GAVRT Solar Patrol
Observer's Guide

Tom Kuiper
with help from Lilibet

January 12, 2021

Contents

List of Figures

List of Tables

List of Code Snippets

Chapter 1

Quick Start

1.1 Quick Look

Code snippet 1 shows how to get a quick look at a day's data. For more details, see section 5.3.1 (page 18).

```
kuiper@kuiper:~$ cd '/usr/local/projects/SolarPatrol/apps/Reduction'
kuiper@kuiper:/usr/local/projects/SolarPatrol/apps/Reduction$ ipython --pylab
...
In [1]: run mapList.py
In [2]: sp = dbplotter.get_session_plotter(year=2019, doy=242); \
             sp.summary(save=True)
INFO:Data_Reduction.DSN.GAVRT.Mysql.plotter.DBPlotter.Session:
                                                2019/242 found 18 boresights
no usable boresights found
```

Snippet 1: Quick look at the data from an observing session.

Chapter 2

Overview

GAVRT Solar Patrol uses a monitor and control (M&C) paradigm in which the operator manages the antenna and the receiver more or less independently using two separate programs. Figure 2.1 shows the GAVRT M&C system. The operator has two graphical user interfaces (GUIs) provided by

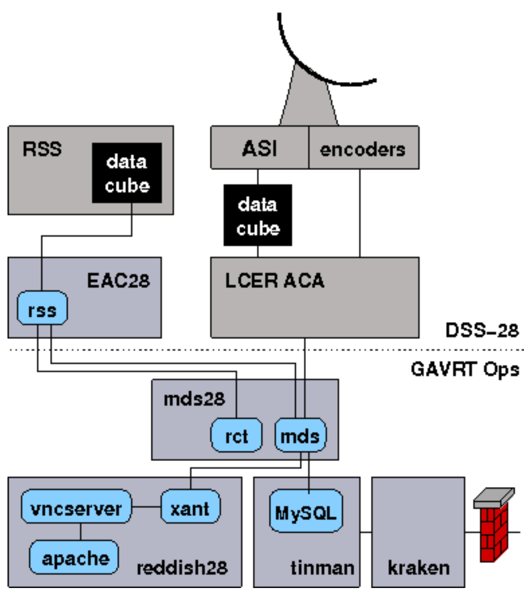

Figure 2.1: The GAVRT M&C system is distributed over the Lewis Center in Apple Valley (control center) and DSS-28 at Goldstone (antenna and receiver).

the program `rss` which manages the Receiver Subsystem, and the program `xant` which mainly manages the antenna but also interacts with `rss` to get readings from the voltage-to-frequency converters for antenna calibration.

There is available a software package which can "wrap" the elements of this system, along with other subsystems such as the digital signal processors, into one system managed by a central server which knows the state of the signals at each subsystem from where the signal enters the antenna to the backend subsystems which record the data [Kuiper and Shaff (2019)]. This may be implemented at a future date.

2.1 Antenna

Figure 2.2 shows the `xant` GUI which is used to monitor and control the antenna. Partially hidden is the display of program `xplot` which is a stripchart

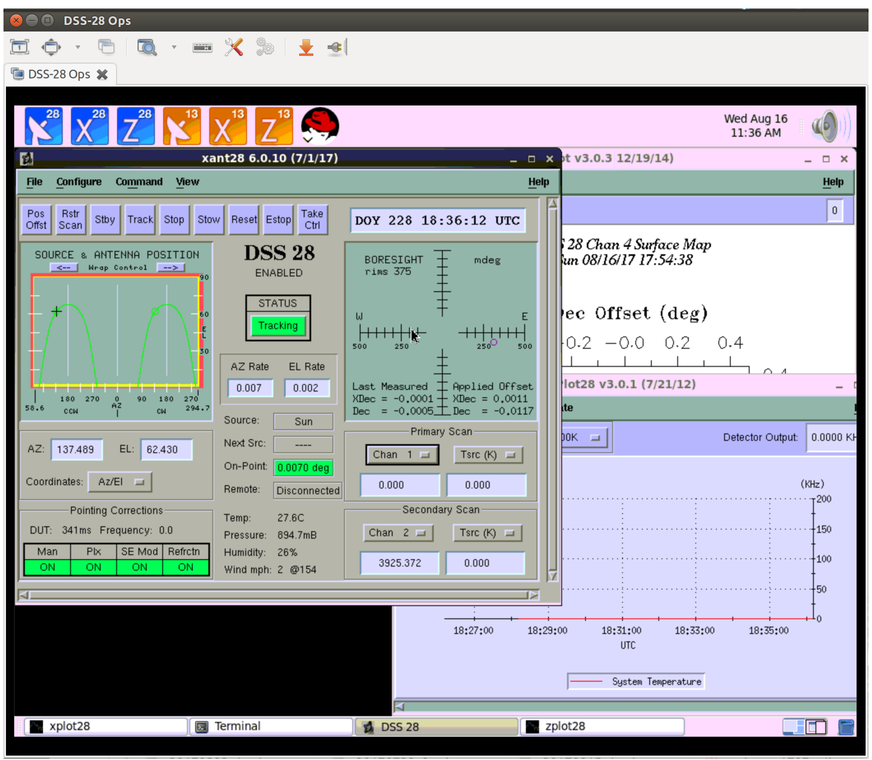

Figure 2.2: GUI for antenna M&C.

showing the system temperature T_{SYS} as a function of time in one or two selected channels. Behind that is the display of program `xmap` which shows a color contour map of the source. Program `xraster`, not shown, shows a 3D image of the source.

2.2 Receiver

Figure 2.3 shows the `rss` GUI which is used to monitor and control the receiver. The panel labeled “Converter 2” is a pop-up window used to con-

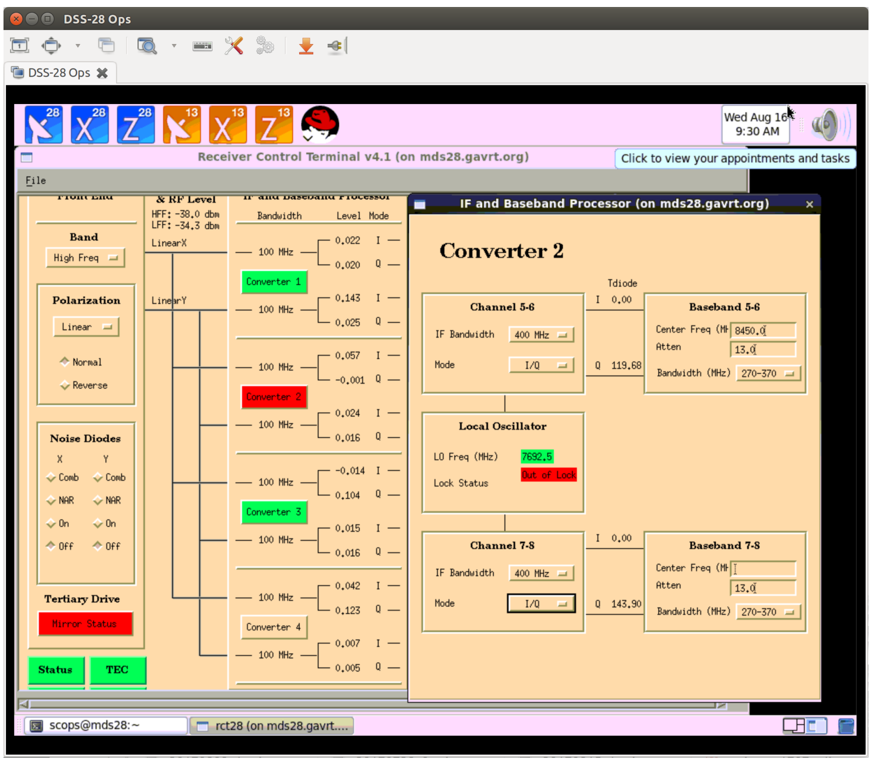

Figure 2.3: GUI for receiver M&C.

figure one of the four down-converters.

Schematics of the DSS-28 receiver are shown in Appendix D. Summarizing, the receiver has two RF inputs accepting the orthogonal linear outputs of the front end. The signal can optionally be combined in a quadrature hybrid to form two circular polarization. Each pair of signals (E-plane and

H-plane, or LCP and RCP) is split by a power divider and one copy of each is sent to one of four down-converter blocks, numbered 1-4. The polarized inputs are referred to as “a” and “b”. Each signal (of which there are now eight copies) are down-converted to baseband in a complex mixer which has in-phase and quadrature-phase outputs. The outputs are usually, but not necessarily, converted to lower and upper sidebands, giving a total of sixteen “channels”. This is summarized in Table 2.1

Table 2.1: Converter and Channel Names

RF Pol.	Down.Conv	Sideband	Channel
X/RCP	1a	U	1
		L	2
Y/LCP	1b	U	3
		L	4
...	...	...	...
X/RCP	4a	U	13
		L	14
Y/LCP	4b	U	15
		L	16

2.3 Software

The software uses the object-oriented programming features of Python[1]

> **Background** In object-oriented programming (OOP), a *class* is source code for a general object, such as “car”. Real objects, such as “Tesla”, are made by creating an *instance* of a class, called an *object*. A class, and therefore an object, has “attributes” (things with associated values, like `name`) and “methods” (functions that do things, like `close()`). These are appended by name to its object name. For example `datetime.strftime(`*format*`)` formats the value of `datetime` according to *format*. A *subclass* is derived from a parent class and has all the attributes and methods of that class, plus additional attributes and methods of its own.

[1] It is possible to use Python as a procedural language such as BASIC, C, or FORTRAN with normal functions. The plotting package supports this in a Matlab®-like fashion. With other packages or advanced `matplotlib` features it becomes a challenge.

The class which opens and interacts with the database is `DSS28db`. The class `DBPlotter` is a subclass of `DSS28db` which adds plotting methods. The class `DBPlotter` is described in Appendix C.4.1.

All the data from a specified year and day-of-year are contained in a class called `Session`. A session object is created by specifying the data, as in Snippet 2. The session has an attribute called `maps`, which are numbered

```
from Data_Reduction.DSN.GAVRT.Mysql.dss28db import Session
session = Session(2019, 88)
```

Snippet 2: Creating a `Session` object.

(*e.g.* `session.maps[130]`). Maps are objects of the `Map` class, which is described in Appendix C.2.4. The subclass `SessionPlotter` adds plotting methods to class `Session`.

Chapter 3

Observing Sessions

3.1 Establish a Session

The preferred method is to start a remote desktop client with the VNC protocol. It is also possible to download and execute a client-side Java script to run VNC in a browser but this invites incompatibility headaches between your browser and your version of Java.

The first step is to call GAVRT Ops at 1(760)946-5414 ext. 270 to enable remote access to the server `reddish28.gavrt.org`.

3.1.1 VNC Viewer

This will vary from desktop to destop (*e.g.* Gnome, Unity, KDE, etc.) but the basic idea is to search for a remote desktop client (under "Network" if a category is requested). The protocol for the connection will be VNC. The remote host and display is `reddish28.gavrt.org:1`. The password is `g4vrt4u`.

3.1.2 Browser

The browser must have Java enabled. It may be necessary to add an exception to allow a Java script from `reddish28.gavrt.org` to run. On a Linux system invoke `jcontrol` in a terminal window. It will open a new window. Under the Security tab, look for the Exceptions site list and edit it to allow `reddish28.gavrt.org`. Then restart the browser[1]

[1] In truth, I could not get this to work for either Google Chrome or Iceweasel (=Firefox) under Debian 7 Linux.

3.1.3 Python Connection

Receiver Configurations

To get the current receiver configuration check the database plotter object's `receiver` attribute as shown in Snippet 3

```
In [1]: from Data_Reduction.DSN.GAVRT.Mysql.dss28db import DSS28db
In [2]: source_db = DSS28db(name='gavrt_sources')
In [3]: dbplotter.receiver
Out[3]: {'if_bw': { 2: 400.0,  4: 400.0,  6: 400.0,  8: 400.0,
                   10: 400.0, 12: 400.0, 14: 400.0, 16: 400.0},
         'if_mode': { 2: 'ul',  4: 'ul',  6: 'ul', 8:  'ul',
                     10: 'ul', 12: 'ul', 14: 'ul', 16: 'ul'},
         'pol': { 2: 'linx',  4: 'liny',  6: 'linx',  8: 'liny',
                 10: 'linx', 12: 'liny', 14: 'linx', 16: 'liny'},
         'sky_freq': { 2: 8450.0,  4: 8450.0,  6: 8450.0,  8: 8450.0,
                      10: 8450.0, 12: 8450.0, 14: 8450.0, 16: 8450.0},
         'utc': { 2: datetime.timedelta(0, 49625),  4: datetime.timedelta(0, 49625),
                  6: datetime.timedelta(0, 49625),  8: datetime.timedelta(0, 49625),
                 10: datetime.timedelta(0, 49625), 12: datetime.timedelta(0, 49625),
                 14: datetime.timedelta(0, 49625), 16: datetime.timedelta(0, 49625)}}
```

Snippet 3: Checking the current receiver configuration. (The output has been reformatted to save space.)

Chapter 4

Solar Observations

4.1 Receiver Configuration

Table 4.1 shows the initial configuration for solar observations. Depending

Table 4.1: Configuration Checklist for Solar Observations

Source	Sun	Venus
Parameter	**Value**	
Absorber plate	in	out
IF bandwidth	100 MHz	
Polarization	LCP, RCP	
Frequency priority	14, 2.8, 8.45, 4.9	14, 2.8, 8.45, 4.9
Attenuator (without plate)	20, 30, 30, 30	0, 13, 7, 7
Attenuator (with plate)	10, 20, 20, 20	0, 3, 0, 0

on the number of working channels, frequencies should be selected in the order shown with both polarizations.

The channels should be configured to write to the `tlog` table.

A checklist, shown in Table 4.1 should be filled out for every source change.

4.2 Checking Progress

A `SessionPlotter` object let's us see what maps and boresights have been done. Code snippet 4 shows how. Step [3] is slow because it searches the database for all the boresight and map data for that day.

Table 4.2: Checklist for Each Source

Channel	2	4	6	8	10	12	14	16
Polarization	RCP	LCP	RCP	LCP	RCP	LCP	RCP	LCP
`tlog` (on)								
Frequency (GHz)								
Attenuation (dB)								
Plate (in/out)								

```
In [1]: from Data_Reduction.DSN.GAVRT.Mysql.plotter import DBPlotter
In [2]: dbplotter = DBPlotter()
In [3]: sp = dbplotter.get_session_plotter(year=2019, doy=242)
In [4]: sp.summary(save=True)
In [5]: sp.maps.keys()
Out[5]: [200, 199]
In [6]: sp.boresights.keys()
Out[6]: [35201, 35202, 35203, 35204, 35205, 35206, 35207, 35208, 35209, 35210,
         35211, 35212, 35213, 35214, 35215, 35216, 35217, 35218]
In [7]: sp.maps[199].get_active_channels()
Out[7]: array([2, 4])
In [8]: sp.maps[200].get_active_channels()
Out[8]: array([2, 4])
In [9]: sp.boresights[35215].channels
Out[9]: array([2, 4])
```

Snippet 4: Checking progress during an observation.

Step [3] should be repeated as needed to refresh the data. `maps` and `boresights` are those present when step [3] is done. `get_active_channels()` reports on the channels present in the `tlog` table. We cannot calibrate maps if there are no boresights using the same channels.

Chapter 5

Data Reduction

All the examples here use iPython with the `-pylab` module.

5.1 Databases

The server has these databases: `dss28_eac` is the default database. It is described at `http://gsc.lewiscenter.org/data_info/dss28_eac.php`. There is a database for spectrometer data called `dss28_spec`. It is described at `http://gsc.lewiscenter.org/data_info/dss28_spec.php`. Radio source information is kept in `gavrt_sources`. It does not have a web page describing it.

The source database can be accessed as shown in Snippet 5. This database can then be searched for calibration sources that fit a specific set of requirements, such as sky position and flux density.

5.1.1 Establish a Connection for Data Reduction

Snippet 6 shows how create a session for data reduction using the GAVRT MySQL database. The `DBPlotter` inherits all the attributes and methods of the `DSS28db` class and adds plotting using `matplotlib`.

5.2 Public Tables

Appendix A (page 22) gives the details of the database tables. If the manual isn't handy, code snippet 7 shows how to ask the database for the tables. To find out more about a table, get a list of its columns. Snippet 5 shows an example.

```
In [1]: from Data_Reduction.DSN.GAVRT.Mysql.dss28db import DSS28db
In [2]: source_db = DSS28db(name='gavrt_sources')
In [3]: source_db.get_public_tables()
Out[3]: (('catalog',), ('class',), ('source',))
In [4]: source_db.get_data_index()
Out[4]:
{'catalog': (('catalog_id', 'int(11)', 'NO', 'PRI', None, 'auto_increment'),
             ('name', 'varchar(32)', 'NO', '', None, '')),
'class': (('class_id', 'int(11)', 'NO', 'PRI', None, 'auto_increment'),
          ('name', 'varchar(32)', 'NO', '', None, ''),
          ('description', 'varchar(128)', 'YES', '', None, '')),
'source': (('source_id', 'int(11)', 'NO', 'PRI', None, 'auto_increment'),
           ('name', 'varchar(16)', 'NO', '', None, ''),
           ('RA', 'float', 'NO', '', None, ''),
           ('Dec', 'float', 'NO', '', None, ''),
           ('size_dec', 'int(11)', 'NO', '', None, ''),
           ('size_xdec', 'int(11)', 'NO', '', None, ''),
           ('catalog_id', 'int(11)', 'NO', '', None, ''),
           ('class_id', 'int(11)', 'NO', '', None, ''),
           ('reference', 'varchar(8)', 'NO', '', None, ''),sec:overview
           ('aka', 'varchar(16)', 'YES', '', None, ''))}
```

Snippet 5: Querying `gavrt_sources` for its tables.

```
In [1]: from Data_Reduction.DSN.GAVRT.Mysql.plotter import DBPlotter
In [2]: dbplotter = DBPlotter()
...
In [6]: dbplotter.close()
```

Snippet 6: Opening and closing a data reduction session using the GAVRT MySQL database.

```
In [9]: dbplotter.get_public_tables()
Out[9]: (('angles',),      ('chan_cfg',),     ('conv_cfg',),   ('fiber_cfg',),
        ('five_point',),   ('pointing_cfg',), ('raster',),     ('raster_cfg',),
        ('rf_cfg',),       ('rss_cfg',),      ('xpwr_cfg',),   ('xscan',),
        ('zplot',),        ('zplot_cfg',))
```

Snippet 7: Example of querying a database for its tables.

5.3 Data Analysis

The strategy here is to give the user the full power of Python without contraint imposed by a program with programmer defined options. Once the basic Python concepts have been learned this is by far the most comfortable

```
In [19]: dbplotter.get_columns('tlog')
Out[19]: (('tlog_id',    'int(11)',       'NO', 'PRI', None, 'auto_increment'),
          ('rss_cfg_id', 'int(11)',       'NO', '',    None, ''),
          ('year',       'int(4)',        'NO', '',    None, ''),
          ('doy',        'int(3)',        'NO', '',    None, ''),
          ('utc',        'time',          'NO', '',    None, ''),
          ('epoch',      'decimal(16,6)', 'NO', '',    None, ''),
          ('chan',       'int(2)',        'NO', '',    None, ''),
          ('top',        'decimal(7,4)',  'NO', '',    None, ''),
          ('integ',      'decimal(5,4)',  'NO', '',    None, ''),
          ('az',         'decimal(7,4)',  'NO', '',    None, ''),
          ('el',         'decimal(6,4)',  'NO', '',    None, ''),
          ('diode',      'tinyint(1)',    'NO', '',    None, ''),
          ('level',      'decimal(3,1)',  'NO', '',    None, ''),
          ('cryo',       'decimal(6,3)',  'NO', '',    None, ''))
```

Snippet 8: List of columns in the `tlog` table.

way to work with the data.

5.3.1 Get an Overview of the Observations

A `DBPlotter` object is a `DSS28db` object with plotting capability. Code snippet 4 (page 15) shows how to get the maps and boresights in a table. It produces two files in the session directory (code snippet 9), which list the

```
In [3]: sp.session_dir
Out[3]: '/usr/local/projects/SolarPatrol/Observations/dss28/2019/242/'
In [7]: import os
In [8]: os.listdir(sp.session_dir)
Out[8]: ['boresight-35214.png', 'boresight-35215.png', 'boresight-35216.png',
         'boresight-35217.png', 'maps.txt',            'xscans.txt',
         'map0199.png',         'map0200.png']
```

Snippet 9: Getting the session directory.

boresights and the maps. The boresights and maps are also shown graphically, one figure for each boresight scan and map. These are uncalibrated VFC counts. For the maps, each figure has all the channels for that map.

5.3.2 Time Series

The `tlog` table has VFC counts as a function of time, as well as antenna angles and other variables. Code snippet 8 shows the columns of the `tlog` table. The advantage of class `DBPlotter` over its superclass `DSS28db` is that you plot any column of the table as a time series. Code snippet 10 shows an example of that.

```
In [20]: from DatesTimes import UnixTime_to_MPL
In [25]: Tsys_data = {2: dbplotter.get_Tsys(2, 1567198299.1872621,
                                           1567198456.2302589),
                      4: dbplotter.get_Tsys(4, 1567198299.1872621,
                                           1567198456.2302589)}
In [26]: plot_date(UnixTime_to_MPL(Tsys_data[2]['epoch']), Tsys_data[2]['top'], '-',
                   label="Ch. 2")
In [27]: plot_date(UnixTime_to_MPL(Tsys_data[4]['epoch']), Tsys_data[4]['top'], '-',
                   label="Ch. 4")
```

Snippet 10: Plotting T_{op} as a function of time.

5.3.3 Boresights

Code snippet 11 shows how to inspect boresight data. Getting boresight data

```
In [1]: from Data_Reduction.DSN.GAVRT.Mysql.plotter import DBPlotter
In [2]: dbplotter = DBPlotter()
In [3]: boresight_data = dbplotter.extract_boresight_data(2019,87)
In [4]: boresight_data.keys()
Out[4]: ['utc',       'el',     'rx',        'xpwr_cfg_id', 'chan',
'source',    'epoch',  'xscan_id',  'source_id',   'tsrc',
'az',        'axis']

In [7]: len(boresight_data['utc'])
Out[7]: 32
In [8]: boresight_data['rx'][0].keys()
Out[8]: ['sky_freq', 'utc', 'pol', 'if_bw', 'if_mode']

In [17]: boresight_data['source']
Out[17]: ['J1800+7828']
```

Snippet 11: Example to code to inspect boresight data

(step [3]) takes a very long time because it involves look-up in many tables managed by the local host, so involves a lot of Internet traffic. Certainly time to have coffee, if not lunch.

5.3.4 Maps

There is a quick way to get a session plotter without having to remember the class and import path. It's shown in code snippet 12. The `save` argument will put summary plots and text in a directory `Observations/dss28/2019/162`.

The maps for this session can be found in attribute `maps` which is a `dict` keyed on map number. More useful for analysis are map plotters which

```
kuiper@kuiper:~$ cd '/usr/local/projects/SolarPatrol/apps/Reduction'
kuiper@kuiper:/usr/local/projects/SolarPatrol/apps/Reduction$ ipython --pylab
...
In [1]: run mapList.py
Suggestions::

* To set up a session:
sp = dbplotter.get_session_plotter(year=2019, doy=98)
* To load the boresight data and report:
sp.make_bs_dir()
* To load boresight metadata only:
xpwr_data, boresights = sp.get_boresights()
* To load boresight metadata and data:
bs_data, channels = sp.get_boresight_data()
* To reduce boresights and maps and put results in session directory:
session_summary(sp)

To do everything at once::
sp = dbplotter.get_session_plotter(year=2019, doy=98); session_summary(sp, save=True)

Don't forget:
dbplotter.close()
when finished

In [2]: sp = dbplotter.get_session_plotter(year=2019, doy=162); sp.summary(save=True)
no usable boresights found
```

Snippet 12: Program `mapList` can be used to get a session summary.

are a subclass of the maps. Snippet 13 shows how to use them. A map

```
    In [5]: mpl = sp.get_map_plotters()
    In [6]: mpl.keys()
    Out[6]: [193, 194, 195, 196, 197, 198]
    In [7]: mpl193 = mpl[193]
    In [8]: mapdata = mpl193.maps_from_tlogs()
    In [9]: centered = mpl[193].center_map()
    In [10]: mpl193.contours(2)
    In [13]: from pylab import *
    In [14]: show()
    In [15]: mpl193.contours(4); show()
    In [16]: sp.show_images()
```

Snippet 13: Producing map images.

as displayed on the operator's console by program `xant` is stored in the database in the table `raster`. Rasters are defined by positions that are offset in cross-elevation and elevation from the source position and by the VFC counts from one receiver channel. The parameters for a raster are in table `raster_cfg`.

Because the map data in `raster` are for only one channels, we need to get the data for other channels from the `tlog` table. In fact, we don't use the data in `raster` at all but use the raster to define the start and end time. We then get the map data from table `tlog` between those to times. That happens in step [8] above. B.t.w., step [7] is not necessary. Step [8] could just as well have been `mpl[193].maps_from_tlogs()`. However, you can't `Tab`-complete a `dict` element to get attributes and methods.

The `tlog` table does not contain position offsets. However, it does have azimuth, elevation, and time. With suitable coordinate conversions, position offsets from the source can be calculated. That is what `center_map()` [step 9] does. `center_map()` also calls method `regrid()` which interpolates the data onto a regular grid using a specified step size or the default from table `raster_cfg`. Then the maps can be plotted (steps [10] and [15]) for all channels.

The result of step [8] can be used for manipulating the maps. Snippet 14 shows the raw map data. These data are stored in an attribute called

```
In [20]: mapdata = sp.maps[199].maps_from_tlogs()
In [21]: mapdata.keys()
Out[21]: ['elevation', 'UNIXtime', 'pol',        'MPL_datenum', 'RA',
'azimuth',   'freq',     'VFC_counts', 'declination']
In [22]: mapdata['freq']
Out[22]: {2: array([ 3100.]), 4: array([ 3100.])}
In [25]: mapdata['VFC_counts'].keys()
Out[25]: [2, 4]
```

Snippet 14: Data returned from `maps_from_tlogs()`.

`map_data`. Centering the map provides additional data, shown in snippet 15. So thus we have a map for each active channel.

```
In [28]: centered = sp.maps[199].center_map()
In [32]: sp.maps[199].map_data.keys()
Out[32]: ['elevation',  'UNIXtime', 'pol',  'MPL_datenum', 'dec_offset',
          'RA',         'azimuth',  'freq', 'VFC_counts',  'xdec_offset',
          'declination']
```

Snippet 15: Centering the map produces position offsets.

Appendix A

Database Tables

The databases used by Solar Patrol are on a MySQL server[1] at the Lewis Center.

The server has these three databases:

Name	Data
`dss28_eac`	all the data from all the observations
`dss28_spec`	data from the digital spectrometers
`gavrt_sources`	radio sources used for observation and calibration

The main databases used by Solar Patrol are `dss28_eac`, which contains all the data about telecope and receiver operations, and `gavrt_sources`. Figure A.1 The diagram can help structure queries.

Database 'gavrt_sources' has these tables:

Name	Contents
`catalog`	
`class`	
`source`	source_id, catalog_id, class_id, name, RA, Dec, size_dec, size_xdec, reference, aka

Database `dss28_eac` has the tables shown in Table A.1. Most columns have self-explanatory names but a few need some description.

`cal_src_id` identifies the source which has a flux `cal_flux` used the scale the data for the source `source_id`.

[1] `http://gsc.lewiscenter.org/data_info/dss28_eac.php`

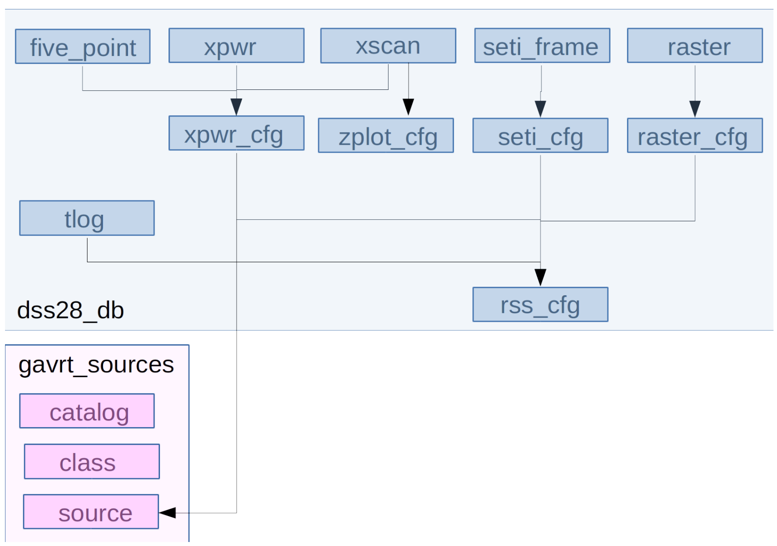

Figure A.1: The relations between the tables used by Solar Patrol.

Table A.1: Columns in the tables of database `dss28_eac`.

Name	Contents
`angles`	angles_id, year, doy, utc, epoch, az, el, status
`chan_cfg`	chan_cfg_id, year, doy, utc, epoch, chan, center_freq, tdiode
`conv_cfg`	conv_cfg_id, year, doy, utc, epoch, converter, mode_a, ifbw_a, bbbw_a, atten_a, mode_b, ifbw_b, bbbw_b, atten_b, lock_status
`fiber_cfg`	fiber_cfg_id, year, doy, utc, epoch, fiber, chan
`five_point`	five_point_id, xpwr_cfg_id, year, doy, utc, epoch, source_id, chan, tsrc, azel, ha, dec, xdec_off, dec_off
`pointing_cfg`	pointing_cfg_id, year, doy, utc, epoch, man, plx, semod, refrctn, delut, model
`raster`	raster_id, raster_cfg_id, year, doy, utc, epoch, xdecoff, decoff, ha, dec, tsrc
`raster_cfg`	raster_cfg_id, rss_cfg_id, year, doy, utc, source_id, chan, freq, rate, step
`rf_cfg`	rf_cfg_id, year, doy, utc, epoch, feed, diode_x, diode_y, pol, transfer
`rss_cfg`	rss_cfg_id, year, doy, utc, epoch, chan, sky_freq, feed, pol, nd, if_mode, if_bw, bb_bw, fiber_chan
`seti_cfg`	seti_cfg_id, rss_cfg_id, year, doy, utc, epoch, frame_name, chan, freq, school_id, comment
`seti_frame`	frame_id, seti_cfg_id, year, doy, utc, epoch, glong, glat, long_err, lat_err
`tlog`	tlog_id, rss_cfg_id, year, doy, utc, epoch, chan, top, integ, ax, el, diode, level, cryo
`weather`	weather_id, datetime, pressure, temp, humidity, wind_speed, wind_dir
`xpwr`	xpwr_id, xpwr_cfg_id, year, doy, utc, epoch, tsys, az, el, ha, dec, offset
`xpwr_cfg`	xpwr_cfg_id, rss_cfg_id, source_id, cal_src_id, year, doy, utc, epoch, axis, chan, cal_flux
`xscan`	xscan_id, xpwr_cfg_id, year, doy, utc, epoch, tsrc, stdev, bl_stdev, az, az_offset, el, el_offset, ha, dec, offset, bw, corr
`zplot`	zplot_id, zplot_cfg_id, offset, tsrc
`zplot_cfg`	zplot_cfg_id, rss_cfg_id, year, doy, utc, epoch, source_id, axis, chan

Appendix B

Flux Calibration

The primary VLA flux calibrators[1] are shown in Table B. The source 3C286

Table B.1: Flux densities (Jy) of Standard Calibrators (January 2012)

Source			Frequency (GHz)				
			1.465	2.565	4.885	8.435	14.965
3C48	=	J0137+3309	15.56	9.80	5.39	3.14	1.77
3C138	=	J0521+1638	8.71	6.17	4.02	2.78	1.89
3C147	=	J0542+4951	21.85	13.75	7.59	4.49	2.59
3C286	=	J1331+3030	14.90	10.03	7.34	5.09	3.39
3C295	=	J1411+5212	22.15	12.95	6.41	3.34	1.62
NGC7027			1.62	3.59	5.38	5.79	5.62

(J1331+3030) is known to be non-variable, and has thus been adopted as the prime flux density calibrator source for the VLA. The adopted polynomial expression for the spectral flux density for 3C286 is[2]:

$$\log(S) = 1.2515 - 0.4605\log(f) - 0.1715\log^2(f) + 0.0336\log^3(f) \quad \text{(B.1)}$$

where S is the flux density in Jy, and f is the frequency in GHz.

The sources 3C48, 3C138, and 3C147 are all slowly variable. Table B shows the flux values four years later[3]. The polynomial expression for the spectral flux density for 3C286 in 2016 is:

$$\log(S) = 1.2481 - 0.4507\log(f) - 0.1798\log^2(f) + 0.0357\log^3(f). \quad \text{(B.2)}$$

[1] `https://science.nrao.edu/facilities/vla/docs/manuals/oss2013B/performance/fdscale`
[2] `https://science.lbo.us/facilities/vla/docs/manuals/oss/performance/fdscale`
[3] `https://science.nrao.edu/facilities/vla/docs/manuals/oss/performance/fdscale`

Table B.2: Flux densities (Jy) of Standard Calibrators (January 2016)

Source			Frequency (GHz)				
			1.50	3.00	6.00	10.00	15.00
3C48*	=	J0137+3309	15.40	8.44	4.42	2.68	1.79
3C138	=	J0521+1638	08.25	5.44	3.39	2.33	1.72
3C147	=	J0542+4951	21.00	12.00	6.45	3.99	2.73
3C196	=	J0813+4813	13.60	6.98	3.38	1.91	1.20
3C286	=	J1331+3030	14.60	9.91	6.39	4.50	3.37
3C295	=	J1411+5212	21.20	11.00	5.06	2.70	1.60

*The flux density scale calibrator 3C48 has been undergoing a flare since January 2018 or so

Appendix C

Data Reduction Tool Documentation

C.1 Radio Astronomy Software Tools

The repository collection "Single Dish Rasdio Astronomy Software Tools"[1] describes the modules used to reduce Solar Patrol data. The principal Python module `GAVRT` in in the package `Data_Reduction`[2]. Figure C.1 shows how it depends on other packages in the collection.

C.2 Package `Data_Reduction`

The base module of this package provides the base classes for the package

C.2.1 Class `Observation`

`Data_Reduction.Observation` is the superclass which provides the data structure and methods common to all observations. Conceptually, the data structure `data` is a table, though it is implemented as a Python `dict`.

Attributes

These are the public attributes of the class:

aliases [`list`] are `data` keys used to replace those in original data, in order to have a common format.

[1] `https://sdrast.github.io/`

[2] `https://github.com/SDRAST/Data_Reduction/`

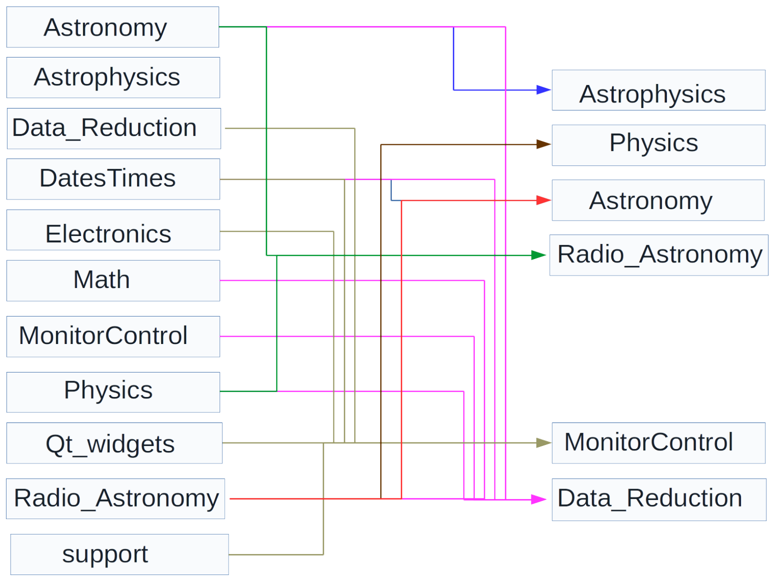

Figure C.1: Interdependency of packages in the collection Radio Astronomy Software Tools.

channel [`Channel` instance] is a signal paths; *e.g.*, for different frequencies and polarizations.
data [`dict`] are the original observation data, *e.g.*, read from file or database.
DOY [`int`] is day of year of observation (1-366).
end [`float`] is the UNIX time (Python `time.time`) at the end of the observation.
latitude [`float`] is the telescope latitude, also an attribute of `obs`.
logger [`logging.Logger` instance] is used to log program events.
longitude [`float`] telescope longitude, also an attribute of `obs`.
name [`str`] is assigned by the user, but defaults to YEAR/DOY.
numdata [`int`] is number of data samples.
obs [`Astronomy.DSN_coordinates.DSS` instance] is the telescope.
session [`Session` instance] is a set of observations, parent to Observation.
session_path [`str`] is the directory for the session's files.
start [`float`] is UNIX time at the beginning of the observation.
year [`int`] is the year at the start of the observation.

Data Structure Keys

The data structure must have these keys (table column names) or version (in `aliases` that can map into them:

unixtime [`float`] the UNIX time in seconds,
chan_name [`str`] the channel name,
integr [`float`] integration time (exposure) in seconds,
azel [`(float,float)` tuple] azimuth and elevation in decimal degrees, and
power [`float`] the power level, if only a single channel (`i.e.` not spectra).

The following keys are optional, but they are reserved names:

diode [`float`] is noise diode power in K.
level [`float`] is an unidentified quantity in GAVRT database "tlog" table.
cryotem [`float`] is the cryostat temperature in K.
windspeed [`float`] is the wind speed km/hr.
winddir [`float`] is the wind direction, azimuth in degrees.
ambtemp [`float`] is the outside air temperature in degrees C.
pressure [`float)` is the atmospheric pressure in mbar.

Data Format Alternatives

The original data come from many different telescopes and so the software is versatile in what it accepts. Accordingly, all the names here are reserved keywords so the software will recognize and convert data as needed.

Time may also be specified as either year, DOY, UTC (`int`, `int`, colon-delimited `str`) or a time string such as `2020/06/14/14:22:21.00`. To facilitate using the `matplotlib` package, time is also retained in the form of `datenum` values.

Power may take the form of `Top` or `Tsys`, or `vfc_counts`.

Channel may be designated with an `int`, as in "IF number", with a lookup table.

Coordinates can take many different forms which are implicitly the same as `az, el` tuples. For example, `az` and `el` may be in separate columns. Other possible specifications are:

radec [`(float,float)`] precessed right ascension in decimal hours and precessed declination in decimal degrees;

radec1950 [`(float,float)`] mean right ascension in decimal hours and mean declination in decimal degrees at epoch of observation;

radec2000 [`(float,float)`] mean right ascension in decimal hours and mean declination at equinox in decimal degrees.

The columns of tuples may also be given as individual columns.

C.2.2 Class `DataGetterMixin`

This class is for getting data from a CSV file.

C.2.3 Class `GriddingMixin`

This is the class for all the data and methods associated with a raster scan map. It is expected that the parent class is a subclass of "Observation" already by virtue of it being a superclass of subclass which inherits these methods.

C.2.4 Class `Map(Observation, GriddingMixin)`

General `Map` class without special features for GAVRT and Malargue. Most of the methods are mixed in to avoid conflicting with subclasses.

C.2.5 Class `Recording`

Class for raw IF voltage data. This is typically the contents of a data file transcribed into a standard format. It may be the data of one `Observation` object, or data for multiple `Observation` objects, or contain part of the data for an`Observation` object.

C.2.6 Class `Session`

This is the base class for an observing session on a given year and DOY. A session usually refers to a telescope, date and project. This will normally define a path to the session directory.

Public Attributes

doy [`int`] is the day of year for the session.

logger [`logging.Logger`] - logging.Logger object

parent may be a data reduction session for multiple observing sessions.
year [`int`]
doy [`int`]
project [`str`]
session_dir [`str`] is the path to results from this session.

C.2.7 Class `Spectrum (Observation)`

This is a class for spectra.

C.3 Module `GAVRT`

The module called `DR` here is the base `Data_Reduction` module desribed anbove.

C.3.1 Class `Observation(DR.Observation, DR.GriddingMixin)`

C.3.2 Class `Map(Observation)`

Attributes

cfg raster configuration
cfg_id entry in the raster configuration table
channels list of channels which took `tlog data`
end UNIX time at end of map
logger logging.Logger object
map_data dict of data from `tlog` table;
name map identifier
raster_data data from the raster table
regrid computes map data onto a rectangular grid
rss_cfg receiver configuration
session observing session to which this map belongs
start UNIX time at start of map

Methods

center_map converts map coordinates to be relative to Sun
get_active_channels returns channels which took tlog data during this map
get_map_config returns a dict with the raster map configuration
get_raster_data gets the data for a raster scan map used for Zplot

get__raster__keys returns rasters associated with a given configuration
maps__from__tlogs re-organizes tlog table data into map form

C.3.3 Class `BoresightScan(Observation)`

Attributes

This is the class for a single scan during a boresight. It inherits from class `Observation`. It has these public attributes.

`axis` (str) direction of the scan.
`cal_flux` (float) flux of the calibrator source.
`cal_src` source used to set the flux scale.
`chan` channel used for the boresight by EAC program 'xant'.
`data` scan data ['el', 'az', 'tsys', 'epoch', 'ha', 'dec'] from 'xpwr' table
`diode` state of the noise diode.
`epoch` (float) UNIX time start of scan (not the time of the first data point).
`freq` (float) channel frequency in MHz.
`IFbw` (float) IF band width in MHz.
`IFmode` (str) IF phasing.
`logger` (logging.Logger) object.
`log_data` data from 'tlog' table.
`name` identifier string based on `xpwr_cfg_id`.
`pol` channel polarization.
`session` parent Session object.
`source` name of scanned source.

Methods

Method resolution order:
    BoresightScan
    Observation
    `__builtin__.object`
Methods defined here:

`__init__(parent, xpwr_cfg_id)` initialize the class.
    parent   'self' of the calling method (Session object)
    xpwr__cfg__id   (int) row identifier in table 'xpwr__cfg'

C.3.4 Class `Session(DR.Session`

C.3.5 Class `DSS28db(mysql.BaseDB`

Inherits from `BaseDB`

C.4 module Data_Reduction.DSN.GAVRT.Mysql.plotter

C.4.1 Class `DBPlotter`

Attributes

Public attributes:

logger logging.Logger object

Attributes inherited from DSS28db:

receiver receivers which provide data
sessions dict of sessions obtained with 'get_session'

Methods

Method resolution order:

- DBPlotter
- Data_Reduction.DSN.GAVRT.Mysql.dss28db.DSS28db
- Data_Reduction.DSN.GAVRT.Mysql.BaseDB

Methods defined here:

___init___(self)
get_session_plotter(self, year, doy) get IDs for an observing session

Methods inherited from Data_Reduction.DSN.GAVRT.Mysql.dss28db.DSS28db:

extract_boresight_data(self, year, doy) Get the metadata for the boresights on the designated day.

> **year** (int) year of observation
> **doy** (int) day of year
>
> The boresights are extracted from table 'xscan'. Missing 'el' data are obtained from table 'xpwr'. The source, scan axis and channel are obtained from table 'xpwr_cfg'. The receiver data are obtained from table 'rss_cfg'.

Returns a dictionary like this::

'utc'	list of datetpime.timedelta
'epoch'	list of float
'az'	list of float
'el'	list of value
'chan'	list of int
'tsrc'	list of float
'axis'	list of str
'source'	list of str
'xpwr_cfg_id	list of int'
'xscan_id'	list of int
'source_id'	list of int
'rx'	list of dict

An 'rx' dict looks like this::

2	'if_bw'	float
	'if_mode'	str
	'pol'	str
	'sky_freq'	float
	'utc'	datetime.timedelta
4	...	
	....	
16	...	

get_Tsys(chan, start, stop) Get system temperatures from tlog

start	(float) UNIXtime at start of selection
stop	(float) UNIXtime at end of selection

get_receiver_data (year, doy, time, columns) Get the receiver state at a given time.

db	(Mysql.BaseDB) database
year	(int) year of observation
doy	(int) day of year
time	(datetime.timedelta) UTC for the requested receiver state
columns	(list of str) data items to be returned

This creates a dictionary keyed with channel number and returns a dictionary of the receiver configuration, keyed with specified in the columns, that was in effect at the given time.

Notes (Logic) The challenge here is to get the latest configuration data for each channel at or prior to the specified time. That channel may have been consubsectiond on the same day or a prior day. The method we'll use is to find the ID of last configuration change and assume that the IDs are sequential in date/time.

get_session(self, year, doy) Get IDs for an observing session

Methods inherited from Data_Reduction.DSN.GAVRT.Mysql.BaseDB:

checkDB() Reconnects to the database if the connection has been lost.

close() Close a connection.

commit() Commits the most recent database transaction.

connect() Make a connection to the database. Creates a cursor object. Automatically invoked when an instance is created; can be called again if the connection is closed but the database object persists

cursor() Creates a database cursor object; same as BaseDB.c but this is better because it handles disconnected a database.

get(*args) Executes a query of the database.

args	query to be executed
return	record (dict)

getLastId(table) (int) ID of the last record.

table	(str) the name of the table
return	ID (int)

getLastRecord(table) Returns the last record as a dictionary.

table	(str) name of the table
return	(dict)

getRecordById(table, rec_id Get the record with the given ID.

table	(str) table name
id	(int) row ID
return]	(dict)

get_as_dict(*args, **kwargs) Executes a query of the database and returns the result as a dict.

At present, only keyword `asfloat: True` is recognized and is the default if not given. It will convert to float any values for which it is possible.

If the query returns multiple rows, each value associated with a keyword will be a list. If nothing was found, an empty dictionary is returned.

db	database connection object
args	query to be executed
kwargs	a dictionary with keyword arguments
return	the record as a dictionary

get_columns(table) Returns information about the columns of a table.

get_data_index()

get_public_tables() List the table names in the database.

return	tuple of tuples of str

get_rows_by_date(table, columns, year, doy) Gets data from a table.

table	(str) table name
columns	(list of str) list of columns to be selected
year	(int)
doy	(int) day of year
return	(dict of numpy arrays) keyed on column name

get_rows_by_time(table, columns, year, doy, utcs) Queries a table for quantities in columns at designated times.

This loops over a list of UTs. For each, it takes the first row it finds matching the date and time. So it has an effective resolution of one second.

table	(str) table name
columns	(list of str) list of columns to be selected
year	(int)
doy	(int) day of year
utcs	(list of int) UNIX times (seconds since the epoch) to be selected; the first occurrence is used
return	dict of numpy arrays keyed on column name

report_table(table, columns) Reports on the columns in a table. Response has keys: Extra, Default, Field, Key, Null, Type

table	(str) table name
columns	(list of str) list of column names
return	result of query

send_query(query) Send a MySQL query

crsr	cursor object for a database
query	(str) MySQL query
return	(str) query response

C.5 Module `Data_Reduction.boresight_fitter`

C.5.1 Class `ScanFitter`

This provides an object to fit a scan in one direction to a baseline and a Gaussian.

Methods

Methods defined here:

`__init__(scan)` Initiate a scan fitter

`fit_gaussian(self, beam_limit=2.5)` Extract the appropriate data.

For raster scans, `xdec` means that the cross-declination stays fixed while the antenna moves up and down; `dec` means that declination stays fixed while the antenna moves left and right.

The Gaussian is assumed to fit the inner 2×`beam_limit` (default: 2.5) beam widths of the data. The baseline is the rest of the data, although the lower baseline includes at least `data[:5]` and the upper baseline includes `data[-5]`.

Attributes

Public attributes:

`atten` (float) receiver channel attenuation for this scan.
`baseline_pars` (nparray) polynomial parameters for baseline.
`calibrator` (Astronomy.Ephem.calibrator) calibrator source.
`data` (nparray) VFC count.
`ddecs` (nparray) declination offsets.
`direction` (str) scan axis.
`dxdecs` (nparray) cross-declination offsets.
`logger` (logging.Logger)
`pars` (nparray) Gaussian parameters.

Appendix D

Receiver

Figure D.1 shows an overall schematic of the DSS-28 receiver. Figure D.2

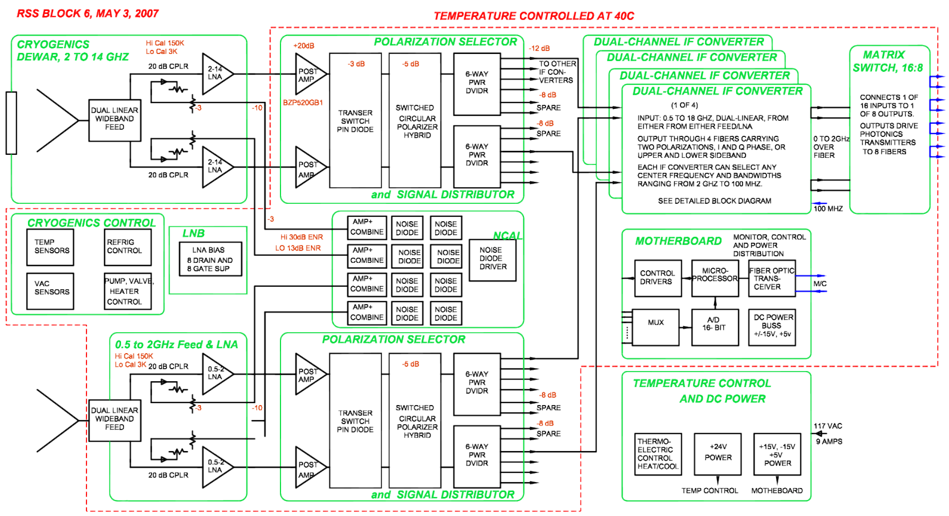

Figure D.1: Schematic diagram of the receiver.

shows an overall schematic of a DSS-28 receiver down-converter.

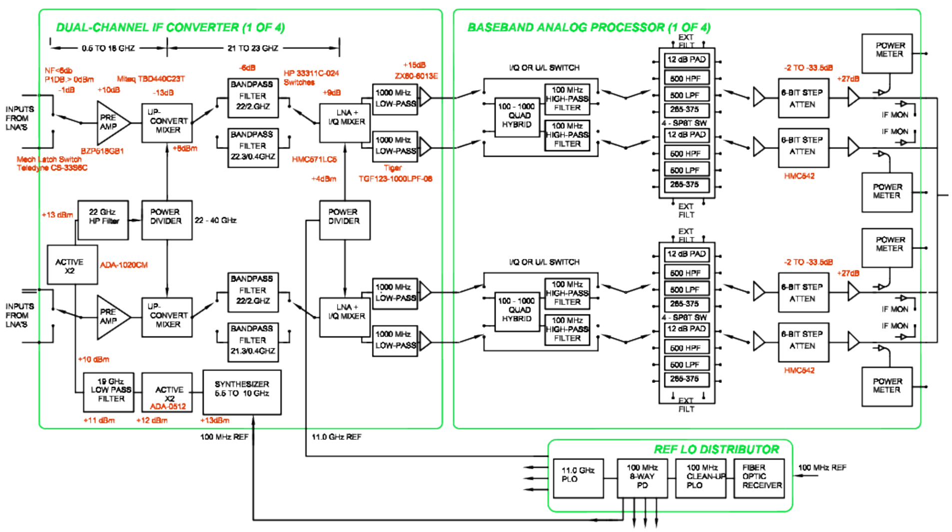

Figure D.2: Schematic diagram of a down-converter.

Bibliography

[Kuiper and Shaff (2019)] Kuiper, T.B.H., and Shaff, D. “General Purpose Radio Telescope Monitor and Control System”, in preparation, 2019.