# Star removal on SDSS images

Dino Bektešević

Supervisor: izv.prof.dr. Dejan Vinković

Split, October 2013

Bachelor Thesis in Physics

Department of Physics
Faculty of Science
University of Split

**Abstract**

Sloan Digital Sky Survey (SDSS) is an astronomical survey program dedicated to categorization of all detected astronomical objects. Here I present my work on developing software tools for analyzing the SDSS image database in search for trails left by objects that crossed the field of view of the telescope with different speeds then the background sky. Such lines have not being detected by SDSS imaging pipeline so far and can be of scientific interest if attributed to meteors.

# Acknowledgements

I like to acknowledge mr. Erin Sheldon for extraordinary effort he put into transcribing IDL packages to Python modules that made SDSS data analysis free and open source. I would also like to extend my gratitude to him, for patience and good will to respond whenever I needed help.

# Contents

# Listings

# List of Figures

# Chapter 1

# Introduction

The Sloan Digital Sky Survey (SDSS) is a high resolution, deep space, multi-filter imaging and spectroscopic redshift survey dedicated to categorization and measurement of all detected objects and their characteristics. SDSS is separated into 3 different phases (SDSS-I, 2000-2005; SDSS-II, 2005-2008; SDSS-III, 2008-2014), with different goals but has always maintained all data publicly available and regularly issued. So far there has been 10 different data releases (DR) with a major database and data quality overhaul made in DR9 to correct for astrometry systematic errors (see Christopher P. Ahn et al., 2012), so extra caution should be used if dealing with older DRs.

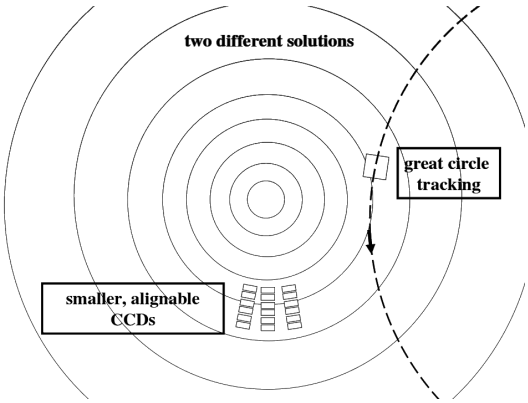


Figure 1.1: Star trails and CCD camera positions.

SDSS uses a dedicated 2.5-m wide-angle optical telescope, in design most similar to Maksutov-Cassegrain type, located at Apache Point Observatory (APO) in Sunspot, New Mexico. Telescope records the sky using drift scan imaging method. In essence telescope is fixed while, because of the apparent motion of the sky, long strips of the sky are recorded. Efficiency of a constantly recording CCD camera when it comes to imaging large parts of the sky, removal of systematic tracking error and consistent astrometry are the biggest advantages of this particular method.

However, compared to tracking method, drift scanning has a disadvantage when it comes to imaging sky far from the equator. At high declinations curvature of star's path become obvious and variation of drift rate across the CCD camera causes distortions (see Figure-1.1). This problem is solvable by using small alignable

CCD cameras or by great circle tracking. As a primary solution to distortions at high declinations SDSS telescope uses great circle tracking and therefore requires choreographed changes in right ascension (RA), declination (DEC) tracking rates and image rotation, I discuss SDSS great circle coordinate system in more detail in section 2.2.

The telescope's camera is made up of thirty CCD chips, each with a resolution of 2048x2048 pixels, totaling approximately 120 Megapixels. The chips are arranged in five rows of six chips (see Figure-1.2). Each row has a different optical filter (u, g, r, i and z) with average wavelengths of 355.1, 468.6, 616.5, 748.1 and 893.1nm and limiting apparent magnitudes of 22.0, 22.2, 22.2, 21.3 and 20.5, respectfully. To reduce noise the camera is cooled to 120 °K (about −80 °C) by liquid nitrogen. For a more detailed technical report see Gunn, Rockosi et al. (1998).

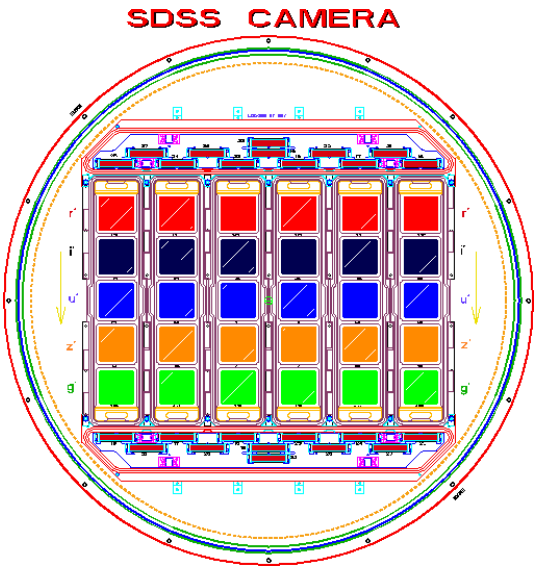


Figure 1.2: Schematic of SDSS CCD camera

Because of the drift tracking method, data are collected as a continuous tapestry. For this paper and for general ability to navigate SDSS database it helps to understand the terms SDSS database uses to describe the data. In the coordinate system of the SDSS CCD camera shown in Figure-1.2 the sky drifts downwards. Each such drift scan is referred to as a **run** and there is an associated integer identifying the run. For science quality runs, the lowest run number is 94, and the highest is 8162. Layout of filters are such that an imaging run results in six continuous images associated with the six columns of CCDs. Each of these six are known as **camcols** and are numbered from 1 to 6. Each camcol is 2048 pixels wide (the width of the CCDs). The gaps between the camcols in most cases are filled by another overlapping run. Each camcol is artificially broken up into a series of overlapping **fields**, each 1489 pixels long. The term artificially is of importance because as a consequence it carries that in a single CCD chip there can be multiple fields. Fields refer to all 5 images in all 5 filter bands and are the basic unit for the photometric pipeline. Astrometrical pipeline, however, processes a **frame**. Frame refers to an image uniquely identified by its run-camcol-filter designation. They are designed to be overlapping by a 128 pixels within a run so that objects are not mis-detected due to being too close to the edge of a field. Finally, there have been multiple

reprocessing of the data over the years[1]. Each reprocessing, called a **rerun**, has been denoted by an integer, the first being rerun 0 and the latest being rerun 301. Each rerun consists only in a change to the photometric pipeline, not to the data itself. Important thing to mention as well is that all image data is in FITS format and is accessed through Science Archive Server (**SAS**) while all extracted data is accessed over Catalog Archive Server (**CAS**)[2].

Main goal of this paper is to use the SDSS data for the purpose of extracting images with "lines". These "lines" are left behind by objects passing with different speed then that of the sky. This includes living creatures, like bugs and bats, and inanimate ones, such as car headlights, asteroids, comets or meteors. CCD related phenomena such as blooming or ghosts also leaves lines. I am interested only in lines most likely left on the image by meteors, and will refer to such lines as **trails**.

Meteors are solid objects ranging in size from microns to meters that burn out upon entering Earth's atmosphere. They are separated into two categories depending on their source of origin: sporadic and meteor showers. Sporadic meteors are solid objects "floating" through the Solar system with no apparent body of origin while meteors from meteor showers are largely attributed to orbits of comets[3] (Jenniskens P., 2006). Determining orbits of meteors in a meteor shower enables us to determine an approximate path of a comet thus allowing us its detection. Due to relatively short lives of comets (Whitman, Morbidelly, Jedick, 2006) most of today meteor showers are but small remnants of a dead comet. However, it is estimated that 6% of the dangerous population of near Earth asteroids (NEO) are comet cores no longer experiencing out-gassing (Whitman, Morbidelly, Jedick, 2006). This fact makes detecting meteor showers of great importance. International meteor organization (IMO)[4] has been collecting visual meteor observation since 1988. In recent times projects like Croatian meteor network (HMM)[4] have started conducting video observation. As mentioned above the limiting apparent magnitudes of CCD camera in SDSS ranges from 20.5 to 22.2 magnitudes which is incredibly dim. In comparison consider that limiting apparent magnitude of human eye in best conditions is 6, limiting apparent magnitude of a video camera is even less ($\sim$4), and that of the Hubble space telescope for visible light is only 12. It follows that trails must have been left by very small objects not capable of saturating the entire image, which makes their detection very valuable. Discovering that there is a faint meteor shower still not detected grants us a better view of distribution of dust, as well as some of its properties such as grain size or density along Earth's orbit and of course valuable information about the population of comets through history[5].

[1] Older literature may refer to deprecated nomenclature (scanline, strip, stripe, field and run)

[2] see `http://www.sdss3.org/dr8/data_access.php` for other download options.

[3] see also: Whipple F. L. (1951)

[4] IMO: urlhttp://www.imo.net/; HMM: `http://www.astro.hr/hmm/`

[5] A long standing debate about the origin of life on Earth proposes that comets played a crucial role by "delivering" water Fernandez J. A. (2006) and provoking shock synthesis of amino acids Martins, Price, Goldman et al. (2013) makes this an interesting information to know

# Chapter 2

# Data Access and Preparation for Line Detection

SDSS database already contains approximately 60TB of data and each year of operation gathers about 5TB of new raw data (Gray, Szalay, Thakar et al., 2012). The main goal of this paper to write a program that saves frames with detected trails and rejects others for the entire SDSS database. Hence, the obvious restriction is the execution time. Furthermore, a high detection rate, that is detection confidence, is desirable. For about 9.5 million images even 80% detection confidence leaves 760 thousand images in need of re-checking manually. That means that the method of trail detection used should be as resistant to "noise" and outliers interference as possible. It is unrealistic to expect a noise-proof method. Thus, noise-reducing algorithms should be considered as a processing tool as well. Of course, a high level of program modularity should also be present in cases where another, more beneficial, method of image processing is found. A quick replacement should be made in such a case, without the need of a major program overhaul. Modularity implies the need for a procedural or object oriented programming paradigm. The need for editing large amount of files also points to a higher level programming language and finally for convenience reasons programming language used should be supported as much as possible.

SDSS, officially, supports and offers solutions only for IDL[1] programming language which comes with a hefty price. After careful considerations and a brief excursion into C++ and NASA written library fitsio[2], I have settled on Python. Python is a dynamically and strongly typed, completely object oriented programming[3] language with high interactive capabilities. Because Python is distributed as open source, it is heavily supported as well. Neat consequence of such rigorous object orientation is seen in it is interactive abilities, where, once a program is

[1] `http://www.exelisvis.com/ProductsServices/IDL.aspx`

[2] `heasarc.gsfc.nasa.gov/fitsio/` Page is currently down due to lack of agreement on funding and "Obamacare"

[3] To the extent that even integers are treated as objects!

compiled, all functions and classes are imported as a stand alone objects callable independently from within IDL. This provides a steep learning curve as well as great debugging and introspection possibilities.

I will use Python in conjecture with three independent modules, equivalent to a library in C++, all of which were written by Erin Sheldon. SDSSPY[4] is a set of tools for working with SDSS data and it was necessary to implement a few changes in the original code for purposes of this paper (see section 2.1). SDSSPY depends on ESUTIL[5] module for coordinate calculation. A special module for handling FITS files is needed as well. Luckily mr. Sheldon already wrote Python wrappers for aforementioned NASA's fitsio library with the same name[6]. These modules depend extensively on NUMPY and SCIPY. NumPy is a powerful scientific open source module for numerical calculations and among other things contains the famous powerful n-dimensional array object: ndarray. SciPy is yet another powerful open source tool designed for numerical calculations, predominantly integration and optimization routines. I would recommend, instead of using NumPy and SciPy separately, to rather use SciPy Stack which consists of various modules including NumPy, SciPy core, Matplotlib and IPython. This also means that because NumPy and SciPy still do not fully support Python versions past 3.0, an older Python version is needed. Last stable version is Python2.7 and that is the used version in this paper.

Now that I posses the necessary tools, a more detailed approach to the image analysis can be made. First step is to remove the most obvious interference - stars themselves. To do that I need to locate their position on the image. I can retrieve star coordinates in equatorial system through CAS. With SDSSPY module, more specifically Astrom class, which is dedicated to coordinate conversion, I can retrieve coordinates in pixels on the image itself. I would need to determine star's approximate size on the image to know how much of the image to delete. After I have gotten rid of all the stars what is left is noise and trails. I defined noise as all isolated, dim pixels. However, I cannot simply go through the image and delete all dim pixels because I would lose all the dim trails. I have decided to, due to peculiarities of FITS format, set all negative pixels to 0 and then do a histogram equalization. Next step would be to search for isolated pixels and remove them, and only after this step I search if an image contains trails or not. I have decided on using Hough transform for locating lines in images. In following chapters I will detail some interesting programming solutions and provide a case study of frame-i-002888-1-139.fits as an example of progress so far.

[4] `http://code.google.com/p/sdsspy/`
[5] `http://code.google.com/p/esutil/`
[6] `https://github.com/esheldon/fitsio`

## 2.1 Changes introduced to SDSSPY

SDSSPY module consists of 10 different modules: astrom, atlas, family, files, flags, util, yanny and window. Modules I am interested in are *files* and *astrom*.

Files module is used for creating filenames based on SDSS database default nomenclature frame-filter-run-field.fits. However files module has no support for creating CAS SDSS names. CAS files are retrievable by a SQL query and are returned in a specified form. For simplicity I retrieve the catalog data in a comma separated value (CSV) file. Upon closer inspection of files module it is obvious that an instance of a class FileSpec uses a function expand_sdssvars to expand variables sent to the FileSpec object and then through string manipulation provide a filename. File locations are generally defined relative to certain root directories, which are themselves defined by environment variables, e.g. $ PHOTO_REDUX. These locations are described for each file type in sdssFileTypes.par which is, for a default install, located in usr/local/share. A short snippet of the sdssFileTypes.par is provided below with line 16 (CSVCoord) already edited in.

Listing 2.1: sdssFileTypes.par

```
1  typedef struct {
2      char ftype[50];
3      char dir[255];
4      char name[255];
5      int  ext;
6  } FILETYPE;
7
8  FILETYPE CSVCoord $BOSS_CAS CAS-$RUNSTR-$COL-$FIELDSTR.csv 0
9  FILETYPE sdssMaskbits $SDSSPY_DIR/share sdssMaskbits.par -1
10 FILETYPE sdssFileTypes $SDSSPY_DIR/share sdssFileTypes.par -1
11 FILETYPE runList $PHOTO_REDUX runList.par -1
```

Note that $RUNSTR means a padded run string, e.g. 000756 whereas $RUNNUM is unpadded. $COL means an unpadded version of camcol, $FIELDSTR means a 4-padded field string, there are no unpadded versions for calcom and field, and finally $FILTER stands for filter designation ('r', 'g', etc...) I was unable to determine the meaning behind extensions, numbers ranging from -1 to 6. After contacting mr. Sheldon he clarified those are error extensions for files that have to be determined before a program runs and therefore not of grave importance to me.

Second thing to edit in SDSSPY module is the astrom module. Astrom module is a container for Astrom class that primarily deals with various coordinate conversions. Changes introduced in this module are purely for error tracking, because, as I discovered, not all coordinate conversions succeed (see section-2.2). Out-of-context edited code snippet is presented (see Listing-2.2), just to point out the exact changes made. Err_ra and err_dec are global variables defined outside the Astrom class and are used to store values of RA and DEC which failed to converge. Code snippet is part of munu2pix method defined in Astrom class.

Listing 2.2: astrom.py

```
if ier != 1:
    raise ValueError("Fsolve() could not find convergence for ra:
        {}, dec: {} initial guess: row:{}  col:{} final guess: row
        :{} col:{} field:{} filter:'{}' mu:{} nu:{}".format(err_ra
        , err_dec, row_guess, col_guess, rowcol[0], rowcol[1],
        field, filter, mu, nu))

if are_scalar:
   row=row[0]
   col=col[0]

return row,col
```

## 2.2 Coordinates input and conversion

As I mentioned in section-2.1, I retrieve Coordinates from an SQL query made to CAS, as seen in Listing-2.3. The retrieved file is then read into a dictionary by using an inbuilt Python module named csv, as seen in Listing-2.4. Python dictionary is similar to lists except each input in the list can have attributes appended to it. Thus a dictionary with input "persons" can hold additional attributes (such as "age", "height", etc.), for each instantiated "person". Originally I used my own parser for CSV files, however that required a lot of maintenance. For example, if I should decide not just to input objects ra, dec and magnitude in filters, but other attributes as well, then I have to add them manually. In comparison, the csv module names the attributes based on the first row of the file, so I have decided against using my parser.

Listing 2.3: Example of an SQL query

```
SELECT p.objid,p.ra,p.dec,p.u,p.g,p.r,p.i,p.z
FROM PhotoObj p
WHERE p.run = 2886 AND p.camcol = 4 AND p.field = 146
```

CSVRead is still a function (see section-3.0.1) and is called by passing a path to the .csv file. This is where change of sdssFileTypes.par (see Listing-2.1) comes in handy. Because I want this program to run automatically on a large lists of files it is unpractical to manually define a path for each CSVCoord file. By editing the the files module I have enabled the function sdsspy.filename(fileType, run, camcol, filter, frame) to dynamically create new filenames for CAS CSV files.

Listing 2.4: CSV reader

```
"""Example of how to call CSVRead function"""
Coord = CSVRead(sdsspy.files.filename('CSVCoord', run=_run,
                        camcol=_camcol, field=_field))
.................................
def CSVRead (path):
    """
    Defines a function that reads CSV file given by
    (str) path into a list of dictionaries.
    Returned list is arranged as
        {[ra:, dec:, u:, g:, r:, i:, z:],
            ...}.
    """
    labels=['ra', 'de', 'u', 'g', 'r', 'i', 'z']
    read = csv.DictReader(open(path), labels, delimiter=',',
        quotechar='"')
    lines = list()
    for line in read:
        lines.append(line)
    return lines
```

The SDSS has two sets of coordinates which are specially designed for the survey geometry. The natural coordinate system to use for processing a given run is the great circle coordinate system for that stripe ($\mu$,$\nu$) in which the equator of the coordinate system is the great circle tracked by the scan. This great circle is inclined by $i = \nu + 32.5^\circ$ to the J2000 celestial equator, with an ascending node of $95^\circ$. Coordinate $\mu$ increases in the scan direction (east) and $\nu$ increases to the north. The second set is the Survey ($\xi$,$\eta$) system. This is a rotated and mirrored spherical coordinate system, where ($\xi$,$\eta$)=(0,90) corresponds to equatorial coordinates $(\alpha, \delta) = (275, 0)$ and ($\xi$, $\eta$)=(57.5,0) corresponds to $(\alpha, \delta) = (0, 90)$. Also, $\eta$ runs only from $-90^\circ$ to $90^\circ$ while the "back" of the sphere is covered by $\xi$, which runs from $-180^\circ$ to $180^\circ$ which is the opposite of expected. To quote Michael Blanton: "These conventions defy logic, and please don't blame me for them."[7]. An especially important approximation for narrow-field astrometry is the tangent-plane mapping (Keel B., 2006) given by equations:

$$\cot(\delta - \delta_0)\sin(\alpha - \alpha_0) = \frac{\xi}{\sin(\delta_0) + \eta\cos(\delta_0)} \tag{2.1}$$

$$\cot(\delta)\cos(\alpha - \alpha_0) = \frac{\cot(\delta_0) - \eta\sin(\delta_0)}{\sin(\delta_0) + \eta\cos(\delta_0)} \tag{2.2}$$

Tangent-plane mapping is basically a projection of part of the celestial sphere outward onto a plane tangent to it at a reference point ($\alpha_0$, $\delta_0$). Luckily for a very small coordinate differences I am allowed to approximate the change of the ($\xi$,$\eta$) in reference to pixel coordinates by a linear transformation where the general equations for tangent-plane mapping are cut drastically short. To that point SDSS offers

[7] http://cosmo.nyu.edu/~mb144/tiling_docs/surveycoords.html

coordinate information of a reference point, pixel scale and pixel orientation of an image in its FITS header. Even more data is available in the photoField.fits files, such as PSF function, distortion factors, error estimations for ($\mu$,$\nu$) coordinates etc... Values of interest are given by keywords:

**CTYPE1** describes type of approximation for following variables

**CTYPE2** describes type of approximation for following variables

**CUNIT1** describes unit type, usually degrees

**CUNIT2** describes unit type, usually degrees

**CRPIX1** gives column pixel coordinate of reference pixel

**CRPIX2** gives row pixel coordinate of reference pixel

**CRVAL1** gives RA of reference pixel

**CRVAL2** gives DEC of reference pixel

**CD1_1** gives change of RA in degrees per column pixel

**CD1_2** gives change of RA in degrees per row pixel

**CD2_1** gives change of DEC in degrees per column pixel

**CD2_2** gives change of DEC in degrees per row pixel

Coordinate conversion is thus made in 2 steps. First I need to convert equatorial coordinates ($\alpha$, $\delta$) from CSVCoord file into great circle coordinates using equations 2.3 and 2.4 and secondly, with the new acquired great circle coordinates, I need to acquire pixel coordinates $(x, y)$ using equations 2.5 and 2.6.

$$\cot(\delta)\sin(\alpha - CRVAL1) = \frac{\xi}{\sin(CRVAL2) + \eta\cos(CRVAL2)} \tag{2.3}$$

$$\cot(\delta)\cos(\alpha - CRVAL1) = \frac{\cot(CRVAL2) - \eta\sin(CRVAL2)}{\sin(CRVAL2) + \eta\cos(CRVAL2)} \tag{2.4}$$

$$\xi = CD1\_1(x - CRPIX1) + CD1\_2(y - CRPIX2) \tag{2.5}$$

$$\eta = CD2\_1(x - CRPIX1) + CD2\_2(y - CRPIX2) \tag{2.6}$$

Unfortunately, given functions are in their implicit form. If I pick a pixel on the image, it is easy to acquire survey coordinates from equations 2.5 and 2.6 and easier still to recover RA and DEC coordinates from 2.3 and 2.4. That is, the pixel to equatorial conversion follows directly from the equations, but the inverse conversion does not and the solution has to be found numerically. Solutions to those equations are its roots, which can be found using scipy.optimize.fsolve() function

that requires an initial guess of (x,y) and then iterates their values until a convergence or precision criteria is met. I have commented only particular lines of interest from conversion code in a code snippet Listing-2.5 because of its length. For context you can locate the full function in appendinx-A

Listing 2.5: Crucial lines from GC to PIX conversion function

```
def munu2pix(self, field, filter, mu, nu, color=0.3):
   import scipy.optimize
   mu,nu,are_scalar=get_array_args(mu,nu,"mu","nu")
   color=self._get_color(color, mu.size)

   #since munu2pix is a class instance in its load
   #function a trans variable is defined containing all
   #fields from the FITS header
   trans=self.trans
   w=self._get_field(field)
   fnum=self._get_filter_num(filter)
   if 'f' in trans.dtype.names:
        fname='f'
   else:
        fname='ff'

    # saving data read from photoField.fits file for
    #the scipy.optimizer module
    #fd is a list that contains aforementioned data [CRVAL etc...]
        fd={'a':trans['a'][w,fnum],
            'b':trans['b'][w,fnum],
    #....snipped for space saving reasons.....

    #initial best guess for (x,y) coord
    row_guess = ( mudiff*fd['f'] - fd['c']*nudiff )/det
    col_guess = ( fd['b']*nudiff - mudiff*fd['e'] )/det
    rowcol_guess=array([row_guess[i], col_guess[i]])

    for i in xrange(mu.size):
            self._tmp_color=color[i]
            self._tmp_munu=array([mu[i],nu[i]])
            rowcol_guess=array([row_guess[i], col_guess[i]])
            #fsolve() is a function that solves non-linear and
            #linear equations and systems of eq. for their roots
            rowcol, infodict, ier, msg = scipy.optimize.fsolve(
                self._pix2munu_for_fit, rowcol_guess, full_output=
                True)

    row[i] = rowcol[0]
    col[i] = rowcol[1]

    #part of astrom.py edited in Changes introduced to SDSSPY
    # also snipped for space saving reasons.....
```

Although the solution should always exist, one is not always found. In some cases the fsolve() function simply does not converge sufficiently fast or the solution to equations 2.5 and 2.5 seems to be unstable and the function excepts. Until the

changes in section 2.1 were not made, this caused the program to terminate, which is why that change is crucial. Python, as well as other higher level programming languages, has a class for error handling enabling a programmer to handle any exception that arises. So far I only print out these errors to determine the stability of the solution. Removing stars now becomes just a matter of repeating the same procedure for each star located in the dictionary of star coordinates:

Listing 2.6: For loop for star removal

```
for star in Coord:
        try:
            ra = float(star['ra'])
            de = float(star['de'])
            xy = conv.eq2pix(_field, _filter, ra, de)
            x, y=xy[0], xy[1]
            img[x-30:x+30, y-30:y+30].fill(0.0)
            #so far I have not yet implemented any kind
            #of star size estimation factors
        except ValueError as err:
            print err.message
            starConv_errnum+=1
            pass
    print "{} coordinate conversions failed. Total number of stars
         on the image: {}".format(starConv_errnum, len(Coord))
```

First comparison of the image with removed stars, shown in Figure-2.2, with the original frame-i-002888-1-139.fits, shown in Figure-2.1, can be made now. It is obvious that the line will be easier to detect now, that stars are removed. However, overall image is still very dim and, as you will see in section 2.3, it hides a lot of noise and other artifacts.

Figure 2.1: Original image

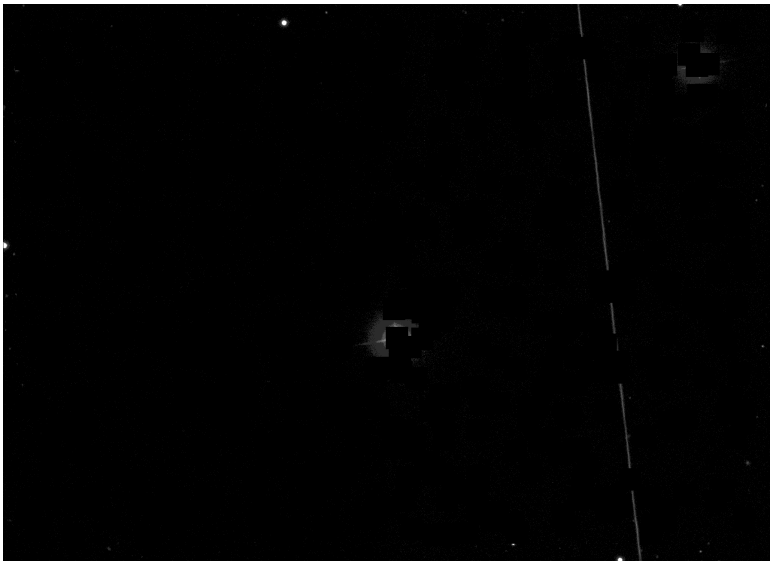

Figure 2.2: Result of star removal

## 2.3 Histogram equalization and noise reduction

Histogram equalization is a process of amplification of image contrast. For example, an image of dynamic range 255, in which pixels never achieve intensity values larger than 100, the entire dynamic range of 155 values is left unused. By equalizing histogram of such an image, highest intensity pixels will assume the values of 255 while other pixels will linearly distribute themselves by intensity depending on the value of the unused dynamic range. This step is necessary for noise reduction because I use a method for noise removal that detects if a pixel is neighbored by less than 3 other pixels with high intensities. First step is a construction of a histogram which is done using an inbuilt NumPy function histogram(). Cumulative distribution function (see equation-2.7) has to be found, which is basically just a histogram of probabilities that a pixel will obtain a particular intensity value. Inbuilt NumPy function cumsum() is used for that. Using a basic linear interpolation function, the general histogram equation 2.8 can be satisfied and new values for pixel intensities obtained. In equation-2.7 $p_x(i)$ stands for the probability of an occurrence of a pixel of level $i$ in the image, while in the general histogram (equation 2.8) $M$ and $N$ are the image dimension in pixels and $L$ is the number of gray levels that $cdf$ has been normalized to. For my example this reduces equation-2.8 to equation-2.9, where $cdf_{min}$ is often equal to 1.

$$cdf_x(i) = \sum_{j=0}^{i} p_x(j) \tag{2.7}$$

$$h(v) = round\left(\frac{cdf(v) - cdf_{min}}{(M * N) - cdf_{min}}(L-1)\right) \tag{2.8}$$

$$h(v) = round\left(\frac{cdf(v) - cdf_{min}}{3049472 - cdf_{min}} * 255\right) \tag{2.9}$$

End result of histogram equalization is shown in Figure-2.3. This step obviously exaggerates the trail which is beneficial but it also exaggerates noise.

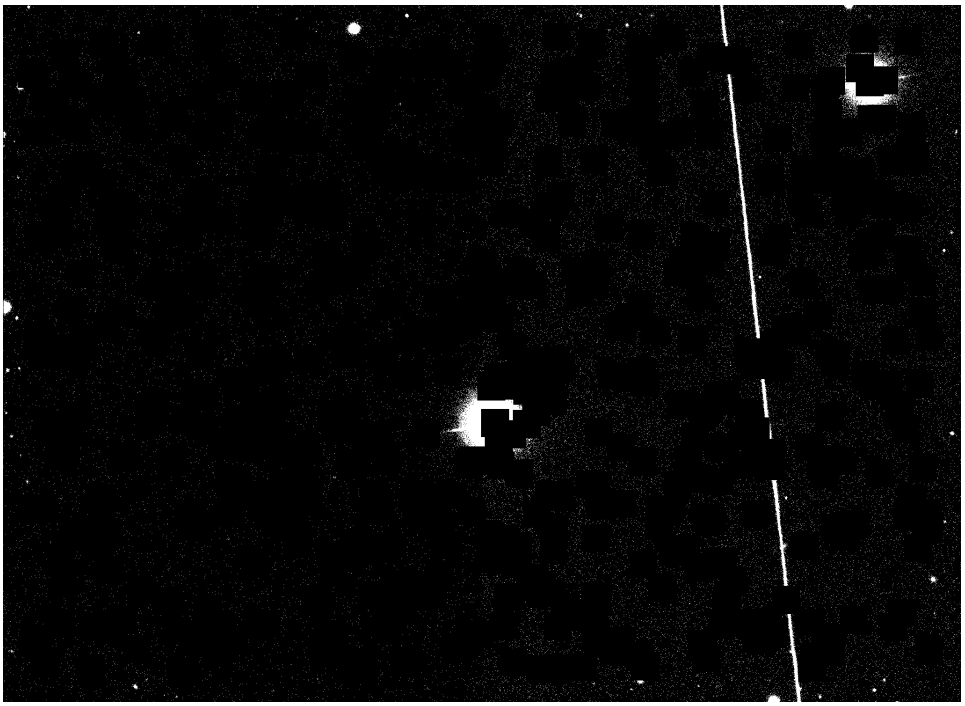

Figure 2.3: Result of histogram equalization

I find that it is easier to deal with noise then it was to detect faint trails. I have an implementation of my own histogram equalization function which functions perfectly. However, later I found that a openCV2 module function equalizeHist() works faster and better because it uses an adaptable area equalization with which areas of image that meet certain criteria are weight-equalized for the values of that local area, thus providing adaptable contrasting throughout the image. I discuss openCV2 module later on in chapter-3. I present you with my histogram equalization function in code-snippet Listing-2.7.

Listing 2.7: Function for histogram equalization

```
def histeq(image,nbr_bins=256):

   #create image histogram
   imhist,bins = histogram(im.flatten(),nbr_bins,normed=True)
   #create cumulative distribution function
   cdf = imhist.cumsum()
   #normalize to 256 bins
   cdf = 255 * cdf / cdf[-1]
   #use linear interpolation of cdf to find new pixel values
   im2 = interp(im.flatten(),bins[:-1],cdf)

   return im2.reshape(im.shape), cdf
```

Noise reduction is done with a "brute-force" method, meaning that I use a 3x3 matrix that I "drag" across the image storing values of the 8 pixels neighboring the central pixel. I calculate average value of that matrix and if average value is smaller than or equal to 85, central pixel is set to 0. The value 85 is a first hand approximation, based on a scenario where a central pixel has 3 neighboring pixels of intensity of 255 and the rest have zero value, which gives the matrix average of 85. I have experimented with other noise reduction algorithms, most of which were actually made for color images, and this one showed the best results for the least processing time spent. Code snippet 2.8 shows the code part that deals with noise reduction. It is integral part of a bigger function process_field() and not a stand alone function:

Listing 2.8: Noise reduction loops

```
n_x, n_y=equ.shape
for x in range (1, n_x-2, 1):
  for y in range(1, n_y-2, 1):
     if equ[x, y] != 0:
        li = [equ[x-1, y+1], equ[x, y+1], equ[x+1, y+1],
              equ[x-1, y],   equ[x, y],   equ[x+1, y],
              equ[x-1, y-1], equ[x, y-1], equ[x+1, y-1]]
         aver = numpy.average(li)
         if aver<=85:
           equ[x, y]=0
```

Results after noise reduction are shown in Figure-2.4.

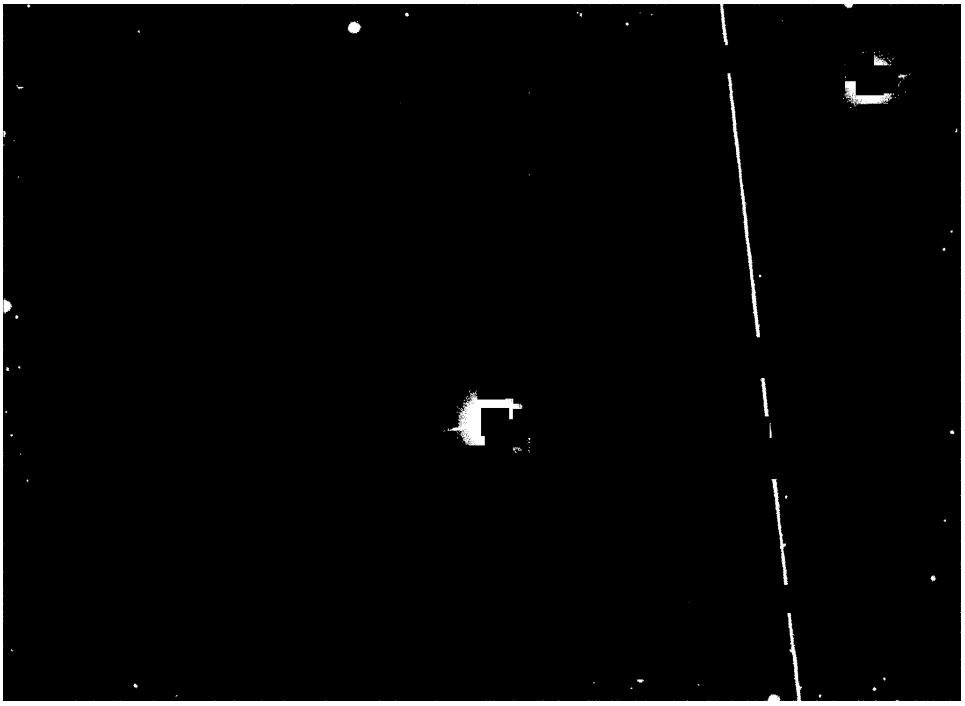

Figure 2.4: Result of noise reduction

# Chapter 3

# Line detection

Finally, with the newly prepared image-2.4 I have a proper basis for line detection. The image is represented in the code by a two dimensional ndarray object storing the new values 0-255 of pixel intensities, where 0 represents black, while all the other values, as a result of histogram equalization, are in the range $[1, 255]$. I have experimented with 2 different algorithms for line detection.

Random sample consensus (RANSAC) method is based on iterative method of randomly selecting a set number of points from all the data and fitting a line model to this set by assuming its points are inliers. The fit is then iteratively corrected by removing outliers and adding new points. Standard deviation and line parameters are saved and process is repeated until satisfying standard deviation is achieved or until all possibilities are exploited. If a satisfying solution is not found then the method returns a zero value. Although fairly easy to program, RANSAC method is very unstable for large images, as in my case, and it is very subjective to noise interference. Instances when 2 clustered high valued pixel sets exist somewhere on the image are usually always detected by RANSAC method because of their small standard deviation.

Hough transform is the second method I experimented with. In theory this method is almost impervious to noise and outliers interference. It consists of mapping the non-zero valued pixel coordinates to a polar coordinate system, as shown in Figure-3.1 taken from mr. Nabin Sharma's webpage[1].

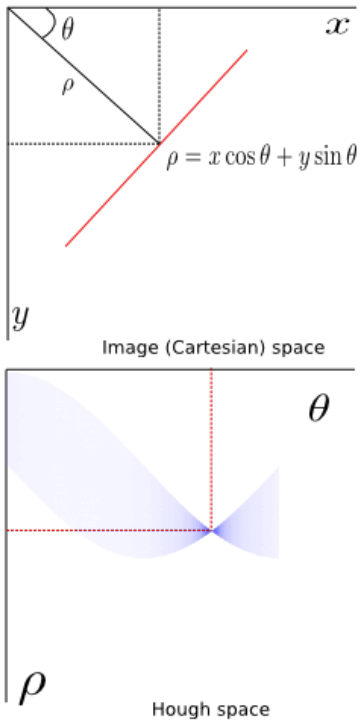


Figure 3.1: Cartesian and Hough space

In Cartesian coordinates a line is represented by equation-3.1, called the slope-intercept form. By parameterizing the line slope $m$ and intercept parameter $b$, we can rewrite the equation-3.1 as equation-3.2 and by rearranging that equation we get to the final form seen in equation-3.3.

$$y = mx + b \tag{3.1}$$

$$y = -\frac{cos(\theta)}{\sin(\theta)}x + \frac{r}{\sin(\theta)} \tag{3.2}$$

$$r = x\cos(\theta) + y\sin(\theta) \tag{3.3}$$

The transformed coordinate space which has $\theta$ and $r$ as its axes is called Hough space. A point in Cartesian space is mapped to a sinusoidal curve in Hough space. A set of points belonging to a line in Cartesian space get mapped to a set of sinusoids intersecting at a point in Hough space. So the problem of detecting a line in an image becomes a problem of detecting a point in Hough space. Once we detect the points in Hough space, we can do inverse transform, by using equation-3.2, to get the corresponding line parameters in Cartesian space. Basic Hough transformation algorithm is fairly easy to produce. Following the instructions[1] firstly I instantiate a 2D ndarray object, or accumulator, with zeros to store the values of $(r, \theta)$. Size of this array depends on the desired accuracy, described by values theta_res and rho_res. Obviously if I take that $\theta$ step is $1^\circ$ my array size will be $180$ columns large, while the, rows, values of $r$ are determined by the maximum possible distance on the image, that is its diagonal. Algorithm "walks" over the image and for each pixel with non-zero value constructs a sinusoidal curve by calculating values of $r$ for all allowed values of $\theta$ determined by step size and stored in 1D array named theta. Constructed sinusoidal curves are thus quantized per-pixel, binned. Next step is to increment by 1 all the pixels in accumulator array, through which the constructed sinusoid

[1] http://nabinsharma.wordpress.com/2012/12/26/linear-hough-transform-using-python/

[1],

passes through. Here a special trick of NumPy ndarrays comes in handy. It is possible to subtract an constant value (i.e. integer, double float) from a ndarray. Returned result is the entire ndarray subtracted member-by-member with the constant. Furthermore because Python is so drastically object oriented, as mentioned in chapter-2, an instantiated object is held as a true value and thus we are able to compare ndarrays holding boolean values with ndarrays holding variables of any other type. This is known as boolean indexing. Comparing two ndarrays of integers using $==$ logical operator will return a ndarray of True values for all the fields containing the same values, and False values for other fields. Furthermore the returned ndarray is a index representative of actual data, $1$ for True and $0$ for False, upon which further integer/float/double manipulation can be made. I provide basic examples in code snippet-3.1 which will, hopefully, shed more light on this matter:

Listing 3.1: Examples of strange ndarray behavior

```
1  # A and B are 1D ndarrays with some values
2  A
3  >>>>array([ 1, -2, -5.5, -7.3])
4  B
5  >>>>array([ 0, 0, 0, 0])
6  C = A == B
7  C
8  >>>>array([ False, False, False, False])
9  C = A > B
10 C
11 >>>>array([ True, False, False, False])
12 C.sum() #sum of all members
13 >>>> 1
14 5-C
15 >>>>array([ 4, 5, 5, 5])
```

In line 31 of code snippet-3.2 this is exactly what is done. First we create an ndarray for binning purposes. This array holds the absolute difference of rho, non-binned evenly spaced ndarray containing values of $r$, and the calculated value rhoval for current theta value. Secondly we find the minimum value of that same array. By comparing the values using $==$ logical operator we return a boolean ndarray containing values True for indices of, now binned, values of $r$. Operator nonzero() returns indices of all the True values in the ndarray and by the last command we increase the value of those indices in the accumulator array H which represents Hough space.

Listing 3.2: Hough transformation

```
import numpy as N
def hough_transform(img_bin, theta_res=1, rho_res=1):
  nR,nC = img_bin.shape #read the size of the image
  #creates a simetrically filled array holding binned
  #values of theta
  theta = N.linspace(-90.0, 0.0, N.ceil(90.0/theta_res) + 1.0)
  theta = N.concatenate((theta, -theta[len(theta)-2::-1]))

  #calculate the size of the diagonal, round it up
  D = N.sqrt((nR - 1)**2 + (nC - 1)**2)
  q = N.ceil(D/rho_res)
  nrho = 2*q + 1
  #creates an evenly spaces, non binned, ndarray
  #holding values for rho, size of the array is
  #determined by rho resolution (rho_res) and
  #the lenght of the diagonal
  rho = N.linspace(-q*rho_res, q*rho_res, nrho)
  #create the acumulator ndarrray
  H = N.zeros((len(rho), len(theta)))
  #list through the entire image
  for rowIdx in range(nR):
    for colIdx in range(nC):
      if img_bin[rowIdx, colIdx]:
        for thIdx in range(len(theta)):
          rhoVal = colIdx*N.cos(theta[thIdx]*N.pi/180.0) + \
              rowIdx*N.sin(theta[thIdx]*N.pi/180)
          #bin and return indices of appropriate rho
          rhoIdx = N.nonzero(N.abs(rho-rhoVal) == N.min(N.abs(rho-
              rhoVal)))[0]

          H[rhoIdx[0], thIdx] += 1
  return rho, theta, H
```

In Figure-3.3 I show the results of Hough transform for a test image shown in Figure-3.2. What remains is to detect local maxima and invert $(r, \theta)$ to $(m, b)$ using equation-3.2. However, I have to note that local maxima detection is a harder problem then it seems. In Figure-3.3 the two maxima are obvious, but notice in Figure-3.4 that maxima are much harder to notice. Because of binning of $(r, \theta)$ it is not necessary that the highest valued pixel is precisely the searched value of $(m, b)$ which can lead to significant deviation for large enough images. Presented algorithm was tested on multiple images and is very unstable. Execution times of presented code Listing-3.2 varies between 15 and 360 seconds and increases drastically with the size of the image, after all it is executing a number of higher level operations per image pixel. Resizing the image to smaller size shortens the execution time, however resizing the image to half its size inevitably deletes half of present information on it. In practice this means that dim and thin trails get deleted which is unacceptable.

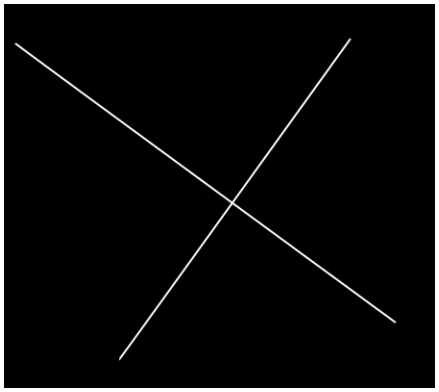

Figure 3.2: Test image for Hough transform

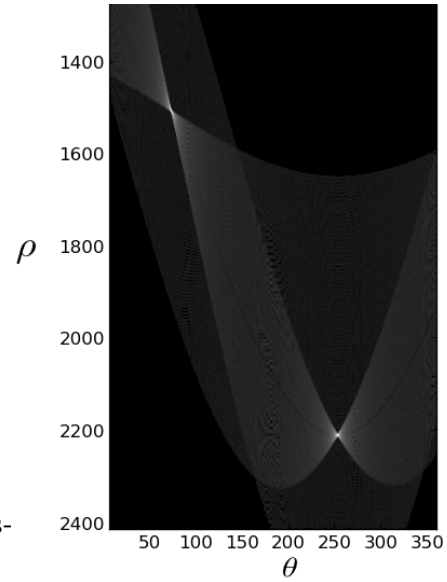


Figure 3.3: Hough space of test image

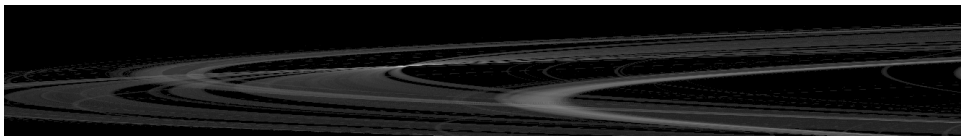

Figure 3.4: Hough transform of example frame

I turned for help to OpenCV (Open Source Computer Vision) - a Python library, originally written in C but ported to Python, Java, C++ and others. OpenCV is an immensely powerful computer vision library able to analyze real time video input and is used as a solution in industrial and commercial software. Available in this module are two different Hough transforms for lines, a probabilistic and "normal" Hough transform. Probabilistic Hough transformation is able to detect line segments, however it is slower and lacks in detection confidence. Due to the straightforward problem I am facing, I decided to use HoughLines function, which is both more precise, stable and faster. I have tried locating the original source code for the function, but despite OpenCV being an open source computer vision and machine learning software library, I was unable to locate a single repository providing me with full source code. What I could make out from the Python wrappers, function firstly detects edges on the image using Canny edge detection algorithm and only then proceeds to search for lines. OpenCV Hough line detection algorithm is extremely stable and fast. It eliminates the need for image resizing and is capable of processing the entire 2048x1489 pixel image in under 10 seconds. Results of OpenCV applied to Figure-2.4 is shown in Figure-3.5 for the first 5 maxima and in Figure-3.6 for the first detected maximum.

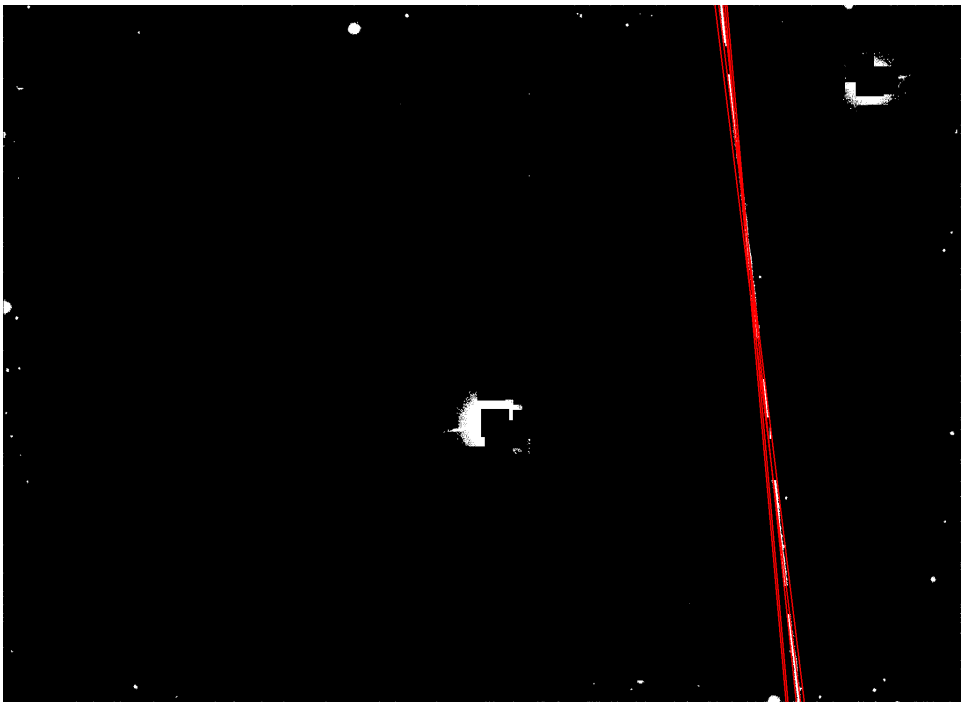

Figure 3.5: First 5 detected lines

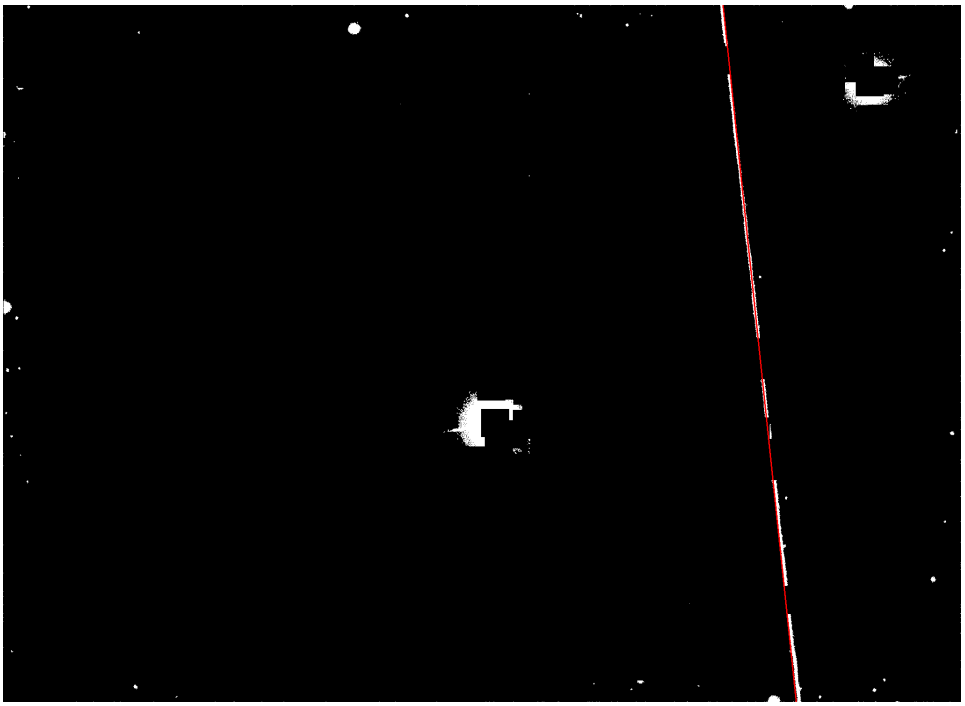

Figure 3.6: First detected line

Pixel intensity in Hough space is related to the length of the line on the image. Hough transform will always produce a line because I will always have residual pixels left after star and noise removal. Because some trails encompass only a short distance between two nearest image edges, I can not set a threshold value for pixels intensities in Hough space. Solution to this problem is obvious. I need to create a matrix similar to the one in section-2.3 and "drag" it along the line described by line parameters to see if this is a line or a false detection. There is one problem though, as mentioned above, but not very visible in Figure-3.6. Because of binning $(r, \theta)$, it is impossible to detect precisely the correct line parameters $(m, b)$ and in large enough images, such as the images in question, this amounts to a substantial deviation. My solution was to increase the size of the matrix, but this is a very poor solution. The code snippet for this process is shown in Listing-3.3.

Listing 3.3: Testing for presence of a line

```
#retrive the rho and theta values
#for first detected line
rho = -hough[0][0][0]
theta = hough[0][0][1]
#calculate
m = -numpy.tan(theta)

bulbasaur = 0
charizard = 1
testy, testx = list(), list()
for x in range(10, n_x-10, 30):
    testx.append(x)
    y = int(m*x+rho)
    testy.append(y)
    aver = numpy.average(equ[x-10:x+10, y-10:y+10])
    charizard+=1
    if aver>=30: #about 50/400 iluminated pixels
        bulbasaur+=1

percent=float(bulbasaur)/float(charizard)
print "{} percent match".format(percent)
if percent > 0.5:
    return True, m, testx, testy, equ
else:
    return False, m, testx, testy, equ
```

In Figure-3.7 light gray rectangles represent positions of matrix for which the average value of the matrix is bigger than or equal to 30, while dark gray rectangles represent position where matrix failed to have its average value bigger than or equal to 30. This shows how this kind of trail presence testing method is extremely poor. Black gaps left behind by removing stars and the small deviations of detected line from the trail occurring because of errors in detection and conversion of line characteristics interfere profusely with testing trail presence. I have not tested this algorithm on a larger scale (>100) images to determine detection confidence but I am certain that it would be low mainly because of this last step.

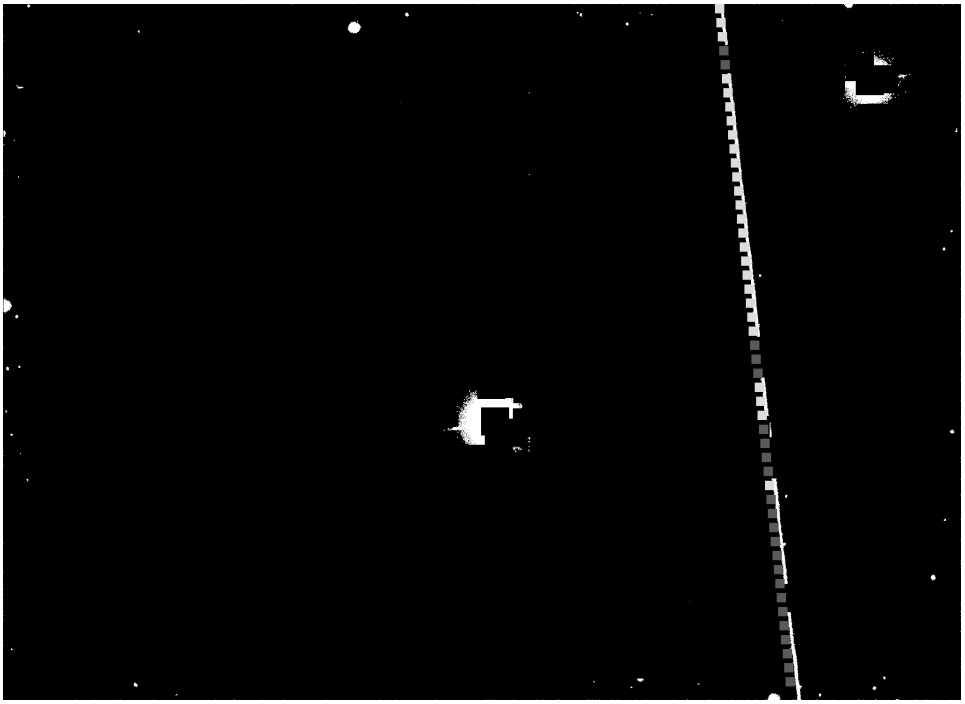

Figure 3.7: Matrix positions

### 3.0.1 Housekeeping

I have not, so far, discussed the general outlay of the entire program but have instead shown only out of context short code snippets. Apart of the despeckle function and the unsatisfying trail presence function (see Noise reduction in section-2.3 and end of chapter-3 for trail presence detection), modularity has been upheld rigorously. Every function that I was satisfied with was extracted into special definition and is functional as stand-alone modular code. Encapsulation is unimportant for this code however it is still present at some level. Ironically for such an object oriented language, encapsulation is not a fully supported Python feature. You designate a method private or protected via double underscore or a single underscore respectfully, but the end result is just name obfuscation and all are still accessible[2]. Be that as it may, current code is still a work in progress and by the time it's finished I plan to move process_field function and CSVRead function, which are still outside the class for debugging purposes, inside the RemoveStars class and make them private, leaving accessible for the end user only the class RemoveStars. General overview of finished program should by then be similar to Listing-3.4.

[2]To quote what author of Python Guido Van Rossum has to say about this: "We are all adults here!"

Listing 3.4: Code organization

```
class RemoveStars:
    """
    (run= , camcol={1-6}, field={all], filter={all})
    Defines a convenience class for end user.
    Use with caution execution time per image ~28sec.
    "Instant" use is possible, ie:
        RemoveStars(run=2888).process()
        RemoveStars(run=2888, camcol=1).process()
        RemoveStars(run=2888, camcol=1, filter='i').process()
        RemoveStars(run=2888, camcol=1, filter='i', field=139).
            process()
    Standard use still applies:
        test = RemoveStars(run= ,[optional: camcol, filter, field
            ])
        test.process()
    init parameters
    ----------------
    Keywords:
        run:    run ID

    Optional:
        camcol: camera column ID
        field:  frame ID
        filter: filter ID {u,g,r,i,z}
    dependencies
    ----------------
    -numpy, scipy, sdsspy, esutil, fitsio, cv2, csv
    -photoField.fits, runList.par and frame files for the run of
        interest
    """
    def __init__(self, **kwargs):
        #private initialization method
        #calls _load() method
    def __load(self):
        #unpacks variables sent to init
        #reports error for uncompleted variable sets
        #instantiates
    def process(self):
        #void public method, Convenience function that runs
        #process_field() for various inputs.
    def __process_field(_run, _camcol, _filter, _field):
        #private method, processes images as described so far
    def _run_info(self), _despeckle(self), _CSVRead(path)
        _trail_confirm(self), _HistEq(self) etc..:
        #protected methods called by __process_field
```

# Chapter 4

# Conclusions and recommendations

## 4.1 Conclusions

This program represents a robust and relatively fast solution for the problem of line detection in SDSS images. There is hardly part a of code not in need of refining and for most issues I already have a general solution. The biggest obstacle to perfecting this code I see in trail confirmation. I have to yet design a clean fast solution for this issue.

## 4.2 Recommendations

First step of image processing is setting negative values in the image to zero using a pixel-by-pixel iteration which can be avoided with ndarray.where(condition) function which should drastically reduce run time. In future versions bug reports should be saved into files and a flag system should be devised for recovery purposes in case code crashes mid-execution. I should also spend some time studying the scipy.KDTree functionality for determining the closest neighbors, which seems to have promising qualities for both noise reduction and trail confirmation section. As for the unfinished functionality for star size approximation and the untrustworthy coordinate conversion, there are CAS products stored in FITS file formats that contain more data about detected objects than what can be accessed by an SQL query. Among the additional information there are keywords describing type of object (galaxy/star) and provide various star radii or, in case of galaxy, provide its eccentricity. Also there seem to be keywords describing centroid coordinates in pixel values, which would avoid the need for numerical coordinate conversion. Unfortunately some of the entries are poorly described and there is obvious discrepancy between SDSS file specifications and actual file content located on their servers. Code should be tested on a larger batch of sample images of various quality for which the expected results are known so that detection rates can be estimated.

# Appendix A

# Munu2Pix conversion function

Listing A.1: Munu2Pix function

```
1
2  def munu2pix(self, field, filter, mu, nu, color=0.3):
3          """
4          Convert between SDSS great circle coordinates and
5          pixel coordinates.
6          Solve for the row,col that are roots of the equation
7              row,col -> mu,nu
8          This is because we only have the forward transform
9          parameters
10         ----------
11         field: integer
12             SDSS field number
13         mu,nu:
14             SDSS great circle coordinates in degrees
15         outputs
16         -------
17         row,col:
18             pixel values
19         """
20         import scipy.optimize
21         mu,nu,are_scalar=get_array_args(mu,nu,"mu","nu")
22         color=self._get_color(color, mu.size)
23
24         trans=self.trans
25         w=self._get_field(field)
26
27         fnum=self._get_filter_num(filter)
28
29         if 'f' in trans.dtype.names:
30             fname='f'
31         else:
32             fname='ff'
33
34
35         # pack away this data for the optimizer
36
```

```
37          fd={'a':trans['a'][w,fnum],
38              'b':trans['b'][w,fnum],
39              'c':trans['c'][w,fnum],
40              'd':trans['d'][w,fnum],
41              'e':trans['e'][w,fnum],
42              'f':trans[fname][w,fnum],
43              'drow0':trans['drow0'][w,fnum],
44              'drow1':trans['drow1'][w,fnum],
45              'drow2':trans['drow2'][w,fnum],
46              'drow3':trans['drow3'][w,fnum],
47
48              'dcol0':trans['dcol0'][w,fnum],
49              'dcol1':trans['dcol1'][w,fnum],
50              'dcol2':trans['dcol2'][w,fnum],
51              'dcol3':trans['dcol3'][w,fnum],
52
53              'csrow':trans['csrow'][w,fnum],
54              'cscol':trans['cscol'][w,fnum],
55              'ccrow':trans['ccrow'][w,fnum],
56              'cccol':trans['cccol'][w,fnum],
57              'color0':trans['ricut'][w,fnum]}
58
59          self._field_data=fd
60
61
62          det = fd['b']*fd['f'] - fd['c']*fd['e']
63          mudiff = mu - fd['a']
64          nudiff = nu - fd['d']
65          row_guess = ( mudiff*fd['f'] - fd['c']*nudiff )/det
66          col_guess = ( fd['b']*nudiff - mudiff*fd['e'] )/det
67
68          row=zeros(mu.size,dtype='f8')
69          col=zeros(mu.size,dtype='f8')
70          for i in xrange(mu.size):
71              self._tmp_color=color[i]
72              self._tmp_munu=array([mu[i],nu[i]])
73              rowcol_guess=array([row_guess[i], col_guess[i]])
74
75              rowcol = scipy.optimize.fsolve(self._pix2munu_for_fit,
                        rowcol_guess)
76              row[i] = rowcol[0]
77              col[i] = rowcol[1]
78
79          if are_scalar:
80              row=row[0]
81              col=col[0]
82
83          return row,col
```

# Appendix B

# Full code listing

Listing B.1: Full code listing

```
1  import csv
2  import fitsio
3  import sdsspy
4  import numpy
5  import scipy.ndimage as nd
6  import scipy
7  import astrometry
8  import cv2
9
10 #do NOT use 'path' as a variable in program,
11 #it's a dummy test variable
12 #consider using sdsspy.files.filename() instead
13 path = '/home/dino/Desktop/test_slike/boss'
14
15 import gc
16 import timeit
17 import time
18
19 class Timer:
20     def __init__(self, timer=None, disable_gc=False, verbose=True)
       :
21         if timer is None:
22             timer = timeit.default_timer
23         self.timer = timer
24         self.disable_gc = disable_gc
25         self.verbose = verbose
26         self.start = self.end = self.interval = None
27     def __enter__(self):
28         if self.disable_gc:
29             self.gc_state = gc.isenabled()
30             gc.disable()
31         self.start = self.timer()
32         return self
33     def __exit__(self, *args):
34         self.end = self.timer()
35         if self.disable_gc and self.gc_state:
```

```
36              gc.enable()
37          self.interval = self.end - self.start
38          if self.verbose:
39              print('time taken: %f seconds' % self.interval)
40
41  def CSVRead (path):
42      """
43      Defines a function that reads CSV file given by
44      (str) path into a list of dictionaries.
45      Returned list is arranged as
46          {[ra:, dec:, u:, g:, r:, i:, z:],
47              ...}.
48      """
49      labels=['ra', 'de', 'u', 'g', 'r', 'i', 'z']
50      read = csv.DictReader(open(path), labels, delimiter=',',
            quotechar='"')
51      lines = list()
52      for line in read:
53          lines.append(line)
54      return lines
55
56  def process_field(_run, _camcol, _filter, _field):
57      """
58      Function that removes stars from image.
59      init parameters
60      ----------------
61      Keywords:
62          run:
63              run ID
64          camcol:
65              camera column ID
66          field:
67              frame ID
68          filter:
69              filter ID (ugriz)
70      """
71      Coord = CSVRead(sdsspy.files.filename('CSVCoord', run=_run,
            camcol=_camcol,
72                                            field=_field))
73      fname = sdsspy.files.filename('frame', run=_run, camcol=
            _camcol,
74                                    field=_field, filter=_filter)
75
76      fits = fitsio.FITS(fname , mode='rw')
77      img = fits[0].read()
78      conv = astrometry.Astrom(run=_run, camcol=_camcol)
79
80      #faster alg then pix iteration?
81      for x in range (0, img.size/img[0].size, 1):
82          for y in range(0, img[0].size-1, 1):
83              if img[x, y] < 0.0:
84                  img[x, y] = 0.0
85
86      starConv_errnum = int()
```

```
87      for star in Coord:
88          try:
89              ra = float(star['ra'])
90              de = float(star['de'])
91              xy = conv.eq2pix(_field, _filter, ra, de)
92              x, y=xy[0], xy[1]
93              #first hand star size approx, expRAd_ugriz,
                    petroR90_ugriz
94              #should work better. Diff between exp (mag fit only?)
                    and petro?
95              #Download fits coord files because of flags?
96              #http://www.sdss3.org/dr8/algorithms/bitmask_flags1.
                    php
97              #Converting arcsec to pix?
98              #Processing without petro or exp?
99              #rowc, colc no need to convert coords?
100             #for gal exp disk fit? de Vaucouleurs fit ln()?
101             img[x-30:x+30, y-30:y+30].fill(0.0)
102         except ValueError as err:
103             print err.message
104             starConv_errnum+=1
105             pass
106     print "{} coordinate conversions failed. \nTotal number of
            stars \
107 on the image: {}".format(starConv_errnum, len(Coord))
108
109     scipy.misc.imsave(path+'/processing.png', img)
110     #http://www.comp.nus.edu.sg/~cs4243/conversion.html
111     #conversions don't work, can't convert ndarray directly to cv2
            image
112     #WARNING CV2 MIRRORS THE IMAGE HORIZONTALY
113     gray_image = cv2.imread(path+'/processing.png', cv2.
            CV_LOAD_IMAGE_GRAYSCALE)
114
115     equ = cv2.equalizeHist(gray_image)
116
117     #faster alg then pix iteration?
118     n_x, n_y=equ.shape
119     for x in range (1, n_x-2, 1):
120         for y in range(1, n_y-2, 1):
121             if equ[x, y] != 0:
122                 li = [equ[x-1, y+1], equ[x, y+1], equ[x+1, y+1],
123                       equ[x-1, y],   equ[x, y],   equ[x+1, y],
124                       equ[x-1, y-1], equ[x, y-1], equ[x+1, y-1]]
125                 aver = numpy.average(li)
126                 #first hand approx of average pix values?
127                 #consider a better substitute?
128                 if aver<=85:
129                    equ[x, y]=0
130
131     hough = cv2.HoughLines(equ, 1, numpy.pi/180, 1)
132
133     rho = -hough[0][0][0]
134     theta = hough[0][0][1]
```

```
135     m = -numpy.tan(theta)
136
137     bulbasaur = 0
138     charizard = 1
139     testy, testx = list(), list()
140     for x in range(10, n_x-10, 30):
141         testx.append(x)
142         y = int(m*x+rho)
143         testy.append(y)
144         aver = numpy.average(equ[x-10:x+10, y-10:y+10])
145
146         charizard+=1
147         #lame first hand approx, find a better one
148         if aver>=30:
149             bulbasaur+=1
150             #draw rectangles for thesis
151             equ[x-10:x+10, y-10:y+10] = 220
152         else:
153             #draw rectangles for thesis
154             equ[x-10:x+10, y-10:y+10] = 90
155
156     scipy.misc.imsave(path+'/rectangles.png', equ)
157     percent=float(bulbasaur)/float(charizard)
158     print "{} percent match".format(percent)
159
160     if percent > 0.5:
161         return True, rho, theta, m
162     else:
163         return False, rho, theta, m
164
165 #time taken: 28.756897 seconds for 2888
166 #time taken: 27.543327 seconds for 5972
167 #time taken: 30.423426 seconds for 2888
168 #time taken: 28.940095 seconds for 5972
169 #time taken: 27.989103 seconds for 2888
170
171 class RemoveStars:
172     """
173     (run= , camcol={1-6}, field={all], filter={all})
174     Defines a convenience class that goes through the entire run
175     and deletes stars from image. If camcol is sent the whole
176     run-camcol is processed. If run-camcol-filter-field is sent
        only
177     that frame is processed.
178     Use with caution execution time per image ~23sec.
179     Instant use is possible, ie:
180         RemoveStars(run=2888).process()
181         RemoveStars(run=2888, camcol=1).process()
182         RemoveStars(run=2888, camcol=1, filter='i').process()
183         RemoveStars(run=2888, camcol=1, filter='i', field=139).
              process()
184     Standard use still applies:
185         test = RemoveStars(run= ,[optional: camcol, filter, field
              ])
```

```
186         test.process()
187     Processed images are placed in path+filename.png folder.
188     init parameters
189     ----------------
190     Keywords:
191         run:
192             run ID
193     Optional:
194         camcol:
195             camera column ID
196         field:
197             frame ID
198         filter:
199             filter ID {u,g,r,i,z}
200     dependencies
201     -------------
202     -numpy, scipy, sdsspy, esutil, fitsio
203     -photoField, runList.par and frame files for the run of
            interest
204     """
205
206     def __init__(self, **kwargs):
207         self.kwargs=kwargs
208         self._load()
209
210     def _run_info(self):
211         rl = sdsspy.files.runlist()
212         w, = numpy.where(rl['run'] == self._run)
213         if w.__len__() == 0:
214             raise ValueError("Run %s not found in runList.par" %
                    self._run)
215         startfield = rl[w]['startfield'][0]
216         endfield = rl[w]['endfield'][0]
217         return startfield, endfield
218
219     def _load(self):
220         self._run, self._camcol, self._field = int(), int(), int()
221         self._filter, self._pick = str(), str()
222         kwargs = self.kwargs
223
224         if 'run' not in kwargs:
225             raise ValueError("send run= ")
226         self._run=kwargs['run']
227         self._pick = 'run'
228
229         if 'camcol' in kwargs:
230             self._camcol = kwargs['camcol']
231             self._pick = 'camcol'
232
233         if 'field' in kwargs:
234             if self._camcol is 0:
235                 raise ValueError("send camcol= ")
236             self._field = kwargs['field']
237
```

```
238         if 'filter' in kwargs:
239             if self._camcol is 0:
240                 raise ValueError("send camcol= or camcol= and
                        field=")
241             else:
242                 self._filter = kwargs['filter']
243                 self._pick = 'camcol-filter'
244 
245             if self._field is 0:
246                 pass
247             else:
248                 self._filter = kwargs['filter']
249                 self._pick = 'field'
250 
251         if (self._run and self._camcol and self._field != 0
252         and self._filter.__len__() == 0):
253             raise ValueError("send filter=")
254 
255     def process(self):
256         """
257         Convenience function that runs process_field() for
258         various inputs.
259         """
260         if self._pick == 'run':
261             startfield, endfield = self._run_info()
262             filters = ('u', 'g', 'r', 'i', 'z')
263             camcols = (1, 2, 3, 4, 5, 6)
264             for i in range (0, 5, 1):
265                 for j in range (0, 4, 1):
266                     for _field in range (startfield, endfield, 1):
267                         process_field(self._run, camcols[i],
                                filters[j], field)
268 
269         if self._pick == 'camcol':
270             startfield, endfield = self._run_info()
271             filters = ('u', 'g', 'r', 'i', 'z')
272             for j in range(0, 4, 1):
273                 for field in range (startfield, endfield, 1):
274                     process_field(self._run, self._camcol, filters
                            [j], field)
275 
276 
277         if self._pick == 'camcol-filter':
278             startfield, endfield = self._run_info()
279             for field in range (startfield, endfield, 1):
280                 process_field(self._run, self._camcol, filters[j],
                         field)
281 
282         if self._pick == 'field':
283             img=process_field(self._run, self._camcol, self.
                    _filter, self._field)
```

# Bibliography

C. P. Ahn, R. Alexandroff, C. Allende Prieto et al., The Ninth Data Release of the Sloan Digital Sky Survey: First Spectroscopic Data from the SDSS-III Baryon Oscillation Spectroscopic Survey, Astrophysical Journal Supplement 203 (2012) 21;

Bill Keel, Lecture Notes, http://www.astr.ua.edu/keel/techniques/astrom.html

Gray J., Slutz D., Szalay S. A., Thakar A. R., VandenBerg J., Kunszt Z. P., Stoughton C., Data Mining the SDSS SkyServer Database, Microsoft Corporation Technical Report MSR-TR-2002-01 (2002)

Jenniskens P. (2006). Meteor Showers and their Parent Comets. Cambridge University Press, Cambridge, U.K., 790 pp.

Whipple F. L. (1951). A Comet Model. II. Physical Relations for Comets and Meteors. Astrophys. J. 113, 464

Martins Z., Price C. M., Goldman N., Sephton, A. M., Burchell J. M. Shock synthesis of amino acids from impacting cometary and icy planet surface analogues. Nature Geoscience (2013)

Fernndez, Julio A. (2006). Comets. p. 315.

Whitman, K; Morbidelli, A; Jedicke, R (2006). "The size−frequency distribution of dormant Jupiter family comets". Icarus 183: 101.

Calvet, N., & Gullbring, E. 1998, ApJ, 509, 802

Gunn, J. E.; Carr, M.; Rockosi, C.; Sekiguchi, M.; Berry, K.; Elms, B.; de Haas, E.; Ivezić, Ž.;