JUMPING JIVE

730884 — JUMPING JIVE — H2020-INFRADEV-2016-2017/H2020-INFRADEV-2016-1

Deliverable 8.2

Submission date: 19.07.2017

Jumping JIVE, deliverable D8.2: Document on SCHED

Bob Eldering, Mark Kettenis, Des Small, Arpad Szomoru

1. Introduction

The full title of this deliverable is “Document detailing what functionality of SCHED will be re-written, and method to be followed, based partly on input from SRFF”, where SRFF stands for the SCHED Re-Factoring Forum, a group of SCHED experts and advanced users.

2. Background

The program SCHED was written in the early 1980s in order to provide a common, generalised user interface for scheduling VLBI observations. It does so by combining observing parameters, source catalogues and frequency setup catalogues, which describe the detailed settings at all different stations. This is by no means trivial, considering that all telescopes are different, in terms of location, architecture, hardware limitations, equipment and frequency coverage. The resulting schedule comes in the form of a so-called VEX file, for which an international standard was defined, a plain-text human-readable equivalent of a database. This file is sent to the stations, where the control computer parses the schedule and translates it into a series of commands to the telescope control system and the recording/transmitting equipment. The decades-old code base makes the program extremely hard to modify in order to adapt it to the modern-day demands of VLBI networks.

The aim of this task is to re-factor the existing code, rather than re-writing all from scratch. This will be done by separating out well-defined bits of functionality and re-writing these as individual modules in a modern language. In this way a “gold standard” will remain available throughout the process, enabling an incremental replacement of the original code base. Static parts of the code that do not need frequent modifications will be kept as they are. The end product will be a modernised version of SCHED that will be far easier to adapt, written in a widely used and well-known programming language. It will be usable for all aspects of the proposal-observational cycle, which means during the proposal phase, the programming by the PI and finally the actual generation of an observing schedule by JIVE staff. This functionality will be essential for VLBI users in the SKA era. As an aside, a forum of SCHED experts and users will be set up, to ensure that the engineering effort will keep the different needs of different VLBI networks in mind.

3. Feedback from forum

One telecon was convened with the SCHED Re-Factoring Forum, on April 4 2017. Apart from explaining the scope of the project and their involvement to the members, they were asked to provide a wishlist with possible improvements and modifications of the software. This list would not be used as a to-do list by the developers, but rather serve as a reminder of what modifications the project in the end should enable. The main items on that list are

- PGPLOT, the plotting package used in SCHED was mentioned several times, as being hard to install and problematic for many reasons.
- Support for VEX 2 (Section 6), which will be needed soon.
- Migrating catalogs to a central database or revision controlled repository. However the off-line use of SCHED should remain possible.
- An integrated sensitivity calculator would be useful.
- Keep up with hardware changes.

4. Plan of action

Through discussions and investigations of the functionality of SCHED and the way it is currently used, the team at JIVE set out to specify the overall goal of the project, the way to reach this goal and the actions needed.

Very early on, the decision was made that Python would be used in this project. Its broad acceptance in the astronomical community, extensive scientific libraries and ease of use make it a natural choice.

The current SCHED code exists in an SVN repository at NRAO. As this project intends to involve outside parties in the development of new functionality, a GitHub repository was created (jive-vlbi), where a copy of the SCHED code, including history, will be stored. This copy will be a tracking branch, and will be kept up-to-date with the “official” SVN version.

At first, the translation of Fortran data structures into C was considered, combined with the use of SWIG [1] to access these data structures from Python. However, soon it transpired that SWIG needs dynamically linked libraries. This led the team to f2py [2], a project within numpy, which creates Python modules to interface to Fortran. Although this provides no access to Fortran parameters, this was not considered to be a show-stopper.

Wrapping SCHED’s top level subroutines with Python modules would allow us to replace the main program with a Python version.

Having decided on the language, environment and interface, a number of specific cases were selected for implementation.

1. Replace the SCHED reader: a Python keyin reader already exists but most likely will need modifications.
2. Implement VEX2 support: fairly urgently needed.
3. Implement support for the Polyphase Filter Bank (PFB) mode of the DBBC backend, used throughout the EVN.

Item 3 in this list will only be attempted after the first two items are successfully completed. These two are well defined sub-projects that will

be very instructive and will help determine how to move to more complicated issues.

After this, more features will be tackled, depending on available resources, after consultation with the SRFF. We will also investigate PGPLOT alternatives, notably the Giza Python package based on the Cairo library [3].

In the following sections some detailed technical background is given.

5. SCHED catalogue files and Keyin format

The SCHED distribution includes a set of catalogues of source positions and station configuration information, which are specified using the "keyin" format. As section 1.2 of the SCHED manual states (p.8): "All input parameters to SCHED are in the keyin free format, named after Tim Pearson’s subroutine that is used to read it. The important features of that format for SCHED are described here. This description is not complete and users of the Caltech package should refer to other documentation for useful capabilities of keyin input that are not normally used for SCHED."

The keyin format is also used in AIPS for TSYS files, but otherwise has not been widely adopted. The specification in the SCHED file is informal; in practice the precise specification is the interpretation that SCHED (or AIPS) gives a keyin file when it is read. Any attempt to move SCHED into the Python era will require addressing this component: Fortran is not a language well suited to parsing and string handling, and parsers developed in Fortran can be hard to maintain and often (as here) do not follow a precise formal specification.

A few years ago, one of the authors of this document (Des Small) built a keyin reader in Python, which has since found a use in the part of the CASA package that imports AIPS-style TSYS data. The data is imported into Python dictionaries, from which it is straightforward to export it to, for example json. An entry in the source catalogue in keyin format:

```
SOURCE='2358+189','J0001+1914'
    RA=00:01:08.6215684 DEC=  19:14:33.801700 RAERR=   0.017
DECERR=   0.029  CALCODE='V'
    REMARKS='GSFC 2015a astro solution, unpublished   2396
observations.'
    FLUX =   2.20,  0.07,  0.07,   8.40,  0.12,  0.11    FLUXREF = 'gsfc2014b'
/
```

is transformed by this code into json format as:

```
  {
      "CALCODE": "V",
      "DEC": 69273.8017,
      "DECERR": 0.029,
      "FLUX": [
          2.2,
          0.07,
          0.07,
          8.4,
          0.12,
          0.11
      ],
      "FLUXREF": "gsfc2014b",
      "RA": 68.6215684,
      "RAERR": 0.017,
      "REMARKS": "GSFC 2015a astro solution, unpublished   2396
observations.",
      "SOURCE": [
          "2358+189",
          "J0001+1914"
      ]
  }
```

Note that neither Python nor Json dictionaries preserve order, and the latter also doesn't permit comments, which may make it unsuitable for catalog management. The Python keyin reader also doesn't have a formal specification, partly because it is subject to revision when examples are found where its interpretation differ from the official Fortran parser, and it doesn't support even some of the documented features of the format, such as inline arithmetic, which are not in practice used in the catalogues.

The Python code is, however, short (under 250 lines, including whitespace and comments) and straightforward. The parsing is done by recursive descent, while tokenising is done by regular expressions.

6. SCHED and VEX

VEX (VLBI EXperiment) is a standard for expressing VLBI experiment schedules. Schedules in VEX format are used by most VLBI telescopes. SCHED can create VEX schedules and this is its default output format. The current version of the standard is 1.5 [4], which has been in use since the late 1990s.

An update to the standard is in progress and nearing completion [5]. This new version will be released as version 2. The main goal of this update is to support a wider range of VLBI data acquisition hardware as version 1.5 is not capable of properly describing the current generation of VLBI equipment. This is achieved by the introduction of a much more flexible description of equipment and how this equipment is connected together. This makes it much easier to specify schedules for data acquisition systems that consist of multiple parallel (digital) back-ends and/or recorders. But VEX 2 includes many additional improvements such as support for complex sampling, observation of spacecraft, specification of intent, etc.

The plan is to extend SCHED to emit VEX 2 alongside VEX 1.5 to assist the verification process of VEX 2. The ability to create a VEX 2 schedule is a pre requisite to testing the software that executes schedules at the telescopes such as the NASA Field System used at most EVN telescopes. Extending SCHED with VEX 2 support will be be done through a new implementation in Python which will leverage the work done to make SCHED's internal data structures available in Python. We expect that this will be less work than extending the existing VEX 1.5 code written in Fortran as string handling/formatting is much easier to do in Python. The initial implementation will focus on supporting the DBBC digital backend in combination with the FILA10G board as this is the most common hardware combination in the EVN that isn't supported properly in VEX 1.5. The ability to create both VEX 1.5 and VEX 2 schedules will allow us to verify the new implementation by comparing the output of schedules created for the DBBC with and without a FILA10G.

Once the VEX 2 implementation has been finished and tested we will consider adding VEX 1.5 support as well. The basic structure of VEX 1.5 and VEX 2 is the same so this would be a relatively low effort. This would allow us to completely replace the existing Fortran VEX code in SCHED with something that is much easier to maintain in the future.

7. f2py

As mentioned earlier, in order to interface from Python to the Fortran SCHED code, we use a program called f2py. F2py takes Fortran code and generates a Python extension module. This Python module exposes the Fortran subroutines as Python functions and the Fortran COMMON blocks as Python classes.

F2py is part of the NumPy project. NumPy is a package for scientific computing in Python [6]. One of the main features of NumPy is that it provides multidimensional array objects. F2py uses these multidimensional arrays to represent Fortran arrays.

To determine whether f2py would be suitable for our purposes, we had to show it would work for two specific cases. The first case is to replace the SCHED main program with Python code.

The second case consists of creating methods to reorder Fortran data structures. The Fortran data structures used in SCHED consist mainly of multiple arrays of single data elements, where in a modern programming language one would use an array of structures containing these data elements. We want to apply this reorganisation to the data structures representing the station catalogs.

7.1. Replacement of main program

We first list the SCHED main program, followed by the Python version.

Main program, Fortran:

```
      PROGRAM SCHED
      INCLUDE   'sched.inc'
     INCLUDE   'schset.inc'
     INCLUDE   'schfreq.inc'
     INCLUDE   'srlist.inc'
     LOGICAL   MKFILES, RESTART

     CALL VERSCHED( VERNUM, VERSION )
     CALL STMSG

     RESTART = .FALSE.

  100 CONTINUE
```

```
      NSET = 0
      SRLN = 0

      CALL INPUT
      CALL DEFAULTS
      CALL SCHPRE
      CALL CHKSC1
      CALL SCHOPT
      CALL DOPFQ
      CALL GETSUN
      CALL CHKSCN
      CALL SCHSUM( RESTART )
      CALL FLUXH( ILOG, LOGFILE )
      CALL PLOTTER( MKFILES, RESTART )
      IF( RESTART ) THEN
        CALL DELSCR( RESTART )
        CALL WLOG( 0, ' ' )
        CALL WLOG( 0,
     1     ' ==================  RESTART  ====================' )
        CALL WLOG( 0, ' ' )
        GO TO 100
      END IF

     IF( MKFILES .AND. OPTMODE .NE. 'UPTIME' .AND. .NOT. NOSET ) THEN
      CALL SCNRANGE
      CALL OMSOUT( RESTART )
      CALL VEXOUT
      IF( DOVSOP ) CALL VSOPWRT
      CALL FLAGS
      CALL STAFILES
     END IF

     CALL DELSCR( .FALSE. )
     CALL PUTOUT( ' -------  Finished  ----------' )
     STOP
     END
```

Running f2py on all the called subroutines results in a Python module which we have named “schedlib”. This is then incorporated in a Python version of the main program:

Main Program, Python:

```
import schedlib

schedlib.vern.vernum, schedlib.verc.version = schedlib.versched()
schedlib.stmsg()

restart = False

while True:
   schedlib.setn1.nset = 0
   schedlib.srlis.srln = 0
   schedlib.input()
   schedlib.defaults()
   schedlib.schpre()
   schedlib.schopt()
   schedlib.dopfq()
   schedlib.getsun()
   schedlib.chkscn()
   schedlib.schsum(restart)
   schedlib.fluxh(29, schedlib.schsco.logfile)
   mkfiles, restart = schedlib.plotter(restart)

   if restart:
      delscr(restart)
      wlog(0, " ")
      wlog(0, " ==================  RESTART  ==================== ")
      wlog(0, " ")
   else:
      break

if mkfiles and schedlib.schsco.optmode != "UPTIME" and not schedlib.schcon.noset:
   schedlib.scnrange()
   schedlib.omsout(restart)
   schedlib.vexout()

   if schedlib.schcon.dovsop:
      schedlib.vsopwrt()

   schedlib.flags()
   schedlib.stafiles()
   schedlib.delscr(False)
schedlib.putout(" ------  Finished  ---------- ")
```

As can be seen, the translation from Fortran to Python is quite straightforward. The most notable differences are:

- When a Fortran function changes the value of an argument, the corresponding Python version of the function returns the new values instead (see the functions versched and plotter). This a more natural way to handle output parameters in Python, since the Python built-in types int (vernum), string (version) and bool (restart and mkfiles) are immutable, which makes using them as output parameters impossible.
- F2py does not automatically detect output parameters, these have to be manually marked. This is done in the so-called signature file which f2py generates as an intermediate product. The signature file contains the signature of the Fortran functions and COMMON blocks that have been extracted from the source code. For example, the part of the signature file which represents the function versched looks like this:

```
        subroutine versched(vernum,version) ! in :schedlib:Sched/versched.f
  real :: vernum
  character*(*) :: version
end subroutine versched
```

  Since version is an output parameter now, the length of the string has to be specified. The number 40 is derived from the only call to versched, in the sched main program.

```
        subroutine versched(vernum,version) ! in
:schedlib:Sched/versched.f
      real intent(out) :: vernum
      character*(40) intent(out) :: version
    end subroutine versched
```

  It is also possible to guide the code generation of f2py by adding comments starting with Cf2py in the Fortran code. In this case the intermediate step of creating the signature file can be skipped.

- The Fortran version includes multiple files, which results in all functions and variables defined in those files being available in the main program. The Python version only imports a single module, all functions and COMMON blocks are available from that module.
- The GOTO logic has been replaced by a `while` loop.

Running the Verify test script (from the examples/ subdirectory) using the Fortran and Python version of the main program produces the same output files.

7.2. Reorganising Fortran data structures

Below is the Python code that uses the Fortran COMMON blocks exposed by the f2py generated module `schedlib`, to make objects that gather all data from the station catalogs on a per station basis. Most of the lines of code simply define which Fortran variables are to be reordered into the Python structures.

```
import schedlib as s

import numpy

class StationCatalog(object):
    """
    INTEGER         ISCHSTA(MAXCAT), MJDRATE(MAXCAT)
    CHARACTER       STATION(MAXCAT)*8
    CHARACTER       STCODE(MAXCAT)*3
    CHARACTER       STCODEU(MAXCAT)*3
    DOUBLE PRECISION XPOS(MAXCAT), YPOS(MAXCAT), ZPOS(MAXCAT)
    DOUBLE PRECISION DXPOS(MAXCAT), DYPOS(MAXCAT), DZPOS(MAXCAT)
    DOUBLE PRECISION ELEV(MAXCAT), LAT(MAXCAT), LONG(MAXCAT)
    CHARACTER       POSREF(MAXCAT)*80
    CHARACTER       CONTROL(MAXCAT)*4, MOUNT(MAXCAT)*5
    CHARACTER       DAR(MAXCAT)*5, RECORDER(MAXCAT)*6
    CHARACTER       DISK(MAXCAT)*6, MEDIADEF(MAXCAT)*6
    CHARACTER       TSCAL(MAXCAT)*4, DBBCVER(MAXCAT)*8
```

```
INTEGER        NBBC(MAXCAT), STNDRIV(MAXCAT), NHEADS(MAXCAT)
LOGICAL        VLBADAR(MAXCAT), USEONSRC(MAXCAT)
INTEGER        NHORIZ(MAXCAT)
REAL          HORAZ(200,MAXCAT), HOREL(200,MAXCAT)
INTEGER        NAXLIM(MAXCAT)
REAL          AX1LIM(6,MAXCAT), AX2LIM(6,MAXCAT)
REAL          AX1RATE(MAXCAT), AX2RATE(MAXCAT)
REAL          AX1ACC(2,MAXCAT), AX2ACC(2,MAXCAT)
REAL          TSETTLE(MAXCAT), MINSETUP(MAXCAT)
REAL          MAXSRCHR(MAXCAT), TLEVSET(MAXCAT)
REAL          ZALIM(MAXCAT), AXOFF(MAXCAT)
"""

class StationCatalogEntry(object):
   def __init__(self, **kwargs):
      for key, value in kwargs.items():
         setattr(self, key, value)

maxcat = s.schsta.ischsta.shape[0]
block_items = {
   s.schcst: [
      'control', 'dar','dbbcver', 'disk', 'mediadef', 'mount', 'posref',
      'recorder', 'station', 'stcode', 'stcodeu', 'tscal'],
   s.schsta: [
      'ax1acc', 'ax1lim', 'ax1rate', 'ax2acc', 'ax2lim', 'ax2rate',
      'axoff', 'dxpos', 'dypos', 'dzpos', 'elev', 'horaz', 'horel',
      'ischsta', 'lat', 'long_bn', 'maxsrchr', 'minsetup', 'mjdrate',
      'naxlim', 'nbbc', 'nheads', 'nhoriz', 'stndriv', 'tlevset',
      'tsettle', 'useonsrc', 'vlbadar', 'xpos', 'ypos', 'zalim', 'zpos']
   }

def read(self):
   # gather a copy (because .T returns a view) of all arrays in C-order
   arrays = {item: getattr(block, item).copy().T
           for block, items in self.block_items.items()
           for item in items}
   # create an entry for each index of the arrays
   self.entries = [
      self.StationCatalogEntry(
         **{key: value[i] for key, value in arrays.items()})
      for i in range(self.maxcat)]

def write(self):
   for block, items in self.block_items.items():
```

```
            for item in items:
                new_value = numpy.array([getattr(self.entries[i], item)
                                         for i in range(self.maxcat)],
                                        dtype=getattr(block, item).dtype)
                # f2py has a bug which doesn't allow setting arrays of strings
                # as a work-around, get a view and assign to that
                getattr(block, item).T[:] = new_value

if __name__ == "__main__":
    s.input()
    s.defaults()

    c = StationCatalog()
    c.read()

    c.entries[0].stcode = "Fo "

    c.write()
```

The class StationCatalog has two methods, read and write. read translates the COMMON blocks into easier to use Python structures (the StationCatalog.entries list). write writes the data back into the COMMON blocks, such that subsequent calls to Fortran code will use the changed value.

The code in the if __name__ == "__main__": block demonstrates that this read-write round trip works.

8. Conclusion

In this document we have outlined a way to start the re-factoring of the SCHED program. We show that creating an interface to Python using f2py is feasible, and have identified a number of features that will be tackled first. This will lay the foundations for a SCHED that is easier to modify, expand and maintain.

References

[1] http://www.swig.org/
[2] https://docs.scipy.org/doc/numpy/f2py/index.html
[3] http://giza.sourceforge.net/)
[4] http://www.vlbi.org/vex/
[5] https://safe.nrao.edu/wiki/bin/view/VLBA/Vex2
[6] https://docs.scipy.org/doc/numpy/index.html