Sams Analysis Documentation

Release 1.0.0

James R. Fowler

October 11, 2017

1 Introduction **1**
1.1 Sams Nomenclature . . . 1
1.2 Log Files . . . 2

2 Installation **7**
2.1 Location of the source code . . . 7
2.2 Required packages . . . 7
2.3 Setting up your environment . . . 7
2.4 Installing the modules and scripts . . . 8
2.5 Building the Documentation . . . 8

3 Tutorial **9**

4 Classes **15**
4.1 SamsLog . . . 15
4.2 SamsCmdLog . . . 17
4.3 segments.py . . . 18
4.4 plot.py . . . 20

5 Indices and tables **23**

Index **25**

CHAPTER

ONE

INTRODUCTION

The Segment Alignment Maintenance System (Sams) at the Hobby Eberly Telescope (Het) contains 480 sensors that measure gap, shear, and temperature values once per second when Sams is running. Sams calculates target sensor values for each of the 480 sensors every seconds. The output from Sams consists of 273 tip, tilt, and piston values, generated every 35 seconds or so when Sams in controlling the Segement Control System (Scs). In addition, a summary file containing average values and overall system state with a total of nine values, also generated once per second. This is a huge data set to analyze by hand. The Sams Analysis Software (Sas) package provides tools written in Python to easily consolidate multiple log files so that they can be used with visualization tools such as Matplotlib or PyQt.

Various names for this packages have been suggested

- Sams Initial Log Look and anal Ysis (Silly)
- SAMS Correlative Log Understanding-Booster (Sams Club)
- Sams Analysis Tools and Objects Required for Inspection (Satori)
- Sams Application Resource for Comprehensive Analytic Synthesis and Measurement (Sarcasm)
- Sams Log Analysis Program (Slap)
- Sams Analysis Tool and Inquiry ResourcE (Satire)
- Sams Log Analysis Program/Platform (Slapp)

but right now its just called the Sams Analysis Software (SAS). Suggestions and votes are appreciated.

1.1 Sams Nomenclature

1.1.1 Segment Numbering Scheme

There are a number of different segment numbering scheme with in the primary mirror array. These various scheme were developed at different times and for different reasons.

- M number (Mnn) - developed by Frank Ray, an astronomer at UT, this was the first numbing scheme used. It numbers the segment starting from the lower left and ending at the upper right. In the first vision for the mirrors the six corner segments were not included. When they are added later in the program these segment continued the scheme starting sequentially starting from the bottom corner and working counter-clockwise around the array. M number is designated by 'Mnn' where 'nn' runs from 01 to 91 and includes the leading zero.
- CR number (cc:rr) - developed by Phillip McQueen, an astronomer at Het, this scheme designates segments by the column and row of the segment. This scheme makes it easier to visualize where the segment is when discussing segments in conversation or written documents. Note that this is not a rectilinear coordinate system. Column:Row is designated with the notation cc:rr where 'cc' is the column number which may have a leading zero and 'rr' is the row number with or without a leading zero. For example, segment 43, in the center of the array may designated as '6:6', '06:6', or '06:06'

- S number (Snn) - developed by BlueLine Engineering during the installation of the Sams system, this system number the segment in a spiral patter start at the center segment. Therefore the center segment M43 == 6:06 == S01. The S number is designated like the M number by 'Snn' where 'nn' runs from 01-91 and includes the leading zero.
- Sensor numbers (seg-n) - each segment has six sensors except for edge mirrors which will only have sensors where there are neighboring segments. The sensors are designated 1 to 6. When referring to a sensor the sensor number is prefixed by the segment number (in which ever scheme your like) followed by a dash and the sensor number. For example, sensor 2, on the center segment, would be designated as M43-2, S01-2, or 6:06-6.

The SAS software will accept any of these forms of segment-sensor designation whenever a segment or a segment-sensor is required.

1.1.2 Segment and Sensor Location

1.2 Log Files

There are a number of log files that Sams records. The files are generally named as `samsYYYYmmddHHMM.suffix` where suffix will be one of `.deg`, `.ebi`, `.gap`, `.sen`, `.tsn`, `.ttp`, `.sum`, `.log`, `.ref`, or `.dat`.

The following files contain 480 sensors values. Rows in the files are constructed of a date/time of the form `YYYYmmddHHMMSS` followed by either 480 data values. The fields are tab separated. Suffixes and their data are:

- `.deg` – temperature files, each row contains a date/time and the temperature of the individual sensor in degrees C. Contains 481 columns and Sams segment-sensor order.
- `.ebi` – ebias file, these are filed with zeros and are not used in general trouble shooting.
- `.gap` – gap file, each row contains a date/time and the gap (in/out) values of the 480 sensors in microns. The sensors can measure accurately to 1500 microns but I don't know where the upper limit is.
- `.sen` – shear file, each row contains a date/time and the shear (up/down) value of the 480 sensors in microns. Data values will range from -384 to +384 with +/-384 designating a saturated value. Contains 481 columns in Sams segment-sensor order.
- `.tsn` – target sensor files, each row contain a date/time and the residual square error in nanometers for each sensor. Contains 481 columns in Sams segment-sensor order.

The following file contains 273 segment values. Rows in the files are constructed of a date/time of the form `YYYYmmddHHMMSS` The fields are tab separated.

- `.ttp` – tip/tilt/piston files, each row contains a date/time and the tip/tilt/piston values for the individual segments. Contains 274 columen index by M number order. That is, the columns will be: data/time M01 tip, M01 tilt, M01 piston, M02 tip, ..., M91 piston. These data are generate every 35-36 seconds when Sams is sending commands to SCS

The following files contain the state or other information about the Sams server itself. Rows in the files are constructed of a date/time of the form `YYYYmmddHHMMSS` followed by either 9 data values. The fields are tab separated.

- `.sum` – summary files, contain the state of the system at the given time. It contains 10 columns. The columns are:
  1. date/time of the form YYYYmmddHHMM
  2. RSE in nanometers
  3. unknown
  4. Target RSE in nanometers
  5. FWHM of the tip/tilt errors in arc-seconds

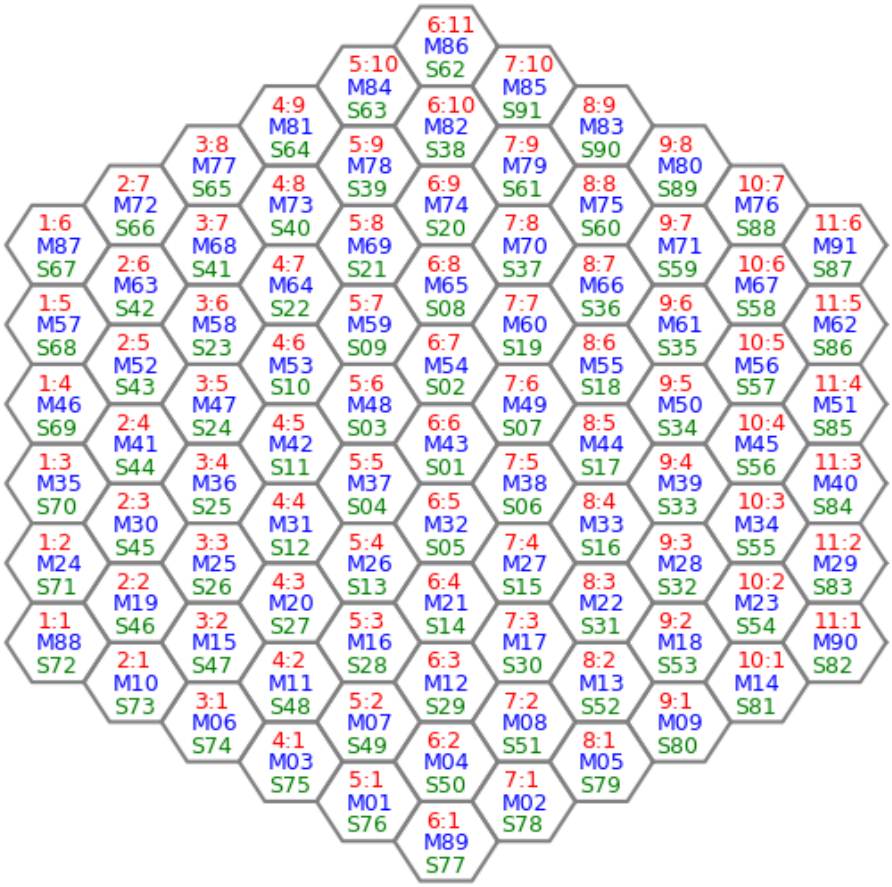

Figure 1.1: The location of the segments in the array and their various designations. The view is looking down on the segment.

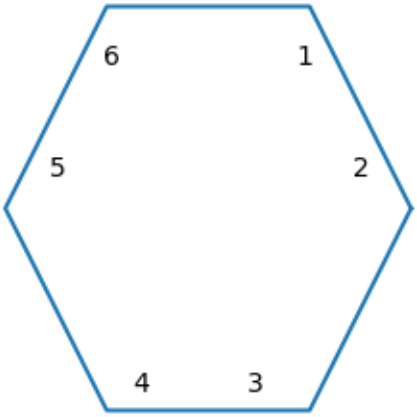

Figure 1.2: The sensor locations on a segment. Sensors are designated 1-6 and the view is looking down on the segment.

6. Average gap in microns
7. Pending GROC (Global Radius of Curvature) correction in microns
8. Total GROC in microns
9. Average temperature of the sensors in degrees C
10. unknown

- `.log` – this file recorded the commands that are send to the Sams server. The files reside in `/home/jove/guider/SAMS/7.1/LOG` with the same file name convention as the other files, i.e. `samsYYYYmmddhhmm.log`. However the time format inside the file is different from the other log files. In this case the time format is YYYY:mm:dd:hh:mm:ss. Interesting commands that can be found are

  1. getsamsdata
  2. getttp
  3. ttpdone nnnnn n
  4. setrefpos /home/jove/guider/SAMS/7.1/REF/sams20170923083048+18.2.ref
  5. setrefpos done
  6. removesen 85 3
  7. installsen 85 3
  8. removeseg 87
  9. installseg 87
  10. setmode Operate
  11. setmode Standby
  12. setcaldata /home/jove/guider/SAMS/7.1/CAL/COEF/coefficients_current

- `.ref` – reference files, contained the reference position for all the sensors. The files are located in `/home/jove/guider/SAMS/7.1/REF`. The naming convention is `samsYYYYmmddHHMMSS[+|-]tt.t.ref`, where tt.t is the temperature in degrees celcius with a plus/minus sign.
- `.dat` – The `.dat` file reside in /home/jove/guider/SAMS/7.1/MIRRORINFO. The naming convention is MirrorInfoYYYYmmddHHMM.dat. These files contain 10 data points consisting of zeros and ones for each of the 91 segments using Mnum for the indexing. It is unclear what the data mean. The source code will have to be searched to find out.

CHAPTER

TWO

INSTALLATION

2.1 Location of the source code

The source code for the **SAS** system may be optained from the svn repository located at the University of Texas at Austin, McDonald Observatory. The machine does not have an anonymous download mechanism and you will need an account on the machine in order to obtain the software. Assumming that you do have an account the command to get the software is

```
svn checkout svn+ssh://<user>@ute2.as.utexas.edu/repos/sas/trunk
```

At the HET the main directory for the source code and the operating environment is `/opt/het/hetdex`.

The executable files are typically installed in `/opt/het/hetdex/bin`; the python packages are typically installed in `/opt/het/hetdex/lib/python2.7/site-packages`; configuration files are typically found in `/opt/het/hetdex/etc/conf`.

2.2 Required packages

The **SAS** software requires the Numpy and and Matplotlib python packages to support operations. These packages may have additional dependencies as well.

2.3 Setting up your environment

Because the executables and libraries are not in one of the standard system locations you will need to modify your environment in order to let the operating system know where to find executables, libraries, and configuration files.

Note that we have a local install of Python 2.7. This was done because the packages available from Redhat are all woefully behind the times.

If you use the bash shell, then you should include the following lines in your .bashrc file.

```
#
# .bashrc setup for HetDex
#

#
# location of third-party and hetdex software executables
#
export HETDEX_DIR=/opt/het/hetdex

#
```

```
# modify environment variables to gain access to the system
#  (this is required to operate the software)
#
export PATH=$HETDEX_DIR/Python-2.7/bin:$HETDEX_DIR/bin:$PATH
export PYTHONPATH=$HETDEX_DIR/lib/python2.7/site-packages
     :$HETDEX_DIR/lib64/python2.7/site-packages:$PYTHONPATH
```

If you use csh or tcsh, then include the following in your .cshrc file.

```
#
# .cshrc set up for HetDex
#

#
# location of third-party and hetdex software executables
#
setenv HETDEX_DIR /opt/het/hetdex

#
# for third-party software locations
#  (this is required to operate the software)
#
setenv PATH $HETDEX_DIR/Python-2.7/bin:$HETDEX_DIR/bin:$PATH
setenv PYTHONPATH $HETDEX_DIR/lib/python2.7/site-packages
     :$HETDEX_DIR/lib64/python2.7/site-packages
```

2.4 Installing the modules and scripts

SAS is a pure python package and can be installed from the **SAS** root directory with the command

```
python setup.py install --prefix=$HETDEX_DIR
```

This will install the module in $prefix/lib/python2.7/site-packages and the scripts in $prefix/bin. This normally done at Het as the user hetdex.

2.5 Building the Documentation

The documentation is created through the Sphinx documentation system. The source files are located in `/doc/sas`.

You can run `make` to get a list of available target formats. Normally the documentation can be built by running `make html latexpdf` in the `./doc` directory. This will build the html pages and the PDF version of the document. The PDF file will be in `./_build/latex/HetPython.pdf` while the HTML files can be found in `./_build/html`. These files may be installed in the directories of your choice.

CHAPTER

THREE

TUTORIAL

Did you read the *Introduction*? Did you read about the *Classes*? You really should! But then again, nobody does. Also it really helps to know a bit about Python as well as Numpy and Matplotlib. But to heck with that. Who reads manuals anyway. Let's get started because this is really the fun part.

Start by launching Python and importing the necessary packages.

```
import matplotlib.pyplot as plt
import numpy as np
from sas import amslog as sl
```

I hope you know where the Sams logs files are kept because we will need to access them. At the Het they can be found in `/home/jove/guider/SAMS/7.1/archive`. I'm just going to use the file names but you may have to prepend an absolute or relative directory to the names in order to get access to it. Let's create our first Sams Log object.

```
sen = sl.SamsLog(['sams201708130000.sen'])
```

Note that you could have listed the five other files to get a full days worth of data or you can use the day prefix as a short cut. For example, you could use this form to get all the files.

```
sen = sl.SamsLog(['sams20170813'], suffix='sen', verbose=True)
Reading file sams201708130000.sen   in 2.734 seconds
Reading file sams201708130400.sen   in 12.328 seconds
Reading file sams201708130800.sen   in 3.099 seconds
File "sams201708131200.sen" does not exist
File "sams201708131600.sen" does not exist
File "sams201708132000.sen" does not exist
Converting time   in 0.286 seconds

sen.files()
['sams201708130000.sen', 'sams201708130400.sen', 'sams201708130800.sen']
```

Refer to the *SamsLog* for more information.

You can find out what type of data are in the object as well as the size of the data.

```
sen.suffix()
'sen'
sen.size()
(26480, 481)
```

The first value (26480) is the number of rows and the second value (481) is the number of columns. Remember that there is a date/time stamp at the start of each row so with 480 sensors that makes 481 columns.

So let's plot some values. These commands will create the following plot.

```
tstamp, sen1, sen2, sen3, sen4, sen5, sen6 = sen.data('M43')
plt.plot(tstamp, sen1, 'r', legend='M43-1')
plt.plot(tstamp, sen2, 'g', legend='M43-2')
plt.legend()
plt.show()
```

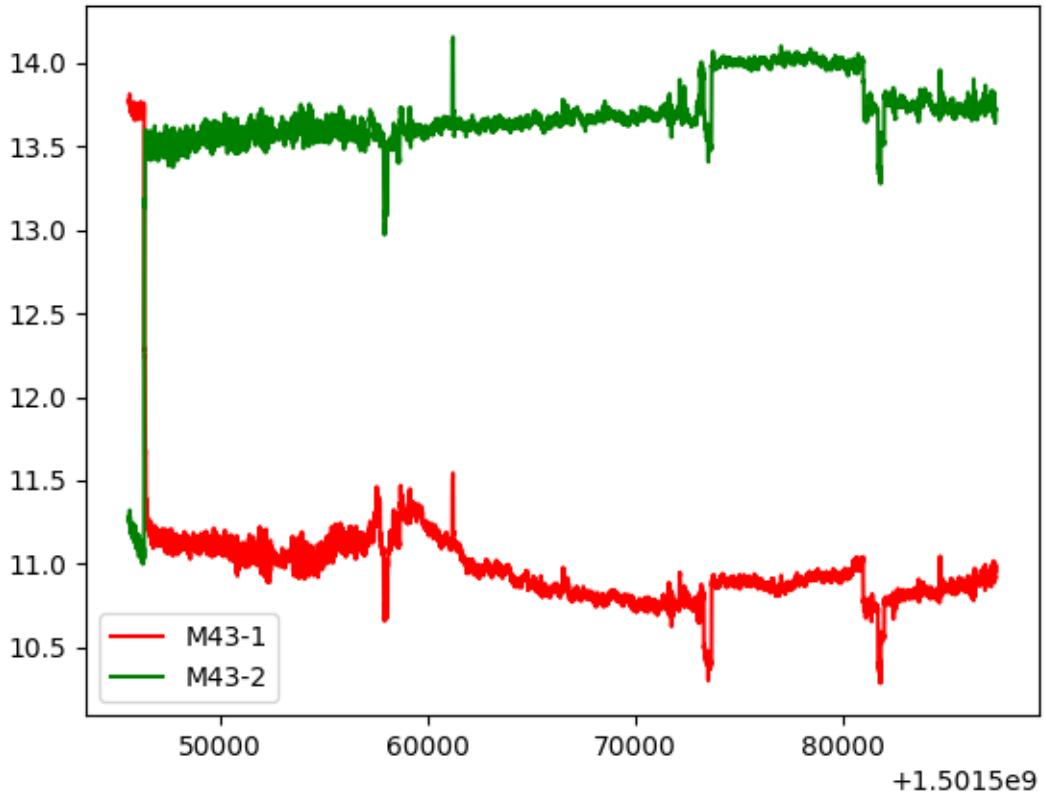

Figure 3.1: Sensor 1 and 2 plots for segment M43

If you ask for an edge segment, then `None` will be returned as the values for that particular sensor data. You should test for this before naively plotting these data.

Similarly you could look at Tip/Tilt data for the same period.

```
ttp = sl.SamsLog(['sams20170813'], suffix='ttp')
ttpstamp, m43tip, m43tilt, m43pist = ttp.data('M43')
plt.plot(ttpstamp, m43tip, 'r', label='M43 Tip')
plt.plot(ttpstamp, m43tilt, 'g', label='M43 Tilt')
plt.ylim(-0.1, 0.1)
plt.legend()
plot.show()
```

Or you could look at the tip/tilt data as a summary plot.

```
plt.quiver(np.zeros_like(m43tip), np.zeros_like(m43tilt), m43tip, m43tilt)
plt.show()
```

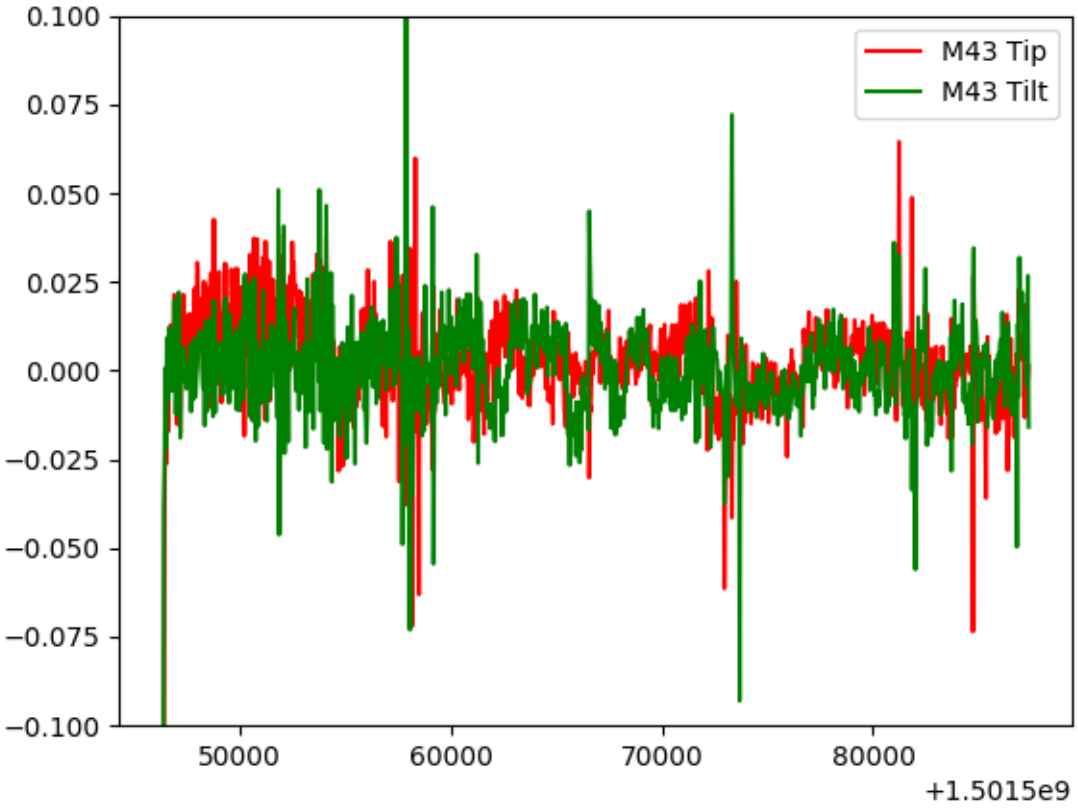

Figure 3.2: Tip and tilt plotted for Segment M43

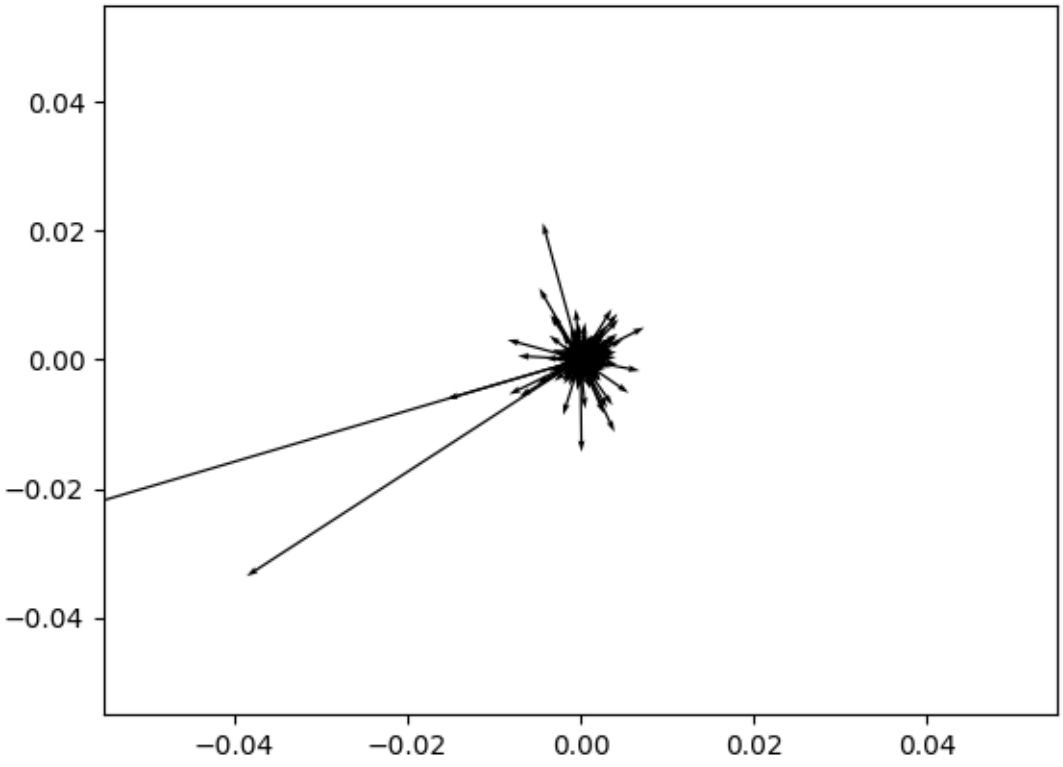

Figure 3.3: All the tip/tilts from segment M43 plotted as vectors

If you want to get even fancier, it is possible to plot an entire nights tip/tilt data for the entire array.

```
ttp = sl.SamsLog(['./data/sams20170813'], suffix='ttp', verbose=True)
plot_ttp_summary(ttp)
```

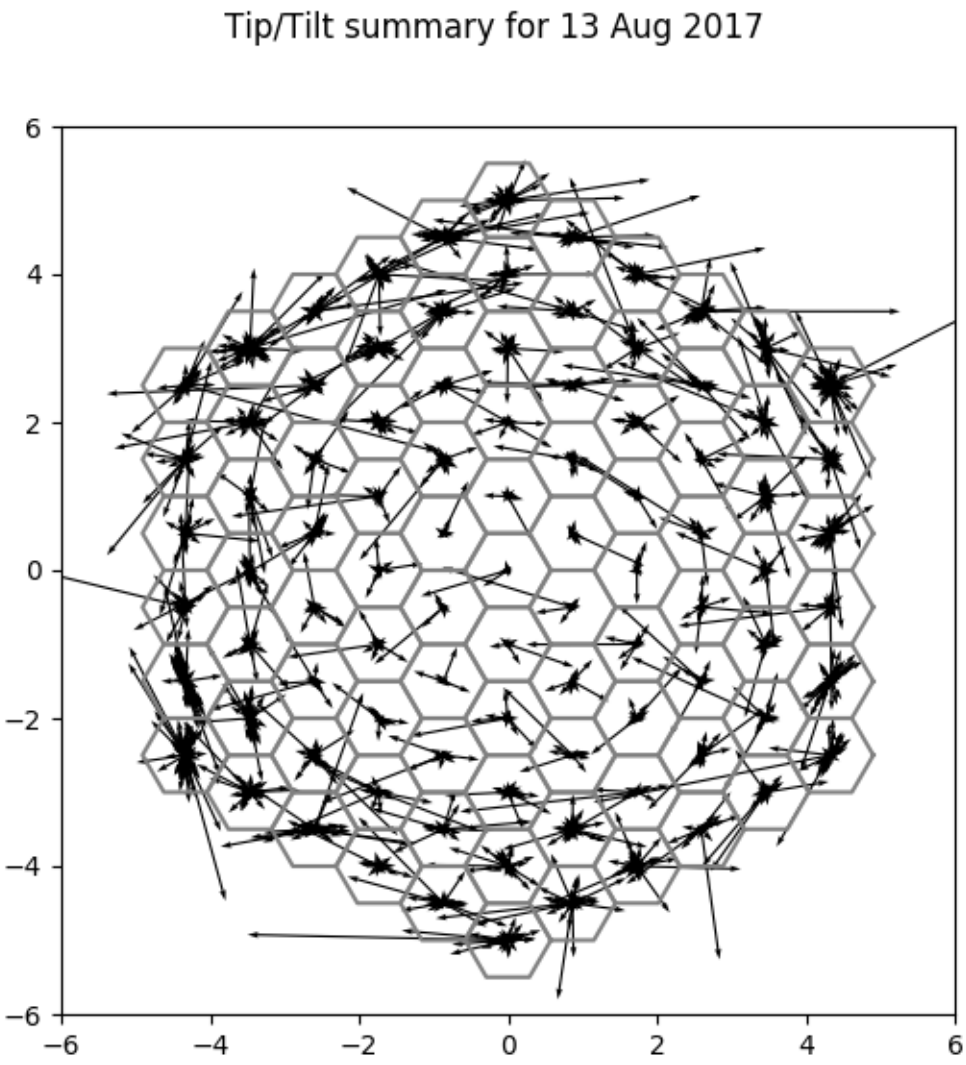

Figure 3.4: The tip/tilt data for all segments.

The summary files contain global and average information for the time period. You can make a summary plot with the command `plot_sum_summary()`.

```
sum = sl.SamsLog(['./data/sams20170813'], suffix='sum', verbose=True)
plot_sum_summary(sum)
```

Numerous possibilities are open to you. You just need to use your imagination.

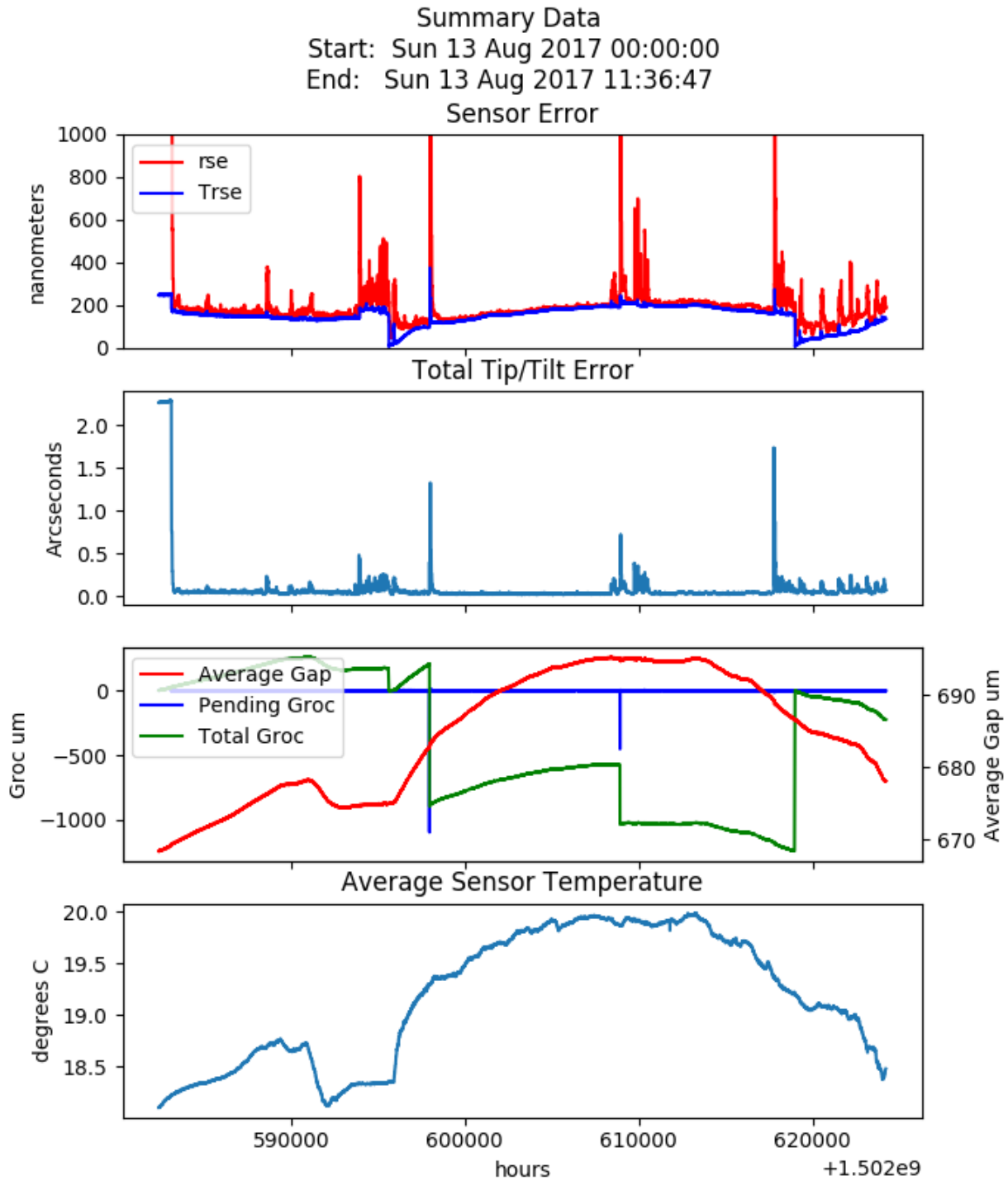

Figure 3.5: The summary plot of global and average values.

CHAPTER

FOUR

CLASSES

SAS is a pure python package containing specialized data containers as well as grahical tools to ease the visualization of Sams data.

4.1 SamsLog

from sas import samslog

The SamsLog() class provides a data container capable of consolidating a number of logs file of the same type into one structure.

class `samslog.`**`SamsLog`**(*filenames*, *suffix=None*, *verbose=False*)

The input variable `filenames` must be a list of valid log file names and/or a list of valid day names. If the names are not appended with a suffix, then the suffix parameter will be used. If the first filename has a suffix and no suffix parameter is given, then the first suffix will be used for the other files. Valid strings for the suffix are `deg`, `ebi`, `gap`, `sen`, `sum`, `tsn`, and `ttp`. c.f. *Log Files*.

Since the Sams log files contain only four hours of data you are allowed to use a syntax like,

```
allday_shear = SamsLog(['sams20170813'], suffix='sen')
```

where `sams20170813` will be expanded to a list with the six files for that day. The suffix `.sen` will be appended. For example,

```
allday_shear.files()
['sams201708130000.sen', 'sams201708130400.sen', 'sams201708130800.sen',
 'sams201708131200.sen', 'sams201708131600.sen', 'sams201708132000.sen']
```

You may also pass in a list of files to read from,

```
allday_shear = SamsLog(['sams201708130000.sen','sams201708130400.sen',
                        'sams201708130800.sen','sams201708131200.sen',
                        'sams201708131600.sen','sams201708132000.sen'])
```

Or you may do a combination of both,

```
allday_shear = SamsLog(['sams201708131600.sen','sams201708132000.sen',
                        'sams20170814',
                        'sams201708150000.sen','sams201708150400.sen'])
```

Note that the class will pick up the `.sen` suffix from the first file name and will apply it to any file name that doesn't have a suffix yet.

Also be aware that the data files are read into the data container in the order in which they are lists. No attempt is made to sort the resulting data by time stamp.

4.1.1 Methods

Once the filename(s) are parsed and the files are open, the class then reads the file(s), works out whether it has sensor data (480 column), segment date (273 columns), or summary data (9 columns). It can then work out the proper address scheme from segment or segment:sensor to a data column.

classmethod `samslog.`**`append`**(*filenames*, *suffix*)
Like the class constructor filenames should be a list of file you want to append to the existing data. Again, these data are not sorted by date/time. The input variable, suffix, has the same connotation as in the class initialization.

classmethod `samslog.`**`data`**(*segsens*)
This method returns two or more lists; the first list contains the time stamps in Unix time format; the remaining lists depend on whether a segment number of a segment-sensor number was sent as the input argument *Segment Numbering Scheme*. If only a segment number is passed in, then all the data relevant to that segment will be returned. For `.sen` files, this would be six lists of sensor data. For `.ttp` files, this would be 3 lists containing the tip/tilt/piston values for the segment. If a segment-sensor value is passed in, then any sensor related file will return a single column with just that sensor's values.

For example,

```
shear = SamsLog('sams201708130000.sen')
tstamp, shear23_2 = shear.data('M23:2')

gap = SamsLog('sams201708130000.gap')
tstamp, gap23_1, gap23_2, gap23_3, gap23_4, gap23_5, gap23_6 = gap.data('M23")
```

Note that for any edge segments, if you ask for all the sensors or a non-existant segment-sensor combinartion then `None` will be returned for the non-existant sensors.

It is not an error to pass a segment-sensor value as an input to a `.ttp` data set, but the sensor value will meaningless.

The return values are of type <class 'numpy.ndarry'> and are one-dimensional row vectors.

classmethod `samslog.`**`starttime`**()

classmethod `samslog.`**`endtime`**()
The class also has instance variables with the start and end time of the file(s) that have been read in. These may be accessed though the methods as follows

```
gap.starttime()
1502582400.0
```

classmethod `samslog.`**`files`**()
This method returns the complete list of files that have been read into the data set. Thus it represents the expansion of the file list passed in during intialization and append functions.

classmethod `samslog.`**`suffix`**()
Return the suffix or type of file with out the dot.

```
gap = SamsLog([sams20170813', suffix='gap')
gap.suffix()
'gap'
```

classmethod `samslog.`**`size`**()
Returns the size of the data contains as the value `rows:columns`.

```
gap.size()
'32614:481'
```

4.2 SamsCmdLog

from sas import samcmdlog

The SamsCmdLog() class is very similar to the SamsLog() class in that it provides a data container for the `log` files. These files list the commands received by the Sams server.

class `segments.`**`SamsCmdLog`**(*filenames*, *suffix='log'*, *verbose=False*)

The input variable `filenames` must be a list of valid log file names and/or a list of valid day names. If the names are not appended with a suffix, then the suffix parameter `log` will be used. The `suffix` argument is still in the calling pattern for the class but it is ignored. However, the class will generate an error if the `suffix` argument is passed in with anything other than `'log'` as the argument. This is done only for consistency with the class *SamsLog*.

The usage of the `filenames` variable is the same as in the *SamsLog* class. You may use an abbreviated file name to include all the files for a full day. Or you may use the explicit list of files, with or with out the suffix.

allcmds = SamsCmdLog(['sams20170813'])

morning_cmds = SamsCmdLog(['sams201708130000', 'sams201708130400', 'sams201708130800.log'])

4.2.1 Methods

Once the files are read into the container, various methods are available to work with the data

classmethod `segments.`**`append`**(*filenames*, *suffix='log'*)
Like the class constructor filenames should be a list of file you want to append to the existing data. Again, these data are not sorted by date/time. The input variable, suffix, has the same connotation as in the class initialization.

classmethod `segments.`**`find_command`**(*command=None*, *starttime=None*, *endtime=None*)
Return all instances of command found between starttime and endtime. An instance is a tuple of the form (time, commandStr). If starttime/endtime are None then the entire list is searched otherwise only the tuples found between startime and endtime are returned. If command is None, then all commands between starttime/endtime are returned. If all three are None, then the entire list is returned.

```
allcmds.find_command('installsen')
[(15345678901, 'installsen 85 3'),
(12456789012, 'installsen 32 1')]
```

classmethod `segments.`**`starttime`**()

classmethod `segments.`**`endtime`**()
The class also has instance variables with the start and end time of the file(s) that have been read in. These may be accessed though the methods as follows

```
allcmds.starttime()
1502582400.0
```

classmethod `segments.`**`files`**()
This method returns the complete list of files that have been read into the data set. Thus it represents the expansion of the file list passed in during intialization and append functions.

classmethod `segments.`**`suffix`**()
Return the suffix or type of file with out the dot. In this particular class the suffix will always be *log*

```
cmds = SamsCmdLog([sams20170813'])
cmds.suffix()
'log'
```

classmethod segments.**size**()
Returns the number of entries in the data container

cmds.size() 3261

4.3 segments.py

from sas import segments

The file segments.py contains the conversion tables between segment number schemes. These conversion tables are simply dictionaries with keys in the relevant mirror numbering scheme, c.f. *Segment Numbering Scheme*.

segments.**Mn2Sn**

segments.**Sn2Mn**
Convert to/from the Mirror number of Frank Ray to the Sams number scheme from Blueline Engineering.

```
Mn2Sn['M02']
'S78'

Sn2Mn['S78']
'M02'
```

segments.**CR2Sn**

segments.**Sn2CR**
Convert to/from the Column:Row numbering scheme of Phillip McQueen and the Sams number scheme from Blueline Engineering.

```
CR2Sn['7:01']
'S78'

Cn2CR['S78']
'7:01'
```

segments.**Mn2CR**

segments.**CR2Mn**
Convert to/from the Mirror number of Frank Ray to the Column:Row of Phillip McQueen.

```
Mn2CR['M02']
'7:01'

CR2Mn['7:01']
'M02'
```

segments.**m_re**

segments.**s_re**

segments.**cr_re**
This are regular expession object as described in the regular expression package re. They are build with `re.compile()`, c.f. re for more information. There is one each for the Mnumbers, the Snumbers, and the CRnumbers.

The regular expression used to create the objects are

- M_re = r'^(M(\d\d))(-(\d))?\Z'
- S_re = r'^(S(\d\d))(-(\d))?\Z'
- CR_re = r'^((\d{1,2}):(\d{1,2}))(-(\d))?\Z'

segments.**SnSen_column**
This array converts from segment-sensor number in the Sams numbering scheme to column number in the data array for 'sen', 'gap', 'tsn', or 'deg' files. It may also be used to see it a sensor exists by testing for a Snum-sen key value and seeing if you get a return value.

segments.**get_seg_sen_type**(*segsen*)
Parse a segment-sensor string and return the type of numbering system used. The functions returns a tuple of five values;

- segment name in string form, i.e. M54 or S23
- segment number or the column number if this is a CRnum as an integer
- row number as an integer of this is a CRnum, otherwise None
- sensor number if provided, otherwise None
- type of number as 'Mnum', 'Snum', 'CRNum', or None

For example:

```
get_seg_sen_type('M43-2')
('M43', 43, None, 2, 'Mnum')
```

A CRnum may have leading zero in either the columm or the row but the number may only be two digits. The returned string however, will not have any leading zeros and may be used to index into the CR2Sn or CR2Mn arrays.

```
get_seg_sen_type('05:04-6')
('5:4', 5, 4, 6, 'CRnum')

get_seg_sen_type('05:04')
('5:4', 5, 4, None, 'CRnum')
```

segments.**get_sams_num**(*segsen*)

segments.**get_mir_num**(*segsen*)
These two functions return the S number or the M number given any valid numbering scheme as input. Two values are returned, the S or M number as a string and the S or M number as an integer:

```
get_sams_num('M42')
('S11', 11)

get_sams_num('2:4-6')
('S44', 44)

get_mir_num('S11')
('M42', 42')

get_mir_num('5:4')
('M26', 26)
```

row_cols = [7, 8, 9, 10, 11, 12, 11, 10, 9, 8, 7]
A list with the number of rows in each column, 1-11. This list is useful if you wish to do looping over the column:row order, e.g.

```
for c = range(0,11):
  row = row_cols[c-1]
  for r in range(0, row):
```

segments.**gen_col_row**()
Generate a list of the column:row strings for the segments in the array. This is useful for running loops in column:row index and is used extensively in the plotting routines, e.g.

```
for cr in gen_col_row():
  m_num, empty = get_mir_num(cr)
  x, y = plot.segment_position[cr]
  plot_something_clever(m_num, x, y)
```

4.4 plot.py

from sas import plot

The file plot.py contains some useful plotting routines. Plotting of the full array or a single segment is based on a unit sized hexagon where the distance from the edge to edge is set to one unit.

As we work out other items that need to be be routinely plotted, they can be added to this file.

4.4.1 Classes

class plot.**Hexagon**(*size = 1.0*, *offset=(0.0, 0.0)*)
The unit hexagon. The create vertices for a hexagon with a 1.0 meter distance from edge to edge. Note that this means that the vertices fall outside the unit cell so watch out for overlap when plotting. The offset (x, y) are added to each vertex and will offset the hexagon from a (0, 0) center position. Useful for plotting routines that want to draw a hexagon.

plot.**x**

plot.**y**
These are the lists of the x, y position of the vertices. They can be access as

```
h = Hexagon()
plt.plot(h.x, h.y)
```

Note that this syntax may change. I would like to make Hexagon a derived class from a parent class Polygon but haven't figured out how to make things work yet.

plot.**vertices**()
Returns a list of tuples of the (x, y) positions of the vertices.

```
h = Hexagon()
h.vertices()
[(,)
]
```

4.4.2 Functions

plot.**make_seg_pos**()
Make a dictionary with CRnums as keys and values as a tuple of the (x, y) position of the segment in the array based on a unit hexagon. This means the plot will be 11.0 units high and 11.15 units wide.

plot.**segment_position**
A dictionary with keys of CRnum and values of (x, y) for the segment positions based on a unit hexagon. This dictionary was initially create with `make_seg_pos()`.

This is useful in loops like

```
from sas import segments as sg
from sas import plot as sp

for cr in sg.gen_col_row():
  x, y = sp.segment_position[cr]
  plot_something_clever(x, y)
```

plot.**plot_array**(*ax*, *color='grey'*)
Plots 91 unit hexagons in their proper positions. See *segment locations* for an example. `ax` is a reference to a `matplotlib.axes` instance. `color` is any valid matplotlib color, e.g.

```
fig = plt.figure(figsize=(7,7))
ax = fig.add_subplot(111)
plot_array(ax, color='red')
```

plot.**plot_ttp_summary**(*ttplog*)
Plot the Tilt/Tilt vectors agains an array of segments

plot.**plot_sum_summary**(*sumlog*)
Plot the data from a summary files. Three plots are created; the sensor error as RSE and TRSE; the gap and groc; and the average sensor temperature.

More functions as we think of them.

CHAPTER

FIVE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

A

append() (in module samslog), 16
append() (in module segments), 17

C

CR2Mn (in module segments), 18
CR2Sn (in module segments), 18
cr_re (in module segments), 18

D

data() (in module samslog), 16

E

endtime() (in module samslog), 16
endtime() (in module segments), 17

F

files() (in module samslog), 16
files() (in module segments), 17
find_command() (in module segments), 17

G

gen_col_row() (in module segments), 20
get_mir_num() (in module segments), 19
get_sams_num() (in module segments), 19
get_seg_sen_type() (in module segments), 19

H

Hexagon (class in plot), 20

M

m_re (in module segments), 18
make_seg_pos() (in module plot), 20
Mn2CR (in module segments), 18
Mn2Sn (in module segments), 18

P

plot_array() (in module plot), 21
plot_sum_summary() (in module plot), 21
plot_ttp_summary() (in module plot), 21

S

s_re (in module segments), 18
SamsCmdLog (class in segments), 17
SamsLog (class in samslog), 15
segment_position (in module plot), 20
size() (in module samslog), 16
size() (in module segments), 18
Sn2CR (in module segments), 18
Sn2Mn (in module segments), 18
SnSen_column (in module segments), 19
starttime() (in module samslog), 16
starttime() (in module segments), 17
suffix() (in module samslog), 16
suffix() (in module segments), 17

V

vertices() (in module plot), 20

X

x (in module plot), 20

Y

y (in module plot), 20