# GAVO DaCHS: File Processing

**Author**: Markus Demleitner
**Email**: gavo@ari.uni-heidelberg.de

## Contents

This is a manual on how to use DaCHS' helpers to preprocess data before ingesting it and do other things based on iterating over lots of sources.

Sometimes you want to change something on the input files you are receiving. While usually we recommend coping with the input through grammars, rowmakers, and the like since this helps maitaining consistency with what the scientists intended and also stability when new data arrives, there are cases when you deliver data to users, most frequently, with FITS files. There, you may need to add or change headers.

However, sometimes you just want to traverse all sources, maybe to validate them, maybe to compute something from them; the prime example for the latter is pre-computing previews.

## Processors

The basic infrastructure for manipulating sources is the FileProcessor class, available from gavo.helpers.

Here is an example checking whether the sizes of files match what an (externally defined) function `_getExpectedSize(fName) -> int` returns:

```
import os

from gavo import api

class SizeChecker(api.FileProcessor):

  def process(self, srcName):
    found = os.path.getsize(srcName)
    expected = _getExpectedSize(srcName)
    if found!=expected:
      print "%s: is %s, should be %s"%(srcName, found, expected)


if __name__=="__main__":
  api.procmain(SizeChecker, "potsdam/q", "import")
```

The call to `procmain` arranges for the command line to be parsed and expects, in addition to the processor *class*, an id for the resource descriptor for the data it should process, and the id of the data descriptor that ingests the files.

As usual, you can raise `base.SkipThis()` to pretend process had never been called for a certain `srcName`.

### Processor Command line

The processors can define command line options of their own. You could, for example, read the expected sizes from some sort of catalogue. To do that, define an addOptions static method, like this:

```
class Processor(api.FileProcessor):
  @staticmethod
  addOptions(optParser):
    api.FileProcessor.addOptions(optParser)
    optParser.add_option("--cat-name", help="Resdir-relative path to"
      " the plate catalogue", action="store", type="str",
      dest="catPath", default="res/plates.cat")
```

Make sure you always do the upward call. Cf. the optparse documentation for what you can do. The options object returned by optParser is available as the opts attribute on your processor. To keep the chance of name clashes in this sort of inheritance low, always use long options only.

Simple FileProcessors support the following options:

| | |
|---|---|
| --**filter** | It takes a value, a substring that has to be in the source's name for it to be processed. This is for when you want to try out new code on just one file or a small subset of files. |
| --**bail** | Rather than going on when a process method lets an exception escape, abort the processing at the first error and dump a traceback. Use this to figure out bugs in your (or our) code. |
| --**report** | More on this in Processor Report Generation |
| -**j** | Number of processes to run in parallel (Parallel Execution) |

### Auxiliaries

Once you have the catalogue name, you will want to read it and make it available to the process method. To allow you to do this, you can override the _createAuxiliaries(dd) method. It receives the data descriptor of the data to be processed. Here's an example:

```
class Processor(api.FileProcessor):
  def _createAuxiliaries(self, dd):
    self.catEntriesUsed = 0
    catPath = os.path.join(dd.rd.resdir, self.opts.catPath)
    self.catalogue = {}
    for ln in open(catPath):
      id, val = ln.split()
      self.catalogue[id] = val
```

As you can see, you can access the options given on the command line as self.opts here.

## Parallel Execution

Processors in principle can be executed in parallel *processes* (using the `-j` flag as with make), provided they are written to support this – which means no temporary files that could have name clashes, no other shared mutable resources without synchronization, and so on.

The main problem with when forking out workers are database connections – in short, if you want to run your processors in parallel, you must make sure you're not using shared database connections. In particular, you cannot use the familiar `with base.getTableConn() as conn:` pattern.

The preferred way to deal with things is to create a database connection in `createAuxiliaries` and call it `conn` (yes, DaCHS looks at the name), like this:

```
class FooProcessor(FileProcessor):
  def _createAuxiliaries(self, dd):
    self.conn = base.getDBConnection("trustedquery")
    FileProcessor._createAuxiliaries(self, dd)
```

Based on the name `conn`, DaCHS will close the connection and reopen it when forking. If all queries go through this connection, all should be well for multiprocessing. Since processors should normally have no business writing to the database, the connection is for the *trustedquery* profile. If you absolutely have to write, use the *feed* profile, but note that you will have to manually commit then.

Note that some processor classes (`PreviewMaker`, in particular) already open such a connection for you so you don't have to do anything for these.

## Gathering Data

If you want your processor to gather data, you can use the fact that procmain returns the processor it created. Here is a version of the simple size checker above that outputs a sorted list of bad files:

```
class SizeChecker(api.FileProcessor):

  def _createAuxiliaries(self, dd):
    self.mess = []

  def process(self, srcName):
    found = os.path.getsize(srcName)
    expected = _getExpectedSize(srcName)
    if found!=expected:
      self.mess.append((srcName, expected, found))


if __name__=="__main__":
  res = api.procmain(SizeChecker, "potsdam/q", "import")
  res.mess.sort(key=lambda rec: abs(rec[1]-rec[2]))
  for name, expected, found in res.mess:
    print "%10d %10d %8d %s"%(expected, found, expected-found, name)
```

## Processor Report Generation

Most of the time, when gathering data (or otherwise), what you are doing is basically generate a report of some sort. For such simple cases, you will usually want to use the --report option. This causes the processor to skip process and instead call a method that will in turn call the `classify(sourceName)` method. It must return a string that will serve as a class label. At the end of the run, the processor will print a summary of the class frequencies.

Here's what such a classify method could look like:

```
def classify(self, srcName):
  hdr = self.getPrimaryHeader(srcName)
  try:
    ignored = "FILTER_A" in hdr
    return "ok"
  except ValueError: # botched cards on board
    return "botched"
```

## Overriding the Sources

By default, processors iterate over all the sources returned by the referenced data element's sources element. Sometimes that is not what you want, typically

because some rowfilter adds things or because the data is completely virtual and the input files only have a very loose relation to what is published through the service.

In these cases, override the processor's iterIdentifiers method. It has to yield things suitable as the parameter for process. It is a good idea to have these be strings, though you might get away with other objects if you accept that some error messages may look funny.

The classical case is getting accrefs from a table, like this:

```
from gavo import api
...

  def iterIdentifiers(self):
    tableId = self.dd.makes[0].table.getQName()
    with api.getTableConn() as conn:
      for r in conn.queryToDicts("select accref from %s"%tableId):
        yield r["accref"]
```

A very typical case is when an "artificial" format generated on the fly gets added to the SDM table to return something for FORMAT=compliant queries. In the RD, this could look like this:

```
<rowfilter procDef="//products#define">
  <bind name="table">"\schema.data"</bind>
  <bind name="mime">"image/fits"</bind>
  <bind name="preview_mime">"image/png"</bind>
  <bind name="preview">\standardPreviewPath</bind>
</rowfilter>
<rowfilter name="addSDM">
  <code>
    yield row
    baseAccref = os.path.splitext(row["prodtblPath"])[0]
    row["prodtblAccref"] = baseAccref+".vot"
    row["prodtblPath"] = "dcc://\rdIdDotted/mksdm?"+urllib.quote(
      row["prodtblPath"])
    row["prodtblMime"] = "application/x-votable+xml"
    yield row
  </code>
</rowfilter>
```

Note that the preview path and mime are the same for both versions, which means that previews should only be computed for the first kind of data. To effect that, write your PreviewMaker like this:

```
class PreviewMaker(api.SpectralPreviewMaker):
  sdmId = "build_sdm_data"
```

```
def iterIdentifiers(self):
  for id in api.SpectralPreviewMaker.iterIdentifiers(self):
    if not id.endswith(".vot"):
      yield id
```

### Utility Methods

FileProcessor instances have some utility methods handy when processing files for DaCHS:

- `getProductKey(fName) -> str` returns the "product key" fName would have; this currently is just fName's path relative to the inputsDir (or an exception if fName is not below inputsDir). This method lets you easily interchange data between your file processor and ignore elements or the inputRelativePath macro in RDs.

## Precomputing previews

While DaCHS can compute previews of 2D FITS images on the fly, in many cases there are good reasons to precompute previews. If you follow some conventions when doing this, the process becomes much smoother.

When making previews, it is usually much more convenient to work with accrefs rather than actual file paths. That is particularly true with spectra, which in DaCHS frequently are virtual data, such that an accref doesn't correspond to an actual file.

Where there are actual files and you didn't do any magic with the accrefs, you can retrieve the full path by computing `os.path.join(api.getConfig("inputsDir"), accref)`.

### api.PreviewMaker

The DaCHS API contains a `PreviewMaker` class with some convenience methods. To use it, give the data descriptor a `previewDir` property, like this:

```
<data id="import">
  <property key="previewDir">previews</property>
  ...
```

– the value is the resdir-relative name of the directory that will contain the preview files.

This previewDir property is evaluated by the preview name generators (and only there; if you set up a naming policy of your own, there's no need to set previewDir). DaCHS currently has two of those, both available as macros for use in products#define. Here's how to use them:

```
<rowfilter procDef="//products#define">
  <bind name="table">"\schema.data"</bind>
  <bind name="mime">"image/fits"</bind>
  <bind name="preview_mime">"image/png"</bind>
  <bind name="preview">\standardPreviewPath</bind>
</rowfilter>
```

The `standardPreviewPath` macro arranges things such that all previews are in one directory with base64 encoded names. This is fairly low overhead and is recommended for smallish data collections up to, say, a few thousand datasets.

For larger data collections, it is recommended to use the `splitPreviewPath{extension}` macro. It arranges the previews in a hierarchy analogous to the data files themselves. In order to avoid confusion, it is recommended to set the extension according to the file type generated (i.e., typically ".png" or ".jpeg"), like this: `\splitPreviewPath{.png}`.

To generate the previews, all you have to do is inherit from `PreviewMaker` and implement `getPreviewData(srcName) -> imageData`. PIL, stuff from utils.imgtools or something similar usually is your friend here. Here's a full example that would compute 200x100 one-channel jpegs for some image format understood by PIL:

```
import os
from cStringIO import StringIO

import Image

from gavo import api

class PreviewMaker(api.PreviewMaker):
  def getPreviewData(self, accref):
    srcName = os.path.join(api.getConfig("inputsDir"), accref)

    im = Image.open(srcName)
    scale = max(im.size)/200.
    resized = im.resize((
      int(im.size[0]/scale),
      int(im.size[1]/scale)))

    rendered = StringIO()
    resized.save(rendered, format="jpeg")
    return rendered.getvalue()
```

```
if __name__=="__main__":
  api.procmain(PreviewMaker, "example/q", "import")
```

If this were in `bin/mkpreview.py`, you could then say:

```
python bin/mkpreview.py
```

to compute previews for all files that don't have one yet, and you can call:

```
python bin/mkpreview.py --report
```

to see if previews are missing.

As another example, here's how you can statically generate the previews that DaCHS would make for FITS images; the classic case when you want this when the service has datalinks as accrefs (which, at least for now, DaCHS doesn't handle automatically):

```
import os

import numpy

from gavo import api
from gavo.utils import fitstools, imgtools


PREVIEW_SIZE = 200

class PreviewMaker(api.PreviewMaker):
  def getPreviewData(self, srcName):
    with open(os.path.join(api.getConfig("inputsDir"), srcName)) as inFile:
      pixels = numpy.array([row
        for row in fitstools.iterScaledRows(inFile,
          destSize=PREVIEW_SIZE)])
    return imgtools.jpegFromNumpyArray(pixels)


if __name__=="__main__":
  api.procmain(PreviewMaker, "plts/q", "import")
```

Finally, here's how you could compute color previews when you have images in three filters in the FITS extensions 2, 3, and 4:

```
import numpy

from gavo.utils import fitstools
from gavo.utils import imgtools
```

```
from gavo.utils import pyfits


def _getArrayFor(srcName, extInd):
  return numpy.array(list(
      fitstools.iterScaledRows(srcName, destSize=200, extInd=extInd)))


class PreviewMaker(api.PreviewMaker):
  def getPreviewData(self, srcName):
    return imgtools.colorJpegFromNumpyArrays(
      _getArrayFor(srcName, 1),
      _getArrayFor(srcName, 2),
      _getArrayFor(srcName, 1))

if __name__=="__main__":
  api.procmain(PreviewMaker, "lmu/q", "import_imgs")
```

### Making Previews for Spectra

If you already have a datalink service defined for making SDM-compliant spectra, you can easily re-use that to generate spectral previews. For that, there's `api.SpectralPreviewMaker`. All it needs is the id of data element making the SDM instances in the `sdmId` class attribute. The following would do in a typical case:

```
from gavo import api

class PreviewMaker(api.SpectralPreviewMaker):
  sdmId = "build_sdm_data"


if __name__=="__main__":
  api.procmain(PreviewMaker, "flashheros/q", "import")
```

By default, this produces spectra that are logscaled on the flux axis. You can set the class attribute `linearFluxes = True` to have linear scaling instead if that works better for your data.

On noisy spectra, presentation might be improved by setting a class attribute `connectPoints = False`.

## Basic FITS Manipulation

For manipulating FITS headers, there are the ImmediateHeaderProcessor and HeaderProcessor classes. The difference is that the full HeaderProcessor first writes detached headers and only applies them in a second step. That's usually advisable for major surgery, in particular with largish files.

Both are FileProcessors, so everything said there applies here as well, except that you usually do not want to override the process method.

With the simple ImmdiateHeaderProcessors, you simply override `_isProcessed(srcName, hdr)` that should return False whenever the action still is necessary (the default always returns False, so it's (overly) safe to just let it stand), and `_changeHeader(hdr) -> ignored`, which is expected to change the primary header passed to it in place. The changed header will then be written back to disk, if possible without touching the data part.

Here's an example for a simple ImmediateHeaderProcessor:

```
import os

from gavo import api

class LinkAdder(api.ImmediateHeaderProcessor):
  def _createAuxiliaries(self, dd):
    self.staticBase = dd.rd.getById("dl").getURL("static")

  def _isProcessed(self, srcName, hdr):
    return hdr.get("FN-PRE", "").startswith("http")

  def _changeHeader(self, srcName, hdr):
    baseName = os.path.splitext(os.path.basename(srcName))[0]
    hdr.set("FN-WEDGE", "%s/wedges/%sw.fits"%(self.staticBase, baseName),
      after="FILENAME")
    hdr.set("FN-PRE", "%s/jpegs/%s.jpg"%(self.staticBase, baseName),
      after="FN-WEDGE")


if __name__=="__main__":
    res = api.procmain(LinkAdder, "kapteyn/q", "import")
```

With HeaderProcessors, you will rather to override the `_isProcessed(srcName) -> boolean` method and one of

- `_mungeHeader(srcName, header) -> pyfits hdr` or
- `_getHeader(srcName) -> pyfits hdr`.

`_isProcessed` must return True if you think the name file already has your new headers, False otherwise. Files for which _isProcessed returns True are not touched.

`_getHeader` is the method called by process to obtain a new header. It must return the complete new header for the file named in the argument. Since it

is very common to base this on the file's existing header, there is `_mungeHeader` that receives the current header.

_mungeHeader should in general raise a api.CannotComputeHeader exception if it cannot generate a header (e.g., missing catalogue entry, nonsensical input data). If you return None from either _mungeHeader or _getHeader, a generic CannotComputeHeader exception will be raised.

Note again that you have to return a *complete* header, i.e., including all cards you want to keep from the original header (but see Header Selection).

A somewhat silly example could look like this:

```
from gavo import api

class SillyProcessor(api.HeaderProcessor):
  def _isProcessed(self, srcName):
    return "NUMPIXELS" in self.getPrimaryHeader(srcName)

  def _mungeHeader(self, srcName, hdr):
    hdr.set("NUMPIXELS", hdr["NAXIS1"]*hdr["NAXIS2"])
    return hdr

if __name__=="__main__":
  api.procmain(SillyProcessor, "testdata/theRD", "sillyData")
```

Call --help on the program above to see FileProcessor's options (if you want to add more, see Processor Command Line. Things are arranged like this (check out the process and _makeCache methods in the source code), where proc stands of the name of the ingesting program:

- `proc` computes headers for all input files not yet having "cached" headers. Cached headers live alongside the fits files and have ".hdr" attached to them. The headers are *not* applied to the original files.
- `proc --apply --no-compute` applies cached headers to the input files that do not yet have headers. In particular when processing is lengthy (e.g., astrometrical calibration), it is probably a good idea to keep processing and header application a two-step process.
- `proc --apply` in addition tries to compute header caches and applies them. This could be the default operation when header computation is fast
- `proc --reprocess` recreates caches (without this option, cached headers are never touched). You want this option if you found a bug in your _getHeader method and need to to recompute all the headers.

- `proc --reheader --apply` replaces processed headers on the source files. This is necessary when you want to apply re-processed headers. Without --reheader, to header that looks like it is "fixed" (according to your _isProcessed code) is ever touched.

Admittedly, this logic is a bit convolved, but the fine-grained manipulation intensity is nice when your operations are expensive.

By default, files for which the processing code raises exceptions are ignored; the number of files ignored is shown when procmain is finished.

If you want to run more than one processor over a given dataset, you will have to override the headerExt class attribute of your processors so all are distinct. By default, the attribute contains ".hdr". Without overriding it, your processors would overwrite the other's cached headers. However, that's usually not enough since on --apply only one header would win. One way of coping is by always applying one processor before running the next. Another could be the use of keepKeys (see below).

By the way, if the original FITS header is badly broken or you don't want to use it anyway, you can override the _getHeader(srcName) -> header method. Its default implementation is something like:

```
def _getHeader(self, srcName):
  return self._mungeHeader(srcName, self.getPrimaryHeader(srcName))
```

The getPrimaryHeader(srcName) -> pyfits header method is a convenience method of FITSProcessors with obvious functionality.

## Header Selection

Due to the way pyfits manipulates header fields without data, certain headers must be taken from the original file, overwriting values in the cached headers. These are the headers actually describing the data format, available in the processor's keepKeys attribute. Right now, this is:

```
keepKeys = set(["SIMPLE", "BITPIX", "NAXIS", "NAXIS1", "NAXIS2",
    "EXTEND", "BZERO", "BSCALE"])
```

You can amend this list as necessary in your _createAuxiliaries method, most likely like this:

```
self.keepKeys = self.keepKeys.copy()
self.keepKeys.add("EXPTIME")
```

You will have to do this if you have more than one processor (using headerExt) and want to be able to apply them in any sequence. This, however, is not usually worth the effort.

Since these operations may mess up the sequence of header cards in a way that violates the FITS standard, after this the new headers are sorted. This is done via fitstools.sortHeaders. This function can take two additional functions commentFilter and historyFilter, both receiving the card value and returning True to keep the card and False to discard it.

Processors take these from like-named methods that you can override. The default implementation keeps all comments and history items. For example, to nuke all comment cards not containing "IMPORTANT", you could define:

```
def commentFilter(self, comment):
  return "IMPORTANT" in comment
```

## Scanned Plates

For scanned plates, the plate archive standard proposes a fairly large and standardised set of headers. DaCHS supports you in generating those with its FITS header template system (that's designed to enable other such templates; see gavo.helpers.fitstricks for how to write these; see also `registerTemplate` in there if you make new templates).

The central funtion is the `makeHeaderFromTemplate` function from `gavo.helpers.fitstricks`. This receives

- a template, which is essentially a sequence of card definitions,
- optionally the `originalHeader`; cards not managed by the template occurring this header will be appended to the new, templated header.
- keyword arguments corresponding to the header values.

The plate archive standard is suppported through `WFPDB_TEMPLATE`. A processor using it could look like this:

```
from gavo.helpers import fitstricks
from gavo import api

class PAHeaderAdder(api.HeaderProcessor):

  def _createAuxiliaries(self, dd):
    # read the observation log from somewher in the resdir
    # it's usually a good idea to use a DaCHS parser for that, but
```

```
    # let's keep this example straightforward.
    self.platemeta = {}
    colLabels = ["plateid", "epoch", "emulsion", "observer", "object"]

    with open(os.path.join(dd.rd.resdir, "data", "platecat.tsv") as f:
      for ln in f:
        rec = dict(zip(colLabels, [s.strip() for s in ln.split("\t")]))
      self.platemeta[rec["plateid"] = rec

  def _isProcessed(self, srcName):
    # typically, check for a header that's not in your input files
    return "OBSERVER" in self.getPrimaryHeader(srcName)

  def _mungeHeader(self, srcName, hdr):
    plateid = hdr["PLATEID"] # more typically: grab it from srcName
    thismeta = self.platemeta["plateid"]

    # you'll usually want to drop some junky headers from hdr
    del hdr["BROKEN"]

    return fitstricks.makeHeaderFromTemplate(
      fitstricks.WFPDB_TEMPLATE,
      originalHeader=hdr,
      DATEORIG=api.jYearToDateTime(float(thismeta["epoch"])).isoformat(),
      EMULISON=thismeta["epoch"],
      OBSERVER=thismeta["observer"],
      OBJECT=thismeta["object"],
      ORIGIN="Contant")
```

– and so on with the whole host of headers defined by the plate archive standard; just just the header names as given there, replacing dashes with underscores (e.g., `RA-DEG` becomes `RA_DEG` for the keyword argument).

# Astrometry.net

## Calibration using Astrometry.net

If you have uncalibrated (optical) images, you can try to automatically calibrate them using astrometry.net. The DC software comes with an interface to it in helpers.anet, and the file processing infrastructure is what you want to use here.

You probably want to inherit from AnetHeaderProcessor, more or less like this:

```
from gavo import api

class MyProcessor(api.AnetHeaderProcessor):
  sp_indices = ["index-4215"],
  sp_lower_pix = 0.1
  sp_upper_pix = 0.2
  sp_endob = 50
```

```
  def _mungeHeader(self, srcName, hdr):
    vals = {
      "OBJTYP": "Galaxy",
      "OBSERVAT": "HST",
      ....}
    return fitstricks.makeHeaderFromTemplate(
      fitstricks.WFPDB_TEMPLATE,
      originalHeader=hdr, **vals)
```

The class attributes starting with `sp_` are parameters for the solver. The anet module docstring explains what is available. The endob parameter is important on larger images because it instructs anet to give up when no identification has been possible within the first endob objects. It keeps the solver from wasting enormous amounts of time on potentially thousands of spurious detections, e.g., on photographic plates.

Overriding `_mungeHeader` lets you add header cards of your own. The default is again to just return the header. Here, we're using DaCHS FITS templating engine (which is generally a good idea and deserves more documentation; please complain if you're reading this and missing docs).

Note that the `_mungeHeader` code can run independently of the (potentially time-consuming) astrometry.net code. Run the processor with `--no-anet --reprocess` to re-create the headers computed there without re-running astrometry.net.

If you want to use SExtractor for source extraction, add a sexControl class attribute. If it is empty, extraction will be done using some default parameters. You can add more (refer to the sextractor manual):

```
sexControl = """
  DETECT_MINAREA   100
  DETECT_THRESH    8
  SEEING_FWHM      1.2
  """
```

-- do not change CATALOG_TYPE, CATALOG_NAME, and PARAMETERS_NAME.

You can even filter what sextractor has obtained. To do that, define and `objectFilter` method (in addition to the `sexControl` attribute):

```
import numpy
from gavo.utils import pyfits
...

def objectFilter(self, inName):
```

```
  """throws out funny-looking objects from inName and throws out objects
  near the border.
  """
  hdulist = pyfits.open(inName)
  data = hdulist[1].data
  width = max(data.field("X_IMAGE"))
  height = max(data.field("Y_IMAGE"))
  badBorder = 0.3
  data = data[data.field("ELONGATION")<1.2]
  data = data[data.field("X_IMAGE")>width*badBorder]
  data = data[data.field("X_IMAGE")<width-width*badBorder]
  data = data[data.field("Y_IMAGE")>height*badBorder]
  data = data[data.field("Y_IMAGE")<height-height*badBorder]

  # the extra numpy.array below works around a bug in several versions
  # of pyfits that would write the full, not the filtered array
  hdu = pyfits.new_table(numpy.array(data))
  hdu.writeto("foo.xyls")
  hdulist.close()
  os.rename("foo.xyls", inName)
```

Just make sure to rename the result you come up with to whatever is passed in in `inName`.

Note, incidentally, that we take pyfits from gavo.utils. You should never import pyfits directly, since this may pull in pyfits in a way incompatible with what the rest of the DC software expects.

If you need more control over the parameters of astrometry.net, override the _runAnet method. Its default implementation is:

```
def _runAnet(self, srcName):
   return anet.getWCSFieldsFor(srcName, self.solverParameters,
    self.sexControl, self.objectFilter, self.opts.copyTo,
    self.opts.beVerbose)
```

So, if you had an attribute `sexControl_in` containing `DETECT_MINAREA %d`, you could do something like:

```
def _runAnet(self, srcName):
  for minArea in [300, 50, 150, 800, 2000, 8000]:
    try:
      self.sexControl = self.sexControl_in%minArea
      res = api.AnetHeaderProcessor._runAnet(self, srcName)
      if res is not None:
        return res
    except ShellCommandFailed:  # Ignore failures
      pass
  raise anet.ShellCommandFailed("No anet parameter worked", None)
```

Since astrometry.net spews out oodles of headers that may not be of huge interest to later users, the AnetHeaderProcessor implements comment and history filters. It is probably a good idea to re-use those even when you want filters of your own. This could look like this:

```
def historyFilter(self, value):
  if "changed" in value:
    return True
  if "left" in value:
    return False
  return api.AnetHeaderProcessor.historyFilter(self, value)
```

To skip computation on some "known bad" cases without overriding `_getHeader`, you can override `_shouldRunAnet(srcName, hdr)`. If you return false there, no astrometric calibration is attempted.

### Analyzing calibration failures

If astrometry.net fails to solve fields, you can get a copy of the "sandbox" in which the helpers.anet runs the software by passing your processing script the --copy-to=path option. Caution: If the directory path already exists, it will be deleted. If you run your processor with --bail, it will stop at the first non-solvable field.

Going to the sandbox directory, you will find at least:

- img.fits -- a copy of the input file
- backend.cfg -- a configuration file for solve-field, in particular containing the indices to be used.
- img.axy -- the extracted source positions in a binary FITS table
- lastCommand.log -- A log of what the commands ran spat out.

There may also be sextractor control files, images generated by solve-field, and more.

To figure out what's wrong, the first stop should be lastCommand.log. In particular, it shows the command lines of the programs executed, so you can modify them to try and figure out things (but the command lines do not include quoting; this is usually harmless for what the astrometric calibration does, but you have been warned).

To rerun SExtractor, say:

```
sextractor -c anet.control img.fits
```

You should sort the result by magnitude, since that's what anet's solver expects. In the normal case, you can do this like so:

```
$ANET_PATH/tabsort  MAG_ISO img.axy out.axy && mv out.axy img.axy
```

To get an idea what the source extraction has done, you can try anet's plotxy. You could use anet's solve-field, but this probably will not reflect what is actually going on within the helper, in particular not if sextractor is in use.

Instead, do something like:

```
gm convert -flip -scale 6.25% img.fits pnm:- | $ANET_PATH/plotxy -I - -i img.axy -C red -P -w 2
```

We use gm (from GraphicsMagick) here since netpbm's fitstopnm has issues with large files. You will want to use different scales for larger or smaller images both in gm convert's scale and plotxy's -S option, or leave them out altogether, like this:

```
gm convert -flip img.fits pnm:- | $ANET_PATH/plotxy -I - -i img.axy -C red -P -w 2  -N50 -s circ
```

for smaller images. Also, change the argument to -N if you change endob in the solverParameters to get an idea which objects are actually looked at.

### What to Try

In the case of calibration failures you may play around with SExtractor's parameters DETECT_MINAREA and DETECT_THRESH. This is done by running:

```
calibrate.py --minarea=MINAREA --detectthreshold=DETECTTHRESH
```

DETECT_THRESH refers to the detection threshold (in sigma) above the local background. A group (of pixels) is formed by a number of pixels connected to each other whose values exceed the local threshold. DETECT_MINAREA sets a lower bound on the number of pixels a group should have to trigger a detection.

The default values used for the calibration are MINAREA $= 300$ and DETECT-THRESH $= 4$. In some cases it is useful to decrease the MINAREA parameter and to increase the detection reliability by increasing the threshold value, e.g.:

```
calibrate.py --minarea=10 --detectthreshold=6
```