>[<letter>], like '187-2037b' or such.
`'file':` Mark4 file name, like 'MS.X.6.3HJQPD'

The beauty of this dictionary-based approach is the ease of tracing back any dubitable value: I can always see the source of the value, find the original Mark4 file and double-check it.

For example, we can print out several data items from the linear Mark4 dataset:

```
ixms = idxl['MS']['I']      # Set ixms to the ‘MS’ subdictionary

# Print time (h), source, mbd, phase, snr values from 20 to 24:
for i in range(20,25):
    print("%5.2f %8s %7.2f %6.2f %7.2f" % (ixms['thour'][i],
          ixms['source'][i], ixms['mbdelay'][i], ixms['phase'][i],
          ixms['snr'][i]))
```

```
2.35 0059+581  460.60 350.33  217.79
2.44 1849+670 -493.41 146.12   79.24
2.49 0613+570 1888.99 212.69  136.39
2.63    3C274 2499.34  49.08   41.27
2.81 1849+670 -203.81  44.78   72.11
```

Here are the fringe-fit files these data were extracted from:

```
ixms['full_fname'][20:25] →
['/home/benkev/Work/2187/scratch/Lin_I/2187/187-2021/MS.X.7.3HJQNK',
 '/home/benkev/Work/2187/scratch/Lin_I/2187/187-2026/MS.X.4.3HJQO4',
 '/home/benkev/Work/2187/scratch/Lin_I/2187/187-2029/MS.X.12.3HJQOD',
 '/home/benkev/Work/2187/scratch/Lin_I/2187/187-2037b/MS.X.6.3HJQPD',
 '/home/benkev/Work/2187/scratch/Lin_I/2187/187-2048/MS.X.5.3HJQQE']
```

`idxs`: 4D Dictionary `idxs[source][time][baseline][data_item]`

For each available celestial source (as the first key) `idxs` contains the times in seconds from the session start available for this source. Each time, in turn, is a key pointing to the baselines available for this source at this time. Each baseline key, in turn, points at the data items. The available sources for the circularly polarized data in `idxsc`:

```
idxsc.keys() →
dict_keys(['0454-234', '0133+476', '2214+241', '2229+695','0738+491',
           '1418+546', '1846+322', '1555+001', 'OJ287', '1144+402',
           '1040+244', '1504+377', '0059+581', '3C446', '2144+092',
           '0119+115', '3C418', '0109+224', '1053+704', '1803+784',
           '0458-020', '0202+319', '1324+224', '1923+210',
           '1124-186', '2113+293', '0955+476', '1213-172',
           '1849+670', '0823+033', 'DA426', '0727-115', '0420+022',
           '0017+200', '0800+618', '0221+067', '1749+096',
           '1741-038', '1958-179', '0003-066', '1015+359',
           '0602+673', '0537-286', '1149-084', '2059+034',
           '1751+288', '1606+106', '0748+126', '2126-158',
           '0529+483', '0718+793', '3C274', '1908-201', '1705+018',
           '1639-062', '2255-282', '0613+570', '1308+328',
           '0322+222', '1639+230', '1243-072', '0115-214', '1406-76',
           '1145+268', '0736+017', '1806+456', '0307+380', '0632-35',
           '2325+093', '0235+164', '1519-273', '1657-261', '1255-16',
           '0847-120'])
```

For example, select an arbitrary source `0454-234` and find all the related observation times:

```
idxsc['0454-234'].keys() →
dict_keys([739.0, 9018.0, 10393.0, 11696.0, 61666.0, 63410.0,
           64836.0, 66763.0, 68124.0, 69542.0, 80098.0, 82947.0,
```

```
           84426.0, 85963.0])
```

Select an arbitrary time 11696.0 and find the involved baselines:

```
idxsc['0454-234'][11696.0].keys() →
dict_keys(['IH', 'IM', 'HM'])
```

For one of the baselines, `'HM'`, find the data item names:

```
idxsc['0454-234'][11696.0]['HM'].keys() →
dict_keys(['mbdelay', 'sbdelay', 'tot_mbd', 'tot_sbd', 'phase',
           'dtec', 'snr', 'pol_prod', 'dir', 'file', 'full_fname',
           'time_tag', 'time', 'thour'])
```

The data items here are the same as in the `idx` dictionaries, but unlike those in `idx`, the data items in `idxs` point at elemental values, and not at Numpy arrays or lists. We can see them all:

```
idxsc['0454-234'][11696.0]['HM'] →
{'mbdelay': 2400.059252977371,
'sbdelay': 4550.499841570854,
'tot_mbd': 11986.570896186558,
'tot_sbd': 11986.573046627314,
'phase': 35.541690826416016,
'dtec': 7.376163386555481,
'snr': 101.31890869140625,
'pol_prod': 'I',
'dir': '187-2114',
'file': 'HM.X.2.3K3GOB',
'full_fname': '/home/benkev/Work/vo2187_exprm/DiFX_pconv/2187/
               187-2114/HM.X.2.3K3GOB',
'time_tag': 1341609296.5,
'time': 11696.0,
'thour': 3.2488888888888887}
```

idxf: 3D Dictionary idxf[dir][file]

These dictionaries facilitate data retrieving by the directory and file names as two indices. For example, in the directory `188-0435a` all the fringe-fit files are:

```
idxfl['188-0435a'].keys() →
dict_keys(['HT.X.4.3HJS31', 'HS.X.1.3HJS31', 'ST.X.3.3HJS31',
           'IH.X.5.3HJS31', 'IS.X.6.3HJS31', 'IT.X.2.3HJS31'])
```

Choose an arbitrary file `HT.X.4.3HJS31` and view the data it contains:

```
idxfl['188-0435a']['HT.X.4.3HJS31'] →
{'source': '1803+784',
 'time': 38118.0,
 'bl': 'HT',
 'mbdelay': -2092.3358388245106,
 'sbdelay': -3383.500035852194,
 'phase': 100.13414764404297,
 'dtec': 26.433500788449898,
 'snr': 141.68231201171875,
 'pol_prod': 'I',
 'tot_mbd': -8538.048102473467,
 'tot_sbd': -8538.04939363753,
 'full_fname':
'/home/benkev/Work/2187/scratch/Lin_I/2187/188-0435a/HT.X.4.3HJS31',
 'time_tag': 1341635718.5,
 'thour': 10.588333333333333}
```

clos: 3D Dictionary of closures by source, clos[source][triangle][data_item]

These dictionaries contain phase and delay closures for mbd, sbd, total mbd, and total sbd. For a celestial source and any available closure triangle it contains arrays of the closure values in time-ascending order. Again, take a source, `1639-062`, and view the closure triangles available while the source was observed:

```
closl['1639-062'].keys()
dict_keys(['EGS', 'EGT', 'EGM', 'EGH', 'GHM', 'EHM', 'GHI', 'GIM',
           'HIM'])
```

Select a triangle, EGH, and list the available data item keys:

```
closl['1639-062'][‘EGH’].keys()
```

```
dict_keys(['bl', 'time', 'thour', 'time_tag', 'cloph', 'tau_mbd',
           'tau_sbd', 'tau_tmbd', 'tau_tsbd', 'phase', 'dtec', 'mbd',
           'sbd', 'tmbd', 'tsbd', 'snr', 'pol_prod', 'file', 'dir'])
```

For the brief description of the common data items see the idx section. The closure data keys are as follows:

`'cloph'`: closure phase
`'tau_mbd'`: multiband delay closure
`'tau_sbd'`: single-band delay closure
`'tau_tmbd'`: total multiband delay closure
`'tau_tsbd'`: total single-band delay closure

For convenience, the data items contain the triplets of values used for the closure computations as well as the triplets of files that provided the data. The triplets are arrays or lists of Nx3 dimensionality, where N is the number of time counts for this particular observation of the source with the closure triandgle:

`'phase'`: Nx3 array of phases giving closure phase in `'cloph'`
`'mbd'`: Nx3 array of `mbd` giving multiband delay closure in `'tau_mbd'`
`'sbd'`: Nx3 array of `sbd` giving single-band delay closure in `'tau_sbd'`
`'tmbd'`: Nx3 array of `tot_mbd` giving total multiband delay closure in `'tau_tmbd'`
`'tsbd'`: Nx3 array of `tot_sbd` giving total single-band delay closure in `'tau_tsbd'`

In the example considered, there are N = 5 times of observations with the triangle EGH:

```
closl['1639-062']['EGH']['thour'] →
array([ 9.441,  9.805, 10.966, 11.382, 13.054])
```

The closure phases at these times:

```
closl['1639-062']['EGH']['cloph'] →
array([ 2.049, 17.289, 12.142,  8.68 , -3.97 ])
```

The triplets of phases used for the closures' computation:

```
closl['1639-062']['EGH']['phase'] →
array([[354.896, 160.623, 153.469],
       [171.142, 338.956, 132.808],
       [ 31.882, 215.796, 235.536],
       [284.023, 139.899,  55.242],
```

```
      [305.597, 127.428,  76.995]])
```

The triplets of directories and files with the phase used for the closures' computation:

```
closl['1639-062']['EGH']['dir'] →
[['188-0326b', '188-0326b', '188-0326b'],
['188-0348', '188-0348', '188-0348'],
['188-0457b', '188-0457b', '188-0457b'],
['188-0522b', '188-0522b', '188-0522b'],
['188-0703', '188-0703', '188-0703']]

closl['1639-062']['EGH']['file'] →
[['GH.X.4.3HJRVY', 'HE.X.3.3HJRVY', 'GE.X.5.3HJRVY'],
['GH.X.4.3HJRY9', 'HE.X.3.3HJRY9', 'GE.X.6.3HJRY9'],
['GH.X.5.3HJS5B', 'HE.X.3.3HJS5B', 'GE.X.4.3HJS5B'],
['GH.X.5.3HJS7W', 'HE.X.3.3HJS7W', 'GE.X.6.3HJS7W'],
['GH.X.4.3HJSI5', 'HE.X.2.3HJSI5', 'GE.X.6.3HJSI5']]
```

`clot`: 3D Dictionary of closures by triangle, `clot[triangle][source][data_item]`

The `clot` dictionary contain exactly the same information as `clos` does, but the first two indices are permuted. So, for example, the data item contents for both

`closl['0003-066']['EHM']`
and
`clotl['EHM']['0003-066']`
are the same:

```
{'bl':      [('HM', 'ME', 'HE')],
'time':     array([ 65587.,  71534.]),
'thour':    array([ 18.218611,  19.870556]),
'time_tag': array([  1.341663e+09,   1.341669e+09]),
'cloph':    array([  7.91127 , -24.428436]),
'tau_mbd':  array([ 12.167962,  12.668082]),
'tau_sbd':  array([-311.999931, -269.999997]),
'tau_tmbd': array([  1.216797e-05,   1.266797e-05]),
'tau_tsbd': array([-0.000312, -0.00027 ]),
'phase': array([[ 153.47937 ,  118.823486,  264.391586],
```

```
                [ 100.181648,  320.209343,   84.819427]]),
'dtec':  array([[ 10.548944, -17.81798 ,  -7.211685],
                [ -9.336928, -19.21364 , -28.841804]]),
'mbd':   array([[-3076.259745,  1711.800811, -1376.626897],
                [-2781.886607,  1668.498036, -1126.056653]]),
'sbd':  array([[-2915.499965,  1010.000007, -1593.500027],
               [-1562.500023,  1242.000028,   -50.499999]]),
'tmbd': array([[ -5918.644996,   5812.875947,   -105.769061],
               [  1609.762262,   8714.191619,  10323.953868]]),
'tsbd': array([[ -5918.644835,   5812.875245,   -105.769278],
               [  1609.763481,   8714.191192,  10323.954943]]),
'snr': array([[ 227.791122,  232.53447 ,  118.25795 ],
              [ 249.248672,  225.304962,  141.638458]]),
'pol_prod': ['I'],
'file': [['HM.X.4.3HJTE3', 'ME.X.2.3HJTE3', 'HE.X.3.3HJTE3'],
         ['HM.X.1.3HJTON', 'ME.X.3.3HJTON', 'HE.X.4.3HJTON']],
'dir': [['188-1213',  '188-1213',  '188-1213'],
        ['188-1352a', '188-1352a', '188-1352a']]}
```

The triangle EHM and the source 0003-066 only have N = 2 time counts, so the the data are quite terse.

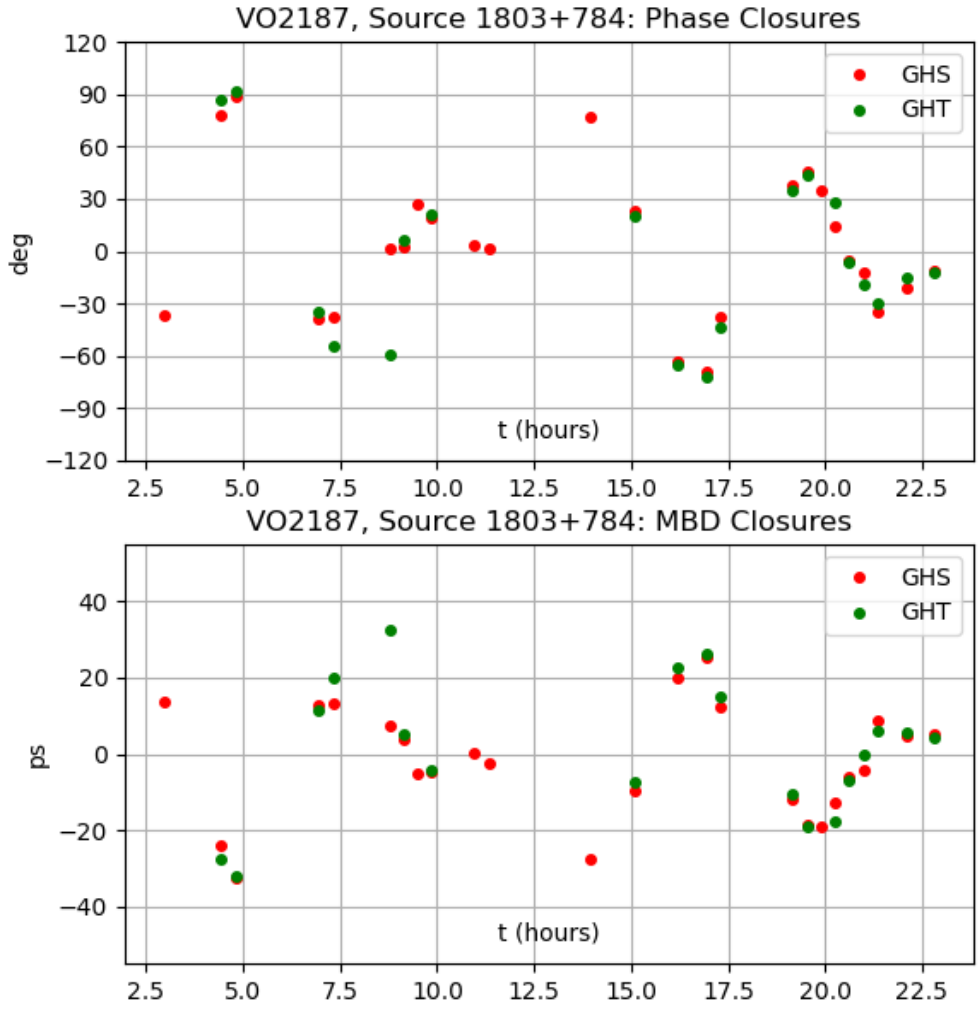
A practical example of using the dictionaries

The code below plots phase and MBD closures for the station triangles GHS and GHT during the source 1803+784 observations. The triangles are almost coincident because the stations S and T are located close to one another.

```
import matplotlib.pyplot as pl
import numpy as np
import pickle

with open('clos2187lI.pkl', 'rb') as finp: closl = pickle.load(finp)

sr = '1803+784'

thr_ghs = closl[sr]['GHS']['thour']
thr_ght = closl[sr]['GHT']['thour']

clp_ghs = closl[sr]['GHS']['cloph']
```

```
clp_ght = closl[sr]['GHT']['cloph']

clm_ghs = closl[sr]['GHS']['tau_mbd']
clm_ght = closl[sr]['GHT']['tau_mbd']

f1 = pl.figure(figsize=(6.4, 7))

ax1 = pl.subplot(2,1,1)
ax1.plot(thr_ghs, clp_ghs, 'r.', ms=8, label='GHS')
ax1.plot(thr_ght, clp_ght, 'g.', ms=8, label='GHT')
ax1.grid(1)
ax1.set_yticks(-120 + 30*np.arange(9))
ax1.set_title("VO2187, Source 1803+784: Phase Closures")
ax1.set_xlabel("t (hours)", labelpad=-35)
ax1.set_ylabel("deg")
ax1.legend()

ax2 = pl.subplot(2,1,2)
ax2.plot(thr_ghs, clm_ghs, 'r.', ms=8, label='GHS')
ax2.plot(thr_ght, clm_ght, 'g.', ms=8, label='GHT')
ax2.grid(1)
ax2.set_ylim(-55, 55)
ax2.set_title("VO2187, Source 1803+784: MBD Closures")
ax2.set_xlabel("t (hours)", labelpad=-35)
ax2.set_ylabel("ps")
ax2.legend()
```

Here is the result:



The Software to Create the Dictionaries

Both the VO2187 fringe-fit dictionaries and the software to create them and work with them are in the GitHub repository, which can be cloned to any location on your local disc in the directory `vlbipol` using the command

```
$ git clone git@github.com:benkev/vlbipol.git
$ cd vlbipol
$ ipython --pylab
```

Then the idx, idxs, idxf, clos, and clot dictionaries can be unpickled from the *.pkl files on disk and used immediately. However, I noticed that sometimes the versions of *.pkl files are incompatible with the user's Python version. This issue can be easily resolved by creating, pickling and saving the pickle files using the software described below with your own Python version.

There is one more reason to recreate the pickled dictionaries using the software described below. In the dictionaries in the GitHub repository the data items `full_fname` contain full paths to the locations on my workstation. Recreating the dictionary on your workstation (or on `demi` server) would set `full_fname` to correct paths.

Function make_idx(base_dir, pol='lin', max_depth=2)

The idx, idxs, idxf dictionaries are created directly from the Mark4 dataset with the use of function `make_idx(base_dir, pol='lin', max_depth=2)` from the module `librd.py`.

Parameters:

`base_dir:` full path to the 4-digit-named directory with the Mark4 fringe-fit data.

`pol`: `'lin'` - linear polarization, `'cir'` - circular polarization

The circularly-polarized Mark4 fringe-fit data files obtained through the conversion chain `PolConvert → difx2mark4 → fourfit` retain the 'linear' notations of their polarization products, so `pol='cir'` parameter makes the replacement `'XX'→'LL', 'XY'→'LR', 'YX'→'RL', 'YY'→'RR', ['XX', 'YY'] → 'I'`.

`max_depth:` makes the algorithm recurse no deeper than max_depth below the root directory given in `base_dir`

For example, in Python or IPython:

```
from librd import make_idx

# Linearly polarized Mark4 fringe-fit database
linI_2187 = "/home/benkev/Work/2187/scratch/Lin_I/2187"
idxl, idxsl, idxfl = make_idx(linI_2187) # Create dictionaries
# Pickle and save on disk
with open('idx2187lI.pkl', 'wb') as fout: pickle.dump(idxl, fout)
with open('idxs2187lI.pkl', 'wb') as fout: pickle.dump(idxsl, fout)
with open('idxf2187lI.pkl', 'wb') as fout: pickle.dump(idxfl, fout)
```

Use `'cir'` second parameter when creating dictionaries with the circularly polarized data:

```
# Circularly polarized Mark4 fringe-fit database
cirI_2187 = "/home/benkev/Work/vo2187_exprm/DiFX_pconv/2187"
idxc, idxsc, idxfc = make_idx(cirI_2187, 'cir') # Create dictionaries
# Pickle and save on disk
with open('idx2187cI.pkl', 'wb') as fout: pickle.dump(idxc, fout)
with open('idxs2187cI.pkl', 'wb') as fout: pickle.dump(idxsc, fout)
with open('idxf2187cI.pkl', 'wb') as fout: pickle.dump(idxfc, fout)
```

NOTE: The `vpal` module imported here changes the matplotlib backend to "Agg", which is non-interactive and can't show GUI windows, it is meant for saving images in files only. You can check which backend is in use:

```
import matplotlib
matplotlib.get_backend()
```

If you import librd and want to use matplotlib code for plotting on screen, you have to reset the backend to interactive. For example:

```
import matplotlib
matplotlib.use('qtagg', force=True)  # force reset the backend
```

Function make_closure_dic(idxs, bls=None)

Creates a dictionary of all possible closures

```
clos[src][tri][data_item]
```

from the dictionary

```
idxs[src][time][bl][data_item]
```

The `bls` parameter is a list of allowed baselines. If not given, all the baselines contained in `idxs` will be used. In the current VO2187 study, the baseline ST and all the baselines with the Y station are excluded by the PolConvert software, so after calling `closl = make_closure_dic(idxsl)` and `closc = make_closure_dic(idxsc)` without the `bls` parameter will create the `closl` dictionary with the closures including the 'Y' station and the `closc` dictionary without those. Also, both will contain closures with the 'ST' baseline. Therefore, to create the `closl` and `closc` dictionaries with similar structures, calling `make_closure_dic()` with the `bls` parameter containing the list of baselines common for both linear and circular polarizations is necessary.

The clos dictionary returned contains not only the closures, but also the data triplets used to compute the closures. All the numeric data are in arrays sorted in time ascending order. Here is an example of creating both `closl` and `closc`:

```
from libvp import make_closure_dic
with open('bls_2187.pkl', 'rb') as finp: bls = pickle.load(finp)
with open('idxs2187lI.pkl', 'rb') as finp: idxsl = pickle.load(finp)
with open('idxs2187cI.pkl', 'rb') as finp: idxsc = pickle.load(finp)
closl = make_closure_dic(idxsl, bls) # With linearly polarized data
closc = make_closure_dic(idxsc, bls) # With circularly polarized data
```

Function clos_to_clot(clos, tribl=None, bls=None)

Creates dictionary of all possible closures with a triangle as first key rearranging a `clos` dictionary into `clot` by permuting the first two indices `src` and `tri`:

```
clos[src][tri][data_item] → clot[tri][src][data_item]
```

The `tribl` parameter is a dictionary `tribl[tr] --> (bl1, bl2, bl3)`
If not provided, it is created from the `bls` list of allowed baselines. If neither `tribl` nor `bls` are provided, tribl is loaded from file `tribl_2107.pkl` on disk.

The returned `clot` dictionary contains not only the closures, but also the data triplets used to compute the closures including the triplets of Mark4 fringe-fit file directories and file names. All the numeric data are in arrays sorted in time ascending order.

Here is an example of creating both `clotl` and `clotc` from `closl` and `closc` saved on disk as the pickled files:

```
from libvp import clos_to_clot
with open('tribl_2187.pkl', 'rb') as finp: tribl = pickle.load(finp)
with open('clos2187lI.pkl', 'rb') as finp: closl = pickle.load(finp)
with open('clos2187cI.pkl', 'rb') as finp: closc = pickle.load(finp)
clotl = clos_to_clot(closl, tribl)
clotc = clos_to_clot(closc, tribl)
```

Creating all dictionaries at once: script make_idx_2187.py

This script has no command-line parameters. Instead, the user should edit its text to set 3 variables to the values needed: save_pkl, linI_2187, and cirI_2187. Their current values are

```
save_pkl = False   # If True, pickle and save on disk as *.pkl
linI_2187 = "/home/benkev/Work/2187/scratch/Lin_I/2187"
cirI_2187 = "/home/benkev/Work/vo2187_exprm/DiFX_pconv/2187"
```

Set `linI_2187` to the path to the Mark4 directory with linearly polarized fringe-fit files.
Set `cirI_2187` to the path to the Mark4 directory with circularly polarized fringe-fit files.
You can perform a dry run with the switch `save_pkl = False`. The dictionaries will be created in memory, but not saved to disk. If there are no errors and the results look good, either run the script again with `save_pkl = True`, or save the results by hand:

```
with open('idx2187lI.pkl', 'wb') as fout: pickle.dump(idxl, fout)
with open('idxs2187lI.pkl', 'wb') as fout: pickle.dump(idxsl, fout)
with open('idxf2187lI.pkl', 'wb') as fout: pickle.dump(idxfl, fout)

with open('idx2187cI.pkl', 'wb') as fout: pickle.dump(idxc, fout)
with open('idxs2187cI.pkl', 'wb') as fout: pickle.dump(idxsc, fout)
with open('idxf2187cI.pkl', 'wb') as fout: pickle.dump(idxfc, fout)

with open('bls_2187.pkl', 'wb') as fout: pickle.dump(bls, fout)
with open('tribl_2187.pkl', 'wb') as fout: pickle.dump(tribl, fout)

with open('clos2187lI.pkl', 'wb') as fout: pickle.dump(closl, fout)
with open('clos2187cI.pkl', 'wb') as fout: pickle.dump(closc, fout)

with open('clot2187lI.pkl', 'wb') as fout: pickle.dump(clotl, fout)
with open('clot2187cI.pkl', 'wb') as fout: pickle.dump(clotc, fout)
```

To prevent accidental corruption of the pickled dictionaries, write-protect them using

```
$ chmod -w *.pkl
```

In case you want recreate them, before running make_idx_2187.py script with
`save_pkl = True`
execute the command

```
$ chmod ug+w *.pkl
```

I. Martí-Vidal *et al*, Calibration of mixed-polarization interferometric observations. Tools for the reduction of interferometric data from elements with linear and circular polarization receivers, A&A, 2016