Linear feature detection on SDSS images

Bektešević, Dino

Linear feature detection on SDSS images

Dino Bektešević

Supervisor: izv.prof.dr.sc. Dejan Vinković

Split, September 2015

Master Thesis in Physics

Department of Physics
Faculty of Science
University of Split

Abstract

Sloan Digital Sky Survey (SDSS), covering just over 35% of the full sky, is the largest sky survey conducted. Many of the images made contain linear features that can only be explained by objects traveling at different angular velocities than the background sky. Certain number of linear features can be attributed to meteors, which makes their extraction from images scientifically interesting. In this work I present and discuss the full technical details of a software tool capable of searching through the entire SDSS imaging database for such linear features.

Acknowledgements

This project started almost 4 years ago when enrolled in a Research project in Astrophysics I course wanting to continue experiment of detecting lunar meteor impacts started earlier that summer. I was unpleasantly surprised when I was met with opposition from professor Dejan Vinković, insisting instead that I start working on a different topic of removing stars from SDSS images. Dejan wanted me to continue the work on a topic of his interest and that did not sit with me well at the time. In hindsight his opposition and stubbornness is the best thing that could have happened to me and as the saying goes, hindsight is always 20/20. Therefore, I would like to extend my deepest gratitude to Dejan Vinković for his persona, his incredible optimism, his stubbornness, the only one I found outmatching my own and his belief in me through the years even when I was sure I do not know what to do next. Through our work I had learned more than I could have ever imagined and for that I will forever be indebted to him.

In addition, special thanks are due to Darko Jevremović for granting access to the cluster Fermi, on which the LFDS was run. Without his technical knowledge and efforts in obtaining the large data set necessary LFDS would never become a true technical sollution, but remain just a proof of concept. I would also like to extend my gratitude for his incredible patience and good will when it came to tolerating all the accidents I caused. I would also like to apologize to Darko, for all the times I managed to completely crash the Fermi cluster and halt program executions of all other users.

I would also like to acknowledge the SDSS and extend my gratitude for sharing their data so openly. Funding for the Sloan Digital Sky Survey IV has been provided by the Alfred P. Sloan Foundation, the U.S. Department of Energy Office of Science, and the Participating Institutions. SDSS-IV acknowledges support and resources from the Center for High-Performance Computing at the University of Utah. The SDSS web site is www.sdss.org.

SDSS-IV is managed by the Astrophysical Research Consortium for the Participating Institutions of the SDSS Collaboration including the Brazilian Participation Group, the Carnegie Institution for Science, Carnegie Mellon University, the Chilean Participation Group, the French Participation Group, Harvard-Smithsonian Center for Astrophysics, Instituto de

Astrofísica de Canarias, The Johns Hopkins University, Kavli Institute for the Physics and Mathematics of the Universe (IPMU), University of Tokyo, Lawrence Berkeley National Laboratory, Leibniz Institut für Astrophysik Potsdam (AIP), Max-Planck-Institut für Astronomie (MPIA Heidelberg), Max-Planck-Institut für Astrophysik (MPA Garching), Max-Planck-Institut für Extraterrestrische Physik (MPE), National Astronomical Observatory of China, New Mexico State University, New York University, University of Notre Dame, Observatário Nacional, MCTI, The Ohio State University, Pennsylvania State University, Shanghai Astronomical Observatory, United Kingdom Participation Group, Universidad Nacional Autónoma de México, University of Arizona, University of Colorado Boulder, University of Oxford, University of Portsmouth, University of Utah, University of Virginia, University of Washington, University of Wisconsin, Vanderbilt University, and Yale University.

Contents

Listings

List of Figures

List of Tables

Chapter 1

Introduction

Sloan Digital Sky survey (SDSS) has been one of the most successful surveys in the history of astronomy. In its operational period it categorized 932 891 133 unique objects (Alam et al., 2015) and provided us with their precise astrometric and photometric measurements. Majority of those objects are most certainly stars, galaxies, nebulas and other deep-space objects but a certain minority of detected objects are much closer to us. Search for solar system objects such as asteroids, comets and other trans-Neptunian objects (TNO's)[1] has been attempted before (Ivezić et al., 2001; Solontoi et al., 2010).

The "closest"[2] of the listed objects are asteroids. Majority of asteroids occupy the asteroid belt located between Mars and Jupiter at 2.1 to 3.3 astronomical units (AU)[3]. These are still incredibly large distances that can not be put into perspective of everyday life. Every now and then, however, SDSS fortuitously manages to image a family of objects that are closer still than asteroids. These objects are meteors and they are visible for a short period in Earth's atmosphere at 80-110km in height.

Meteors are generally prerogatively investigated by the International Meteor Organization (IMO) through the combination of visual, video and radar observations. All currently used tools for meteor observation suffer from the same issue - a high limiting magnitude. In recent years efforts in detecting fainter meteors has been made by telescopic observations of meteors. Iye et al. (2007) reports on the findings of 13 meteors and 44 artificial objects on the 8.2m large Subaru telescope. However, no larger effort has been made to systematically detect telescopic meteors yet.

Unlike IMO or Subaru, SDSS does not suffer from high limiting magnitude or a narrow field of view. SDSS also prides itself in keeping all its data

[1] such as minor planets, Kuiper belt objects (KBO's), Oort cloud objects (OCO's) etc...

[2] Reader should note that some of these objects have a very elongated and/or irregular orbits which can lead them very close to Earth. Objects that pose a possible collision threat are known as Near-Earth objects (NEO's).

[3] 1 AU is roughly the distance between Earth and Sun, 150 million km.

publically availible. In this thesis I present the first ever software tool written for a fast systematic search of the large SDSS database in search for linear features left behind by meteors or satellites. I detail its mode of operation, performance and detection limits and describe possible improvements.

1.1 Meteors

Meteoroid is 10μm to 1m in size naturally formed body moving in interplanetary space. A micrometeroid is a meteoroid 10μm to 20mm in size (Rubin and Grossman, 2010). Meteor is the light phenomenon resulting from the passage of a meteoroid or a micrometeoroid through the Earth's atmosphere (IMO, 1997). Most meteors are fragments or remains of a comet (Jenniskens, 2006) or an asteroid (Vokrouhlický and Farinella, 2000) while others can be attributed to collision impact debris ejected from larger bodies such as planets (Morris et al., 2000). Meteors whose origin can be attributed to a comet appear annually around the same dates. These events are known as meteor showers.

A point on the sky from which meteors apparently originate is called the radiant. Thus meteors whose origin body is a comet are known under the name of the radiant's (Perseids,...) they originate from, while meteors whose apparent origin on the sky does not belong to any known radiant is referred to as a sporadic.

Initial danger at detection meteor trails would most definitely come from short trails. Trails that are tens of pixels long would be incredibly hard to detect among other objects present on the images. Meteors are fast moving transient phenomena hence, due to their large angular velocities it is not expected that they will leave short trails. Approximating their expected minimal trail lengths, however, is a nontrivial matter. According to the McLeod (1993) a good trail length measuring comparisons for visual observations include

> Pointers of the Big Dipper, Pollux and Castor, and Aquila's head (all 5 degrees), Orion's Belt (3 degrees), and the open side of the Big Dipper's bowl (l0 degrees)

indicating that not very often a meteor trail spanning less than 3° on the night sky is recorded. Such a meteor would surely cover the entire SDSS imaging field (see section-1.2.1), and in the worst case scenario, if imaged fully, would produce a trail at least 1489 pixels long. Considering these are visual observations, it is likely that values are biased towards the longer trails than actually occurring.

We can approximate the angular trail length even further by using the formulas and tables for their angular velocities provided by IMO (1995a). Theoretical angular velocities of meteors range from $0.2°/s$ for meteors

whose end-point is within 5° of their radiant that is located no higher than 10° above the horizon, to 33°/s for meteors whose end-point is perpendicular to their radiant located in the zenith, therefore with the meteor close to horizon. Meteors appearing at lower altitudes (closer to the horizon) and meteors appearing and ending near the radiant point are dramatically slower. These are extreme cases of the slowest and fastest angular velocity *theoretically* possible. Mostly, depending on the position of the observer relative to the radiant, meteor angular speeds are inside the 7-17°/s range. According to McLeod (1993):

> A streak or flash with no moving body visible is 0.2 to 0.3 second, which constitutes the majority of meteors. Rather rare is 0.1 second, only a few per year at most.

which means that the worst case scenario is a trail length of 0.7°. If imaged fully, corresponding trail length would be approximately 1000 pixels long.

Another criterion for assessing of meteor trail length is posed by the limits of SDSS instruments. Large telescopes are often limited from observing bellow certain elevations above horizon due to atmospheric extinction and mirror deformations that occur. According to York et al. (2000) SDSS reaches this limit at 35° of elevation. Furthermore according to IMO (1995b):

> For radiant elevations higher than 30° the apparent path length l of a shower meteor amounts at most to half the distance from the radiant to the start point.

Which means that even for a meteor originating 1° from its radiant, if imaged fully, the trail would be at least approximately 400 pixels long.

Even in the worst-case scenario the trails visible on the images are spanning a little under $1/3^{rd}$ of the entire image. These trails, therefore, present major objects on the images and will dominate over the noise, large stars and galaxies that could also be present in them.

Another trail characteristic alongisde its length, is its width. At 110-70km in height, compared to 300-32 000 km distant satelites, we can expect meteors to leave a wider trail on the SDSS camera. Because meteors enter Earth's atmosphere under an angle, decreasing their height by 10-30km during their duration, we can also expect the starting parts of their trails to be thinner than the ending parts of their trails. This can be attributed to the defocusing effect (Iye et al., 2007) of SDSS optical systems focused at infinity.

Considering that we expect to see more satelite trails than meteor trails, a precise differentiation criteria will be needed separating the two. Alongside to the trail thickness, which is expected to be larger for meteors due to defocusing effects, meteors would not exhibit significant periodicity in their

light curves. Satelites could, and usually do, exhibit periodical brightness change that can be attributed to their rotation and subsequent periodical reflection of sunlight from their solar panels.

Additionally it is expected that there will be a significant difference in velocities. Fastest meteors recorded are those in retrograde orbits with respect to Earth. Such meteors have speeds of ~70km/s (Trigo-Rodriguez et al., 2008). By using the equation-1.1 we can calculate that at the height of 200km orbital speed of a satelite would have to be 8km/s, at 300km 7.5km/s etc. For the fastest satelite orbiting the Earth, we still have an order of magnitude difference in velocity from that of a meteor. We can additionally calculate the largest satelite angular speed to be 1.5°/s.

$$v_{satelite} = \sqrt{\frac{GM_z}{R_z + h}} \tag{1.1}$$

Following the tables and formulas provided by IMO (1995a), including in SDSS observing height limitation, we can see that this would match the angular speed of a shower meteor traveling 25km/s ending within 10° radius from its radiant. Speeds of 25km/s are among the slowest speeds meteoroids can have and are attributed to meteoroids moving in the same direction as Earth (Trigo-Rodriguez et al., 2008). Therefore only a small subset of meteors will produce a potentially ambiguous trail.

1.2 Introduction to SDSS

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 have been 12 different data releases (DR's) totaling to 60TB of data. Considering that a systematic astrometry error has been fixed no earlier than DR9 (Ahn et al., 2012), caution is advised when using the software on any of the older DR's. In this paper I have used DR12 (Alam et al., 2015), the latest DR available. SDSS uses a 2.5m modified version of Ritchey-Chretien telescope which, due to its hyperbolic mirrors, has a 3° field of view devoid of any major optical errors (Gunn et al., 2006). Telescope is located at Apache Point Observatory (APO) in Sunspot, New Mexico.

1.2.1 Camera

Light collected by the telescope is directed at a large imaging camera, shown on figure-1.1, with a 2.5° field of view, constructed from two arrays of time delay and integrate charged coupled devices (Gunn et al., 1998).

Time delay integration (TDI), or drift scanning, is an imaging technique developed in the 1950-1960's for aerial reconnaissance. TDI enabled longer exposure and prevented image smear. The gist of the technique is to have a very long roll of film, which would then be dragged in front of constantly opened shutter at the same relative speed as the imaging target. This meant that for a camera mounted on underbelly of an airplane the film would roll at the, preferably constant, air-to-ground speed of the aircraft. The survey operates under the same assumptions: imaging target (the sky) is moving at a constant velocity relative to the CCD arrays. If the registers of the CCD are clocked in synchronization with the imaging target then the signal is added to the same register each clock cycle, therefore the end signal is integrated through the full length it traversed over the CCD.

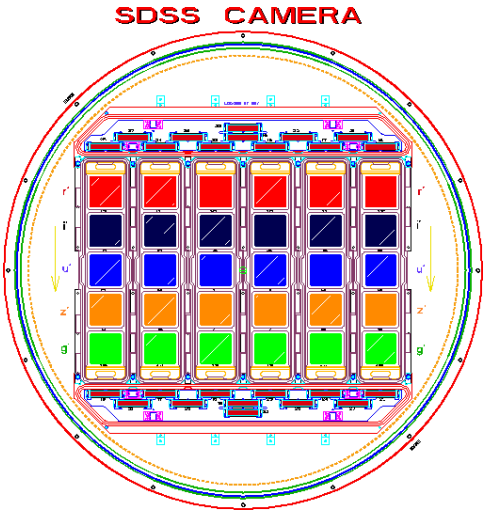


Figure 1.1: SDSS Camera layout.

As mentioned, the camera is constructed out of two CCD arrays: the photometric array and the astrometric array. On a technical figure-1.2 of the telescopes focal plane, the photometric array is represented by the array of squares in the middle. It is constructed out of 30 SITe/Tektronix 2048x1489 pixel CCDs arranged in six columns of five CCDs, each aligned with the pixel columns of the CCDs themselves. Filters r, i, u, z, and g cover the rows of the array respectively. These CCDs produce the images that I will be processing. Astrometric array, see figure-1.2, is represented by the rectangles above and below the photometric array. Astrometric imaging array is constructed out of 24 additional 2048x400 pixels CCDs placed around the photometric array. This array produces the precise object astrometry and keeps track of the focus of the telescope.

Because of the way camera is designed and the fact it functions in a drift scan mode, it produces five images of a given object, all from the same column of CCDs, one from each CCD in that column. It takes an object 54 seconds to move from the beginning of a CCD to the end, so the effective exposure time in each filter is 54 seconds. Because there is some space between the rows of CCDs it takes an image 71.7 seconds to move from the beginning of one row to the next. Each row corresponds to a different filter, so each object has one image in each filter, taken at 71.7 second intervals.

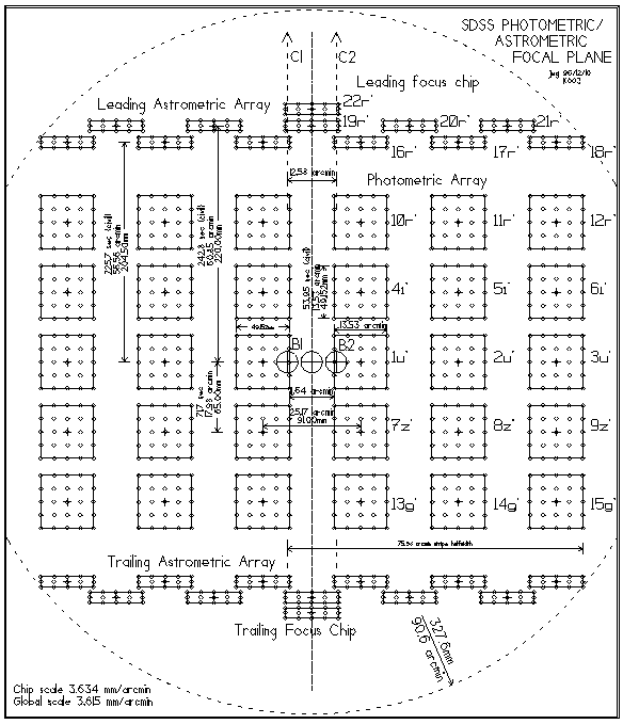


Figure 1.2: SDSS telescopes focal plane.

1.2.2 Run, camcol, filter, field

Consider a situation in which the telescope is tracking in such a way that it is guaranteed that the sky will drift in parallel with the columns of the camera shown in figure-1.2. One such continuous scan of the imaging telescope is called a **run**. Each run therefore scans six long separated tracks on the sky, one for each of the camera columns. These six tracks taken together are also called a **strip**. For science quality runs, the lowest **run** number is 94, and the highest is 8162.
Each column of CCDs is assigned an integer value from 1 to 6. These columns are referred to as **camcols**. As noticeable from figure-1.2 between each camcol there is a large gap. These gaps are filled by another overlapping run. Two strips from the two overlapping runs make for a **stripe**.

It is possible to access each of six individual tracks of a run by referring to them with a **run-camcol** designation. This would return us a single track of the sky in five different filters. Naturally, it is also possible to further make the selection by specifying a **filter**. This **run-camcol-filter** sky track is then further divided into sections 2048x1361 pixels large, which are called **fields**. Each field has the first 128 rows of the following field attached to it, so that all survey images actually have a size of 2048x1489 pixels. Such fields, that have the attached adjacent fields, are often referred to as **frames**.

Finally, there have been multiple reprocessing of the data over the years. Each reprocessing, called a **rerun**. Rerun is not necessarily a repeated run; a rerun could be any re-processing of the data. Usually reruns are just old data reprocessed by newer software. Reruns have arbitrarily assigned indices, therefore different runs with the same reruns are not necessarily produced with the same software. However, currently, all runs, except very few exceptional ones used for supernova search, have the same rerun: 301.

1.2.3 Data access, files and folder structure

Gray et al. (2002) projected that each year of operation, SDSS gathered about 5TB of new data. According to data volume table standings on SDSS3 webpages[4] in October 2015. database "weighed" around 60TB. Fortunately, not all the data are needed in order to look for linear features.

In essence there are two ways to access SDSS data. Using Science Archive Server (SAS)[5] and Catalog Archive Server (CAS)[6]. SAS serves the binary image files and full spectroscopic results, it allows users to download raw (FITS) images and spectra from the survey. CAS provides access to the database that contains the object catalogs and related data such as various derived parameters. CAS's data is stored in a commercial relational database management system (DBMS) - Microsoft's SQL Server. A former attempt at writing the software, (Bekteševi ć, 2013), utilized both data access points. That approach, however, proved hard to maintain and inefficient to apply in a production environment. Currently only CAS files are used in the processing of images.

Maintaining a constant folder structure, shown on figure-1.3, is of crucial value when dealing with such a large number of files. SDSS maintains a very logical per-experiment based folder structure. Top-most folder is named after the DR in question. That folder, i.e. for DR12: `/SAS/DR12`, contains folders of all the experiments currently preformed at SDSS such as APO, APOGEE, MAnGA, MARVELs and BOSS.

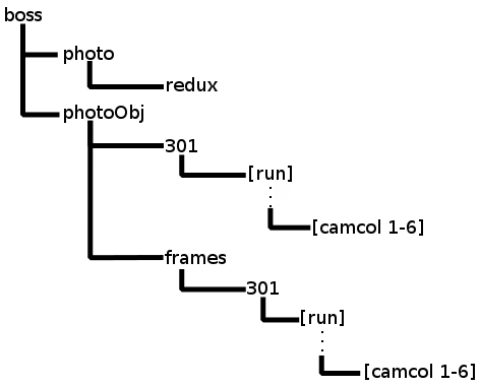


Figure 1.3: CAS folder structure.

[4] https://www.sdss3.org/dr9/data_access/volume.php
[5] http://data.sdss3.org/
[6] http://skyserver.sdss.org/

BOSS, short for Baryon Oscillation Spectroscopic Survey (Dawson et al., 2013), is the folder I am most interested in. It contains the frames taken by the camera, corrected for distorsions and CCD anomalies, and their photometric calibrations. Corrected frame files in FITS format are located in `/boss/photoObj/frames/301/run/camcol/`, where, of course, the run and camcol have to be replaced with the actual values.

Each frame fits file has a naming convention of its own:

frame-[ugriz]-[0-9]6-[1-6]-[0-9]4.fits.bz2

where [ugriz] is the filter, [0-9]{6} is a zero-padded six-digit number containing the run number, [1-6] is the camera column ('camcol') and [0-9]{4} is the zero-padded, four-digit frame sequence number. For example, run 94's first field was numbered 11. If we then still specify that we want the first camcol in the near infrared area, filter i, we would locate that file as:

`/boss/photoObj/frames/301/94/1/frame-i-000094-1-0011.fits.bz2`

Notice how the path values for run and camcol are not padded while the file ones are. In addition notice that the fits file itself is compressed in a bz2 format.

Frame files have 4 header units, HDU[0-3]. HDU0 is a 2048x1489 "image" of the sky. It is a 2048x1489 array of floating point values that represent sky-subtracted corrected frame. HDU1 is the calibration vector, a 2048 element float32 array used to calibrate HDU0. It is mostly used to de-calibrate the frame back to the original state. HDU2 is a 259x192 float32 array that represents the noise values of the sky. It has been already subtracted from the raw data to obtain HDU0. HDU3 is an asTrans structure. asTrans provides us with precise astrometric conversion coefficients for each field. Full description of frame files can be found on the data model webpages[7].

Another file type we are interested in, are the photoObj files. They are located in `/boss/photoObj/301/run/camcol/` and have the similar naming convention as frame files:

photoObj-[0-9]{6}-[1-6]-[0-9]{4}.fits

photoObj files contain the full calibrated outputs of SDSS photometric pipeline. Each photoObj file has two headers. HDU0 is just the keyword template for HDU1. HDU1 contains a table of all objects registered by the deblender and resolver pipeline and their full photometric profiles. These files provide us with already calculated object center in pixel coordinates on the image itself, while the asTrans provide only the means to calculate them, with the radii containing 50% and 90% of petrosian flux, with an

[7] http://data.sdss3.org/datamodel/files/BOSS_PHOTOOBJ/frames/RERUN/RUN/CAMCOL/frame.html

object type estimate (star, galaxy...), likelihoods that those estimates are correct, various usefull warning flags and if possible with deVaucouleurs fit[8]. These information are then used in *removestars* (see section-2.2.5.2) module to delete these objects from the image.

In addition to data files SDSS maintains so called parameter files. These files end with a ".par" extension. Parameter files are ASCII encoded and are informally called "Yanny" files, or, rarely, FTCL files. They are used in SDSS-III for storing moderate amounts of data in a form that is both human and machine readable. Parameter file I am most interested in is called runList. From this file it is possible to extract a list of all the runs, their starting and ending frames, and state of completeness. runList.par file is located in `/boss/photo/redux/runList.par` and its internal structure is shown in listing-1.1.

Listing 1.1: sdssFileTypes.par

```
typedef struct {
  int    run;         # Run number
  char   rerun[];     # Relevant rerun directory
  int    exist;       # Does a directory exist (has it ever been
       queued) (0/1)?
  int    done;        # Is this field done (0-Not done/1-Done)?
  int    calib;       # Is this field calibrated (0-No/1-Yes)?
                      # All the fields below are only updated if
                           the run is done
                      # Otherwise, they are set to zero.
  int    startfield; # Starting field - determined from fpObjc
      present
  int    endfield;    # Ending field
  char   machine[];   # The machine the data is on
  char   disk[];      # The disk this run is on....
} RUNDATA;

RUNDATA 1933 157 1 1 1 0011 0122 unknown unknown
RUNDATA 3172 157 1 0 0    0    0 unknown unknown
RUNDATA 94   301 1 1 1 0011 0546 unknown unknown
...
...
```

1.3 Space and time constraint estimates

Using the runList.par file it is easy to find there are 842 runs. If we summed the difference between endfields and startfields of each run, we would find that the total number of 2048x1489pixel "sections" the runs were divided into, is 187 044. Following the definition of a run (5 filters, 6 camcols), it is easy to calculate that there are total of 5 611 320 frame files. If I was to

[8]de Vaucouleurs profile describes how the surface brightness of an elliptical galaxy varies as a function of apparent distance from its center

spend a second of processing per frame the total time required to process all the frames would be 65 days. If such a job would be done in parallel over multiple different runs at the same time it would be possible to shorten the total duration of the execution depending on the number of parallel executions started. For example, if we run this task on a cluster with 10, single-core, machines total execution time would be just about six and a half days. For a cluster with, either, more machines, more cores or more CPUs per machine, multiple independent processes can be started thus shortening the execution time necessary even further.

However, the question if it is possible to constrain my calculations to a second of execution remains! Quoting Bekteševi ć (2013)

> Execution times of presented code [for detection of linear features] 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.

Stated later was that with the use of OpenCV library functions a performance gain could be achieved that reduces the execution time of some frames to under 10 seconds. However, based on a code listing titled Code Organization, average execution time was still about 28 seconds. If we spend half a minute per frame file the total execution time is approximately 2000 days. This is unsustainable execution time for any production environment.

Apart from the concerns about execution times there is also a concern about the amount of HDD and RAM space required. In section-1.2.3 I stated that "fortunately, not all 60TB of data are needed". However the data we do need ranks first in the SDSS data volume table. Corrected frame files constitute the largest dataset of SDSS with 15.37TB of data. Complete photometric catalogue is fifth on the list, constituting another 3.4TB of data. Thus total of 18.8TB of HDD space is needed.

Consider, additionally, that the space requirements listed above are stated for compressed bz2 files. It is safe to assume the largest contribution to the size of a frame file is the HDU0. A 2048x1489 float32 array takes up 11.63MB of RAM. This is without taking into consideration the extra information stored in HDU1-3. HDU1-3 for all frames contain the same data types and the same number of fits fields, just different values in them, therefore they all take the same amount of space. Thus, the exact, constant, size of the frame fits files can be calculated to be 12.4MB. The average fits file is therefore about 3.3 to 3.7 times bigger than its bz2 compressed file. This means that the total size of all bz2 files, extracted, would be in the range of 51-57TB.

If such HDD space cannot be allocated in advance, then each file will have to be bunziped during the program execution, hindering its time performance. A short test example, such as in listing-1.2, demonstrates the effect of bunzip on time performance. Two tests, 100 samples each with

resets in between, were conducted on an Intel i5 Asus laptop with 7.7GB RAM and 500GB 7200rpm Seagate Momentus (ST9500420AS) HDD. HDD benchmark on a 1000 samples, 10MB each, showed the average read rate to be 78.6MB/s with 15.76 ms access time.

Listing 1.2: bunzip time test

```
1 >>> timeit.timeit(stmt="os.popen('bunzip2 -qkc .../boss/
    photoObj/frames/301/125/1/frame-i-000125-1-0216.fits.bz2 >
    test')", setup="import os", number=100)
```

Results of the first test indicate that the average time per a bunzip command is 0.807 seconds while the average of the second test was 0.804 seconds. Similar times returned by the tests indicate that bunzip command is stable and executes constantly in time. Remember, for a 1 second of processing per frame, total execution time was 65 days on a single-core CPU. Even if the workload is split over several machines these data extraction times alone do not permit me longer processing times.

It is also highly likely that a multiple of different copies of the original image will be made in the program. For example, for 10 copies of the frame file HDU0 in a single process, 120-130MB of RAM has been allocated. Ten copies of the "same" array might sound unreasonable; however it is worthwhile noticing that numpy arrays[9] (the HDU0 is represented by numpy array) are immutable. That means a copy will have to be made every time a function is called on that array. It might also be mandatory to keep several copies, all processed in a different way, in order to compare the results of linear feature search to achieve a more trustworthy result. Additionally it will be necessary to store other variables (photoObj data, lists, dictionaries, objects, etc..). It is not unreasonable to expect that a single process can take up to 250MB-300MB of RAM. If we presume we are working on nodes with quad-core CPU the amount of RAM needed increases four times, for a node with two quad core CPUs RAM requirements increase eight times, for a node with 3 quad core CPUs - twelve times... This might present a problem if the production environment is a cluster with a high number of CPUs when the required RAM per machine increases drastically. However, I find it unlikely that the limiting factor to execution of the software will be RAM requirements.

Lastly, when trying to analyze 5.6 million frames minimizing Type I ("false positive") and Type II ("false negative") errors is crucial. If we imagine that the software rejects 99% of images, as images without linear features, software would still return 56 000 images possibly containing trails. Each one of those would have to be checked manually to determine if it is

[9]Numpy is the fundamental package for scientific computing in Python. It contains a special N-dimensional array type called ndarray that enables fast computation due to its wrapped C++/FORTRAN implementation. See http://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.html.

a false positive or not. On the other hand severely restricting detection parameters to let through only the most confident of detections would certainly introduce a very biased sample of only the most-brightest, longest and thickest linear features and would therefore be a poor statistical sample unfit for further analysis.

1.4 Why Python?

SDSS, officially, supports and offers solutions only for IDL programming language[10] which comes with a hefty price. Obvious choices are fast, compiled languages such as `C`, `C++`, Google's Go or even Scala. However, while researching I noticed that there are no wrappers or, code rewrites, of the original IDL code for any of them which meant I would have to code everything myself. SDSS IDL library is quite encompassing and to do that would certainly be a daunting task, if not impossible, for me. However, I did find the entire IDL code written or, parts that allowed it, wrapped, as a Python module. Alongside Pythons fast learning curve, this contributed significantly to the decision to choose python as the programing language for this project.

Python is a dynamically and strongly typed, object oriented programming language with high interactive capabilities. It is open source and very popular which means it has a plethora of available tools and easy-to-find support. In retrospective the meritorious efforts of other Python users when it comes to programming support and education is the main reason LFDS was written. Python, additionally, offers incredible introspective abilities which are invaluable as a tool for debugging. Because it is an interpreted language it will alleviate me from a series of non-obvious environment dependencies that accompany compiled languages. This meant I could code on a PC, with all the benefits of GUIs, debuggers, IDEs etc. and expect it to work with minimal adjustments, on any other machine with installed Python and necessary modules.

Unfortunately interpreted languages come with serious performance costs. Python is slower than `C++` by a factor of 10 to a 150 times, depending on the task at hand[11]. Other programs usually rang slower than `C++`, with exception of C, but not by much and not always. Because `C++` is so easily wrapped for Python[12] and I had the most experience with it, I had decided that **any** and **all** demanding operations must be written in `C++` in order to shorten the execution time. There are couple of ways of doing this. Easiest way is through a module as a mediator, which thankfully exist in abundance

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

[11] http://benchmarksgame.alioth.debian.org/u32q/compare.php?lang=python3&lang2=gcc.

[12] there is even a way to write C++ code directly in Python source code and then compile it just-in-time with numpy's weave

for Python. Second way is by wrapping my own code either by "weaveing" it in Python source itself, or by using outside helping tools such as SWIG.

Additionaly, the choice of Python proved to be fortuitous as information spread that Large Synoptic Sky Survey (LSST) was to be coded entirely in Python. LSST, currenlty under construction in Chile, is a scaled-up heir of SDSST. It will use an 8.4m telescope and a 3 200 megapixel camera to cover the entire visible sky in just a few nights. It is predicted that LSST will produce 15 TB of data each night. Considering that the predicted field of view of LSST is three times that of SDSS, it is a far better candidate for telescopic meteor search than SDSS is.

Chapter 2

Linear Feature Detection Software

In the previous section-1.3 I tried to provide an overview of all the problems I will have to solve and think about in advance while writing the linear feature detection software (LFDS). I believe section-1.3 can shed a lot of light on topics discussed here.

In this chapter I will attempt to, as briefly as possible, describe the most important modules used in my program, their function in my program and explain how a few selected functions, that I consider important, work. After that follows a general layout of my software and reasons it is organized the way it is. I will try to describe everything LFDS can do, how to use it, all the benefits and downfalls of current solutions and future prospects for developing. Finally I will try to explain the internals of the module responsible for feature detection, how it works, what a normal frame file goes through when it is being processed and its current performance achievements.

LFDS main purpose is to detect linear features on images and extract data describing these features and images they were located on. There are two main ways LFDS was intended to run. One is in a large-scale production environment described in section-2.2.2 where only the *detecttrails* module, described in section-2.2.5, is needed and the other is locally where user can exploit all available modules. To run LFDS in the production enviroment see section-2.2.2. To run a job locally user should run the `start.sh` scrip located in the home folder of LFDS. Contents of the `start.sh` script are displayed in listing-2.1. Each user should modify the `BOSS` environmental variable and python version (s)he is using. Locally `start.sh` script will run a python-idle, its default GUI, in an environment that enables the *sdss*, section-2.1.1, to work.

Listing 2.1: start.sh

```
#!/bin/bash
##Copy the exact SDSS tree structure from boss onwards,
##somewhere on your machine and point BOSS to that address
export BOSS=~/Desktop/boss
export FITSDMP=~/Desktop/fits_dump/
##change following env.var. at your own risk:
export PHOTO_REDUX=$BOSS/photo/redux
export BOSS_PHOTOOBJ=$BOSS/photoObj
export BOSS_CAS=$BOSS/CAS
##Feel free to change your Python version
##not compatible to Python3
idle-python2.7&
```

2.1 Dependencies

Less known non-built-in modules are described in their respective subsections. Additionally there is a set of popularly used non-built-in modules that do not need any special attention such as: *numpy*, *scipy*, *AstroPy*, *matplotlib* and *PIL*.

2.1.1 Erin Sheldon's SDSSPY

SDSSPY[1] module consists of 10 different sub-modules: *astrom*, *atlas*, *family*, *files*, *flags*, *util*, *yanny* and *window*. Its dependencies are numpy and *esutil*. Because I will be using a lot of modules, but at the same time want to stay user-friendly I will be getting rid of the modules I will not need and do not use. Therefore I downloaded *esutil*[2] and *sdsspy*[3] modules and started deleting files and dependencies I will not need. From *sdsspy* I deleted everything except *files*, *util* and *yanny* modules. I nested the *esutil* module folder within the *sdsspy* module folder and deleted *fits*, *random* and *recfile* modules from it. I wanted to avoid having to alter the PYTHONPATH environmental variable to be dependent on the location of the *esutil* folder, so all the import statements in *sdsspy* have to be changed to relative import statements, as shown for *files* module by listings 2.2 and 2.3.

Listing 2.2: original SDSSPY files module

```
import esutil
from esutil.ostools import path_join
from esutil.numpy_util import where1

import sdsspy
from .util import FILTERNUM, FILTERCHARS
```

[1] https://code.google.com/p/sdsspy
[2] http://code.google.com/p/esutil/
[3] http://code.google.com/p/sdsspy/

Listing 2.3: edited SDSSPY files module

```
1 from esutil.ostools import path_join
2 from esutil.numpy_util import where1
3 from esutil import io
4 from esutil import ostools
5 from esutil import sqlite_util
6
7 from .util import FILTERNUM, FILTERCHARS
```

Notice how the *esutil* is no longer a global name. Now I am accessing the folder within the *sdsspy* module folder, not the *esutil* module installed in the default `\usr\local\lib\python2.7\dist-packages`. This also means that all the functions referencing *esutil* module will have to change to reflect this. As an example, again, let us use the *files* module. It contains 3 functions that need to be updated: read, _glob_hdfs_pattern and array2sqlite. Shortest among them is the _glob_hdfs_pattern, so let us display the change that has to be made in listings 2.4 and 2.5 (line 4).

Listing 2.4: original _glob_hdfs_pattern function

```
1 def _glob_hdfs_pattern(pattern):
2     # must make an external call
3     command = "hadoop fs -ls "+pattern+"|awk 'NF==8{print $8}'"
4     exit_code,stdo,stde=esutil.ostools.exec_process(command)
5     if exit_code != 0:
6         raise ValueError("Error executing command '%s': '%s'" %
              (command,stde))
7     fl = stdo.split('\n')
8     fl = ['hdfs://'+f for f in fl if f != '']
9     return fl
```

Listing 2.5: edited _glob_hdfs_pattern function

```
1 def _glob_hdfs_pattern(pattern):
2     # must make an external call
3     command = "hadoop fs -ls "+pattern+"|awk 'NF==8{print $8}'"
4     exit_code,stdo,stde=ostools.exec_process(command)
5     if exit_code != 0:
6         raise ValueError("Error executing command '%s': '%s'" %
              (command,stde))
7     fl = stdo.split('\n')
8     fl = ['hdfs://'+f for f in fl if f != '']
9     return fl
```

These types of changes have to happen in all files that rely on *esutil* module. This way I removed the need to install *esutil* and *sdsspy*. I can now place *sdsspy*, renamed into *sdss*, where I need it and import it straight from there. *files* module basically does everything described in section-1.2.3. It creates filenames, folder names, retrieves lists of runs from which startfields and endfields can be read, expands *sdss* environmental variables etc... One crucial design decision that allowed this was the fact that *sdsspy*

uses environmental variables to locate the folders, and those are easily set-up in any *nix environment.

2.1.2 Erin Sheldon's FITSIO

FITSIO[4] is a module for handling FITS file input/output actions. This is a wrapper of NASA's cfitsio library of C and Fortran routines used to hide the internal complexities of FITS files and enable users high-level fits file handling. *FITSIO* has a couple of lower-level function used for quick reading of fits files as well as a very powerful fits class. *FITSIO*'s last major commit was in 2013, but the repo is still active, signaling that it is a very stable module. I have not bundled this module, nor do I plan to, because it contains C and Fortran files that have to be compiled on the system itself.

Originally I used *PyFITS* module developed by Space Telescope Science Institute (STScI). However, I have repeatedly run into errors and incompatibilities when reading SDSS's non-frame fits formats. Meanwhile (~2010), an open-source package *AstroPy* began development.*AstroPy*'s primary goal is to unify all currently availible modules for astronomy under a single name. By community decision, *PyFITS* became the primary module for handling FITS files. Since then *PyFITS* has restarted its developing and is currently bundled with the *AstroPy* package. Because of the AstroPy's popularity and the fact I already use it in my program, it would be prudent and user-friendly to ensure that the user has a choice between *FITSIO* and *AstroPy*. This can be handled by a try-catch block import statements that check for the existance of either package but is currently not yet supported.

2.1.3 OpenCV

Open Source Computer Vision[5] is a Python library, originally written in C but ported to Python, Java, `C++` and others. OpenCV is immensely powerful; it can even analyze video input in real time, and is used as a solution in industrial and commercial software. OpenCV is the cornerstone of this whole project. Without the endless list of extremely well optimized functions within OpenCV, writing this project in python would be impossible. Unfortunately it is the least user-friendly module/library from the list of used modules. With poor documentation and my, at the time, low level of entry knowledge it was hard to learn how to use it.

There are a number of functions I use to remove noise (erosion and dilatation), enhance dim features (histogram equalization) and generally clean up the image before I ever get to detecting for the presence of lines, many more functions I use to confirm if the detection is plausible or not (Canny edge, contours, minimal area rectangle fitting), but to delve into

[4] https://github.com/esheldon/fitsio
[5] http://opencv.org/

details of each of them here would be impossible. The bottom line is that the effect of all these functions can be explained in a couple of sentences and that they all serve to prepare the image for the line detection algorithm thus, I will provide only a short run-down of their functionality and devote more time to the line detecting algorithm. Imagine that for a test image we use a black image with a white ring in the middle.

2.1.3.1 Erosion and Dilatation

Erosion and dilatation are morphological operators. They both need a kernel that they drag over the image. Kernel is a matrix with a defined anchoring element, usually its center. A new value of the element located in the anchor point is decided by its neighbouring elements for which the kernel values are not zero.
Erosion will assign the anchored pixel the minimal value found in that pixels kernel neighbourhood.
Dilatation will assign the anchored pixel the maximal value found in that pixels kernel neigbourhood.
Various combinations of Erosion and Dilatation can result in other morphological operators such as opening, closing, blackhat, tophat and morphological gradient.

2.1.3.2 Histogram equalization

Histogram equalization is a technique for adjusting contrast. Imagine our test image is not completely white and black. Imagine our ring was bright gray, i.e. intensity value 50, in the middle and darker gray, intensity value 30, at its edges. Histogram equalization would register that intensity value "50" was the largest such value and assign it the maximal possible value, 255 for grayscale images. The rest of the gray shades would then be assigned their new values with respect to the total cumulative distribution function (CDF). Because new values are calculated based on the CDF, equalization differs from normalization.

2.1.3.3 Canny edge detection

Canny edge detection is a method of finding edges in a picture first developed by Canny (1986). First the image is blurred with a 5x5 Gaussian blur matrix. Using a gradient kernel (Sobel, Roberts cross, Prewitt) an intensity gradient for each pixel is found and that gradient gets proclaimed an "edge". Edge is then rounded in one of 4 directions (2 diagonal, vertical or horizontal) after which so called non-maximum suppression is applied. All edges are checked to see if they are indeed a local maximum in the direction of the gradient by comparing them to the neighbouring pixels. Remaining edges are filtered through two user defined thresholds to remove noise-induced edges. This is

known as hysteresis thresholding. Those pixels that are above the higher threshold are automatically declared to be actual edges. Pixels bellow the lower threshold are automatically discarded. Remaining "in-between" pixels are judged by the connectivity criterion. If the in-between pixel can be connected through any number of steps to a pixel above the higher threshold it is declared an edge. In the case pixel can not be shown to be connected to any of edges above the threshold, it is discarded. Applied to our image its result would be 2 concentric circles, one following the outer edge of our ring, one following the inner.

2.1.3.4 Contours detection

A contour is defined as a closed edge. This method, first developed by Suzuki and Keiichi (1985), takes edges returned by Canny and finds only those among them that form a closed shape. There are two important parameters to OpenCv's contour function; its mode and method. Mode stands for contour retrieval mode of which there are four types shown in table-2.1. Method stands for contour detection method approximations. Methods and their meanings are listed in table-2.2.

Mode name	Mode description
RETR_EXTERNAL	retrieves only the extreme outer contours.
RETR_LIST	retrieves all of the contours without any hierarchy.
RETR_CCOMP	retrieves all of the contours and organizes them into a two-level hierarchy, external and internal boundaries.
RETR_TREE	retrieves all of the contours and reconstructs a full hierarchy tree.

Table 2.1: Contours mode options.

Method name	Method description
CHAIN_APPROX_NONE	stores all contour points.
CHAIN_APPROX_SIMPLE	stores only the end points.
CHAIN_APPROX_TC89_L1 CHAIN_APPROX_TC89_KCOS	applies one of the flavours of the Teh and Chin (1989) chain approximation algorithm.

Table 2.2: Contours method options.

2.1.3.5 Minimal Area Rectangle

This a function that can look through the contours and fit on them rectangles of minimal surface area that still encompass the entire contour. These

rectangles can have arbitrary rotation. A fast, $O(n)$, method for finding minimal area rectangles was first proposed by Toussaint (1983). The author describes the following procedure for finding minimal area rectangles: first, vertices of the objects minimal and maximal x and y coordinates are found and calipers are constructed from the points of intersection of those vertices. These calipers are rotated around the object. Because the minimal area rectangle enclosing a convex polygon will have a side collinear to one of the edges of the polygon, we only have to check the angles the calipers close with the polygon sides. The area of each rectangle formed by the two constructed calipers is calculated and compared until a minimal result is found.

2.1.3.6 Hough transform

Hough transform (Hough, 1959) refers to a feature extraction technique used in computer vision that is based on a voting procedure that favors certain class of shapes. In Cartesian coordinates a line can be represented by an equation in a so called slope intercept form, shown in equation-2.1. By parameterizing the line slope, m, and intercept parameter b, we can rewrite the equation-2.1 as equation-2.2 and by rearranging that equation we get to the final form seen in equation-2.3.

$$y = mx + b \tag{2.1}$$

$$y = -\frac{cos(\theta)}{\sin(\theta)}x + \frac{r}{\sin(\theta)} \tag{2.2}$$

$$r = x\cos(\theta) + y\sin(\theta) \tag{2.3}$$

The transformed coordinate space which has θ and r as its axes is called Hough space. If for a given (x_0, y_0) in Cartesian coordinates we plot a family of lines going through that point we get a sinusoid. 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. Thus we reduced the problem of detecting a line in an image to a problem of detecting points in Hough space. Once we detect the local maxima of Hough space, we can do an inverse transform, by using equation-2.2, to get the corresponding line parameters in Cartesian space.

Basic Hough transformation algorithm is fairly easy to reproduce. Set a threshold value that determines which pixels on an image are considered "active", i.e. whose intensity is greater than threshold. For each active pixel, which sets our (x_0, y_0), we draw a family of lines. Each consecutive line in a family has its θ increased by a preset value `theta_res` determining the resolution we want. For each θ, we calculate r and increase the vote in Hough space at (r, θ) for one. Once all active pixels have been exhausted we need to search through the coordinates of Hough space for accumulators

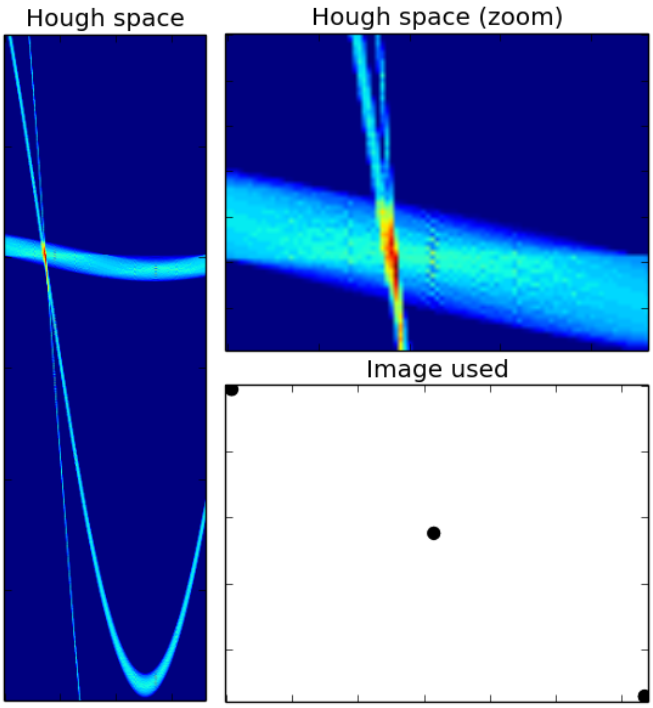


Figure 2.1: Hough space of an image with 3 dots.

with the largest amount of votes. This is not hard to do, however a lot of subtlety can lie in the way these local maxima are found.

Figure-2.1 shows why careful determination of max. accumulator coordinates is so important. On the zoomed section of Hough space (top right) we see that there is actually a large number of highly active accumulators in the neighbourhood of our detection. Each of those will fit a line that passes through the three dots with varying precision. A proper centroid location method would probably yield the best (r, θ) values, but considering how symmetrically spaced the active accumulators are around the actual value, a good, computationally less complex, approach would be averaging a set of top n active accumulators. Notice, as well, how pronounced and tight are sinusoids left by full dots. Dots are in fact disks. Within them there is a lot of neighbouring active pixels. All of them will produce a similar r and θ and therefore leave a very distinguishable compact track. It is important to notice that the line doesn not actually exist, but a line is detected anyhow.

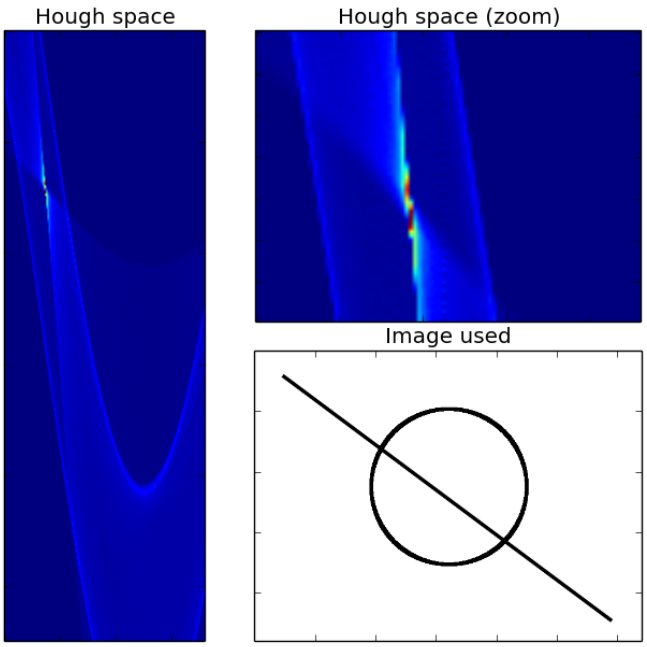


Figure 2.2: Hough space of an image with a circle and a line.

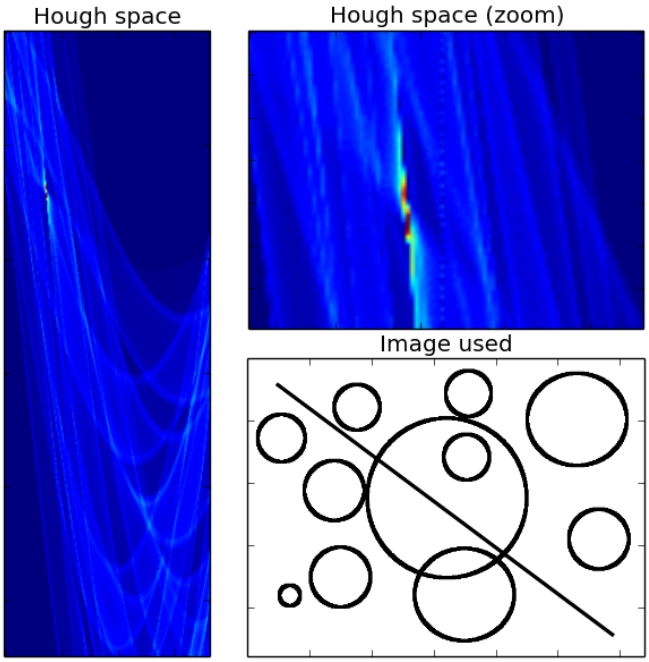


Figure 2.3: Hough space of an image with a line and lot of circles.

Figures 2.2 and 2.3 show the opposite case of a disk. We see that, unlike the filled dots, circles and generally objects that are not filled in produce wide, less distinguishable, tracks in the Hough space. Additionaly, figure-2.3 shows how robust and unaffected by noise Hough transform is. Line accumulators remained as visible as in the case of a single circle, figure-2.2. It would seem it is not as important to maintain the complete trail length as long as parts left behind are "filled out". This also means that any disk-like objects (i.e. stars, galaxies, nebulae) will have to be "hollowed" out or completely removed, otherwise I risk a false detection. If a couple of bigger, brighter stars laid on the same apparent line, Hough might detect them instead of the actual line, as was the case in figure-2.1.

2.2 LFDS Modules

LFDS is divided into five top modules: *createjobs*, *errors*, *results*, *analyze* and *detecttrails* as shown on figure-2.4. Currently the following naming convention is practiced:

- module names are completely lowercase strings, no special characters,
- classes are CamelCase capitalized, no special characters,
- methods are lowerCamelCase capitalized, underscore and dunder allowed only as a part of Python syntax,
- functions are completely lowercase; words are separated with an underscore.

Often the names of classes will be the same as the names of modules, different naming conventions still applied though. In fact the only case where this isn not true, are the utility files. End idea is to keep the namespace clean by exposing, if possible, only a single class per module. Exposed class should be able to reproduce all major functionalities of that module. This is done so that once I convert LFDS to a full-fledged python package I can expose all the functionality from all the modules under a single name. Currently LFDS is still not a package and all the modules have to be imported by themselves. This can lead to confusing situations where you can influence an imported module1 by changing it, accidentally or not, through module2.module1 when module2 is importing module1. Currently this is not a huge problem but since I wrote the *imagechecker* GUI, I have been considering the possibility to expose more of the module's functionality through it. In that case I would have to expose certain objects, i.e. Results and Errors instances, on a global level. That way I could access them through the *imagechecker* GUI, interact with them, return to the Python prompt and still have them, changed, and ready for further processing from within the terminal. In that case this will most likely present a problem.

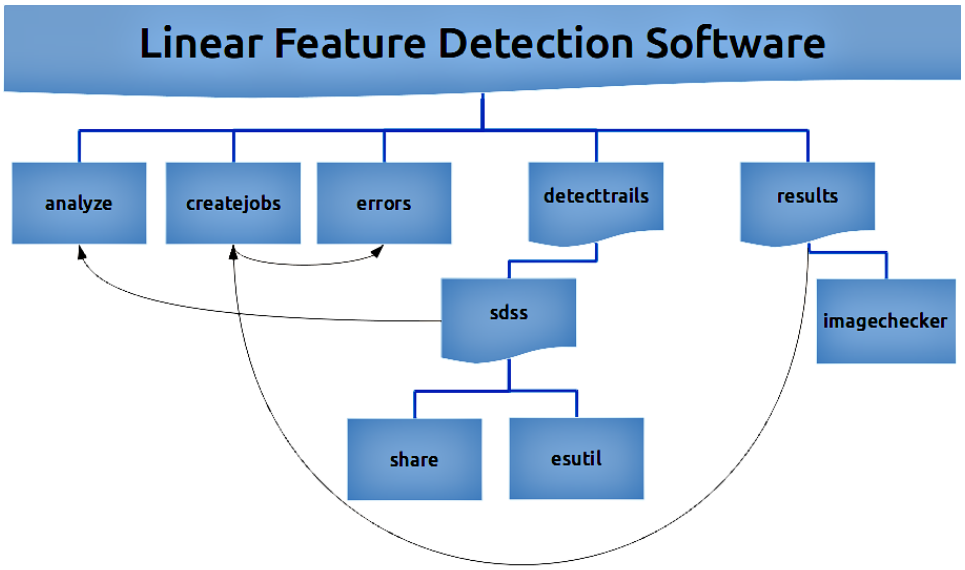


Figure 2.4: Modules and submodules of LFDS

Generally I strive to make each module independent of each other and dependent models are generally "stacked" such that the parent module contains the dependancy. Therefore *sdss* contains *esutil*, and *detecttrails* contains *sdss* module. However, in order to analyze the results it is often necessary to create filenames and locations which is why *analyze* still imports *sdss*. *createjobs* module imports *results* module because Jobs class can be instantiated by Results object in order to create new jobs containing only the frames found in the Results instance used. At the same time *errors* module can use *createjobs*, since parameters can be estimated based on the error, to generate a new set of jobs automatically. What the jobs are and why they are used is explained in more detail in section-2.2.2.

2.2.1 analyse

This module deals mainly with analyzing the end results. It is a utility module that is barely in its infancy. Ideally this module would expose to the user a set of generalized functions capable of plotting any value vs any other value of the results, it would be capable of producing a set of useful graphs that the user could create from within the *results* module to inspect the validity of his results and it was also envisioned as a module to wrap various statistics tools useful for the *results* module. Currently it can produce certain fixed plots of our preliminary results, has certain capabilities to make plots from yanny files (see section-1.2.3) but that is about it. Issues that are mostly plaguing me in this module are all the extra dependencies introduced by matplotlib plotting module, especially when it comes to

postscript, TEXand metafonts. I am not certain if I will keep this module at the time I convert LFDS into a package or not, but right now it is useful as a tools dumping grounds for scripts and tools I currently use for displaying results.

2.2.2 createjobs

To explain the purpose of this module I have to briefly describe the production environment my program is running in. Currently LFDS is run in Belgrade on a cluster called Fermi. Fermi cluster has a master node, 12 computing nodes and 2 storage nodes. Master node, called fermi, administers the computing nodes, handles file transfers, controls and executes programs on the computing nodes and is the main communication channel from users to the computing nodes. Computing nodes are called fermi-node and are labeled 1-12. Each fermi-node has 24GB RAM, 1TB HDD, 2 Intel Xeon X5675 3GHz six core CPUs for a total of 12 cores and 2 Nvidia GPU M2090 Computing Modules based on Fermi CUDA architecture. Storage nodes are named fermi-stor1 and 2. Together they have a total of 6 HDDs 4TB each for a total of 24TB, which is enough to store the files we need (see section-1.3).

Because this is a cluster and not a "regular" computer, to execute a program user has to submit jobs through the master node. A job is a sequence of control statements that represent a single "unit of work" for an Operating System (OS). Job describes what environment the OS will provide for the execution of one or more programs. A series of jobs, or programs, executing without the need of user intervention is called a batch. That means that in a batch all input parameters for the execution of jobs must be provided beforehand through scripts, command-line arguments or control files. (T)erascale (O)pen-source (R)esource and (Que)ue Manager or TORQUE provides control over batch jobs and distributed computing services on fermi. TORQUE uses control files to provide all the necessary environment, data and execution parameters to a job in the form of a distributed queuing system file (dqs). In listing 2.6 I show a generic dqs file that is used by *createjobs* module. In essence the actual and generic dqs files are the same except all the capitalized strings are replaced with actual values.

Listing 2.6: Generic dqs file

```
#!/usr/bin/ksh
#PBS -N JOBNAME
#PBS -S /usr/bin/ksh
#PBS -q QUEUE
#PBS -l nodes=NODEFLAG:ppn=PPN
#PBS -l walltime=WALLCLOCK,cput=CPUTIME
#PBS -m e
#QSUB -eo -me
cd ~
user=`whoami`
hss=`hostname`

if  [ "$PBS_ENVIRONMENT" != "" ] ; then
 TMPJOB_ID=$PBS_JOBID.$$
 JOB_ID=${TMPJOB_ID%%[!0-9]*}.$$
 ARC=`uname`
fi
nodefile=$PBS_NODEFILE
if [ -r $nodefile ] ; then
    nodes=$(sort $nodefile | uniq)
else
    nodes=localhost
fi

export FITSDMP=/scratch/$hss/$user/fits_dump
export BOSS=/scratch1/FERMINODE/dr9/boss
export PHOTO_REDUX=$BOSS/photo/redux
export BOSS_PHOTOOBJ=$BOSS/photoObj
mkdir -p  /scratch/$hss/$user
mkdir -p  /scratch/$hss/$user/test_trails
mkdir -p /scratch/$hss/$user/fits_dump
mkdir -p /home/fermi/$user/run_results/
mkdir -p /home/fermi/$user/run_results/$JOB_ID

cd /scratch/$hss/$user/test_trails
mkdir -p  /scratch/$hss/$user/test_trails/$JOB_ID
cd $JOB_ID

echo $nodes >nodes
echo $PBS_EXEC_HOST >aaa2
set >aaa3 #contains host parameters
cp /home/fermi/$user/run_detect/*.py* /scratch/$hss/$user/
    test_trails/$JOB_ID/
mkdir sdss
cp -r /home/fermi/$user/run_detect/sdss/* sdss/
source ~/.bashrc

COMMAND

cp *.txt /home/fermi/$user/run_results/$JOB_ID
rm a* nodes *py*
rm -rf sdss
```

LFDS is not designed to run in parallel but concurrently on disjoint subsets of data. Taking into account the amount of information user has to provide per dqs file, and the amount of jobs I would be starting, writing each by hand was out of the question. In the beginning I used various shell tools, such as sed and grep, wrapped in python scripts to change dqs files. Soon I found that approach too restricting, time consuming and it produced too many errors. Therefore, I wrote *createjobs* module that is able to write these dqs files with the flexibility I required.

The main pillar for job creating is Jobs class. In the background it uses writer module that is hidden from the user to write the parameters into a template. Default template is called "generic", see listing-2.6, and can be found in the same folder *createjobs* module resides in. Since the environment paths and cp paths can not be changed through Jobs class it might necessary to edit the template, or a new one can be sent in its place by specifying template_path variable at Jobs instantiation time. When writing your own template it is important to specify all parameters, shown in table-2.3, as uppercase single words.

Parameter name	Description
JOBNAME	Name of the job as seen by qstat
QUEUE	TORQUE queue type (parallel, serial, standard). Note that wallclock and cputime limitations depend on queue type.
NODEFLAG	Node on which the job will be executed. By default set to "1" which lets Maui optimize job execution. Read Jobs docstring for details.
PPN	Sets max. number of processors per node program is allowed to execute on. Default is 3. Note that because of the RAM requirements, large ppn might initiate swapping.
WALLCLOCK	Sets maximal allowed real-world execution time. Default: 24h.
CPUTIME	Sets maximal allowed time spent executing on CPU. Default: 48h.
COMMAND	Command that gets executed on fermi-node's. See *createjobs* docstring for details.

Table 2.3: Generic dqs file parameter names and description.

There are 3 ways Jobs class is designed to be used. The simplest scenario is just specifying the number of jobs you want as shown in listing-2.7. Jobs will then create a series of jobN.dqs files, where N ranges from 0 to number of jobs specified, N_{jobs}. In each dqs file the COMMAND keyword will be replaced with a series of N_{runs}/N_{jobs} command attributes of Jobs class.

Runs are taken from the runlist yanny file (see section-1.2.3) which should be located in `$BOSS/photo/redux` folder. Only runs with the rerun 301 are taken into consideration. Alongside with the dqs file, another file called batch.sh gets created. That file can be used to launch all the jobs at once by using `source` command in terminal.

Listing 2.7: Jobs use case 1

```
>>> import createjobs as cj
>>> jobs = cj.Jobs(500)
>>> jobs.create()
  There are no runs to create jobs from.
  Creating jobs for all runs in runlist.par file.

  Creating:
      765 jobs with 1 runs per job
      Queue:     standard
      Wallclock: 24:00:00
      Cputime:   48:00:00
      Ppn:       3
      Path:      /home/user/Desktop/.../jobs
```

User will be notified about all important parameters that were set, including path where the files are stored. Notice that we specified 500 jobs to be created but 765 jobs were actually created. This is intentional. If Jobs was instructed to create 100 jobs, it would create 110 jobs with 7 runs per job. Jobs class looks for the next larger whole number divisor to split the jobs between. It is better to send in more jobs than to risk jobs failing because WALLCLOCK or CPUTIME was exceeded.

Second way Jobs class can be used is by sending a list of runs as shown in listing-2.8. This enables user to select which runs specifically, (s)he wants to process.

Listing 2.8: Jobs use case 2

```
  >>> runs = [125, 99, 2888, 1447]
>>> jobs = cj.Jobs(2, runs=runs)
>>> jobs.create()
  Creating:
      2 jobs with 2 runs per job
      Queue:     standard
      Wallclock: 24:00:00
      Cputime:   48:00:00
      Ppn:       3
      Path:      /home/user/Desktop/.../jobs
```

In both examples so far what is actually being executed on the fermi-node is the following COMMAND:

```
python -c "import detecttrails as dt;
          dt.DetectTrails(run=125).process()"
python -c "import detecttrails as dt;
          dt.DetectTrails(run=99).process()"
```

It is possible to edit the command that is going to be executed. Specifying additional keyword arguments (kwargs) to Jobs class helps you utilize DetectTrails class run options. Sent kwargs are applied globally across every job. It is not possible to specify separate kwargs for each job. For example, continuing the example in listing-2.8 we have the following listing-2.9:

Listing 2.9: Jobs use case 2, further specification

```
>>> jobs = cj.Jobs(2, runs=runs, camcol=1)
>>> jobs.create()
```

which would create 2 jobs with 2 runs per job as the above example. But the actual calls to DetectTrails class would now look like:

```
python -c "import detecttrails as dt;
          dt.DetectTrails(run=125, camcol=1).process()"
python -c "import detecttrails as dt;
          dt.DetectTrails(run=99, camcol=1).process()"
```

This would process only the camcol 1 of given runs. The actual processing selection is done by *detecttrails* module described in section-2.2.5. All *createjobs* module does is write the dqs files. Listing-2.10 shows it is possible to utilize the DetectTrails class flexibility even further by specifying the filter to be processed.

Listing 2.10: Jobs use case 2, fully specified.

```
>>> jobs = cj.Jobs(2, runs=runs, camcol=1, filter="i")
```

Processing of a single frame can be done locally even on a relatively weak PC thus I have not extended this feature to cover specifying frames or frame ranges.

Because the DetectTrails class is so flexible, in the sense that apart from the target of execution user can, by changing execution parameters, influence the execution itself, it is possible to even further fine tune your job behaviour. By changing the default COMMAND it is possible to supply additional execution parameters. By default, keyword argument command is set to:

```
python -c "import detecttrails as dt; dt.DetectTrails($).
    process()"
```

Where "$" sign gets expanded by the writer module depending on the parameters sent to Jobs class at instantiation, as seen in listings 2.8, 2.9 and 2.10. Because the "$" sign stands for arguments of DetectTrails class at instantiation, there should **always** be a "$" character present in the command and it should **always** be within the round brackets following DetectTrails. Listing-2.11 shows how much control the user has over writing dqs files and program execution by combining the full specification of execution target and a COMMAND override.

Listing 2.11: Jobs use case, overriding COMMAND

```
>>> jobs = cj.Jobs(2, runs=runs, camcol=1, filter="i")
>>> jobs.command = 'python -c "import detecttrails as dt;'+\
                   'x = dt.DetectTrails($);'           +\
                   'x.params_bright[\'debug\'] = True;'+\
                   'x.process()"\n'
>>> jobs.create()
#or a more user-friendly approach would be a multiline comment
>>> newcommand="""
python -c "import detectrails as dt;
x=dt.DetectTrails($);
x.params_bright['debug'] = True;
x.process"
"""
>>> jobs = cj.Jobs(2, runs=runs, camcol=1, filter="i", command=
    newcommand)
>>> jobs.create()
```

One of the dqs files that would get written by listing-2.11 would have their COMMAND string replaced by the string shown in listing-2.12. The other dqs file would have his COMMAND string replaced with the similar command string, except the selected run identifier would be 99.

Listing 2.12: Example of a command attribute overriding COMMAND string in a dqs file.

```
python -c "import detecttrails as dt;
           x = dt.DetectTrails(run=125,camcol=1,filter=i);
           x.params_bright['debug'] = True;
           x.process()"
```

In the listing-2.11 first example notice that quotation-marks are trice nested as follows ’ (" (\’ \’) ") ’. Outter single-quotes, ’ ,encompass the entire command (python -c....\n), these single quotes are necessary because we have to send a string as our command attribute of Jobs class. Inner double-quotes, " ,enclose the string that will be executed by the python -c command in the actual job#.dqs file. Innermost escaped single-quotes, \’, mark a new string that will get interpreted as an argument inside the string sent to python -c. General useful guidelines are to put

1. the outter-most quotation as single ” marks,
2. everything past "-c" flag in double quotation marks "",
3. further quotation marks should be escaped single quotations,
4. in the case of a single-line string an **explicit** newline character should **always** be at the end.

Since this is really complicated to remember, I recommend using Python’s multiline comment. In that case user only has to worry about having a

single/double quotation marks for everything past the -c flag and then to use double/single quotation marks respectively for any string declared inside. Finally, all of these are also valid options to send to Jobs class for usage case 1, see listing-2.7.

So far I have shown how to create dqs job files for all runs or for some runs-camcol-filter combination. However it is still impossible to create jobs that can handle frames as the basic data specification unit. Solution to this problem is to instantiate a Results object and send it to Jobs class as shown in listing-2.13. Results class and *results* module is explained in more details in section-2.2.4.

Listing 2.13: Jobs use case 3

```
1  >>> import results as res
2  >>> r = res.Results(folderpath="/home/user/.../res")
3  >>> jobs = cj.Jobs(5, runs=r)
4  >>> jobs.create()
5    Creating:
6       6 jobs with 1372 runs per job
7       Queue:     standard
8       Wallclock: 24:00:00
9       Cputime:   48:00:00
10      Ppn:       3
11      Path:      /home/user/.../createjobs/jobs
```

This time it is not runs that are executing but frames. Because frame execution times are much lower than run execution times (or any camcol-filter specification, see section-3.1), a larger number of frames per job can be executed.

2.2.3 errors

Errors are of particular concern in the execution of LFDS. Imagine that a file on one of the disks on fermi-store got corrupted. If that file were to cause an error somewhere in internals of the detection algorithm that error would propagate through the DetectTrails module all the way up until the job execution of that entire run would have to be terminated. To stop such spurious errors from interrupting the execution of a possibly week long job, all errors are suppressed. In order for the user to have full oversight of the job execution and results it is very important to log all errors alongside with their reasons so that they can be appropriately dealt with in the after-processing and results analysis.

detecttrails module currently outputs two files on the fermi-node where it is executed, results.txt and errors.txt. The jobs dqs file instructs the OS to copy (see listing-2.6 line 49) the output files to a folder on the master node named after the unique identifier JOB_ID. That folder gets created only if it does not exist already, see listing-2.6 line 33. This module is designed to read in and display these errors in a more human readable format, groups them

by run-camcol-filter identifiers to help narrow down where the problems may arise, provides some basic statistics about error origin (DetectTrails or Fermi) and is capable of creating new jobs just for the erroneous files. In the future I would like to be able to sort the error reports based on the type of error, provide some basic histogram plots of most general error types (file error, arithmetic error, unknown...) through the analyse module and possibly include these capabilities in the GUI described in section-2.2.4.1. Since currently I am still dealing with a smaller developing sample this module still is not of such a crucial value and has been the least developed (time-vise).

Example of a single error report is shown in listing-2.14. Error report format is as follows: run, camcol, field and filter data are placed in the first line, followed by a 3 recursions deep traceback message, and in the last line the error message itself is printed again.

Listing 2.14: Error format

```
94,3,251,i
Traceback (most recent call last):
  File "detecttrails.py", line 47, in process_field
    **params_removestars)
  File "removestars.py", line 207, in RemoveStars
    field=_field))
  File "removestars.py", line 84, in photoObjRead
    header1, header2 = fitsio.read(path_to_photoOBJ, header="
        True")
ValueError: No extensions have data
No extensions have data
```

2.2.4 results

results module is used to read and interpret the results.txt files. Main class in *results* module is the Results class. Results is a container class for Result objects. Results references the same Result object twice. Once when the Result is read into a list, called lsitr, and the second time when Result is sorted into a dictionary, called dictr. Because it is the same object referenced in both listr and dictr, changing Result in either of them will update the Result object in the other one. There are two ways to read results into Results. First way is by reading them from folders created by the dqs file by providing the Results class a folderpath keyword argument. Second way is by providing the Results class keyword argument results. Variable sent to the results kwarg should be a list of Result objects. This is useful if the user wants to mock a Results objects for couple of frames only so that (s)he could use the *createjobs* module as shown in listing-2.13. The way Result objects are referenced and structure of the dictionary is shown on figure-2.5.

This complication was necessary because a lot of time we want fast access to certain run\camcol\filter\frame and sometimes we want to perform an

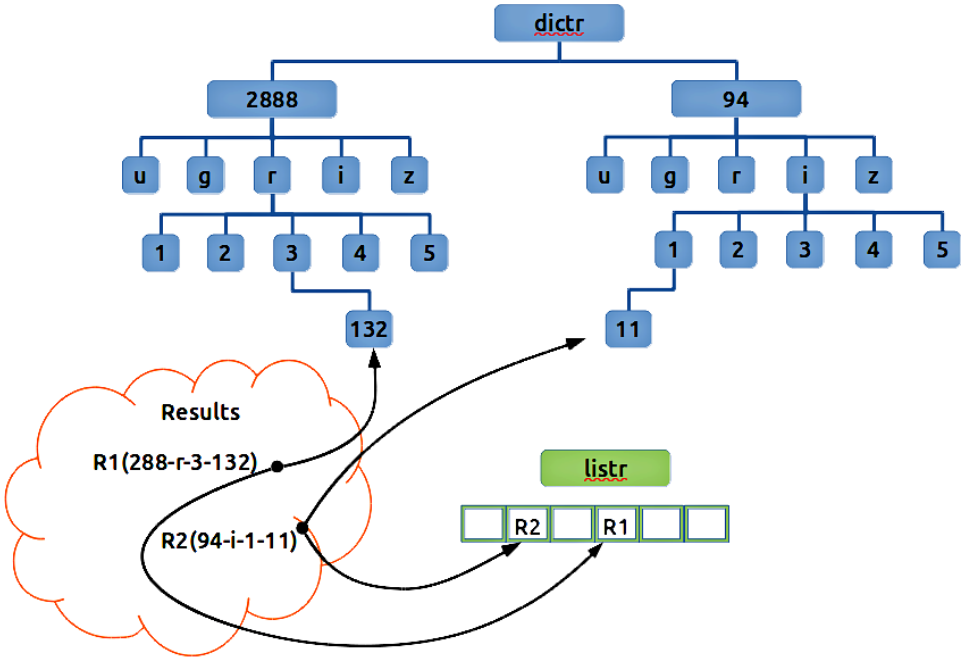


Figure 2.5: Design of Results class.

operation on all Result objects in Results. Accessing method for a list would comprise of searching through the entire list for the wanted Result which is $O(n)$ operations. Unless the list was sorted, at which point a linear relationship between the index and Result objects could be made for a $O(1)$ access time. Until we try to append or remove an Result object from Results, at which point the entire sorting procedure would have to be repeated. Additionally, it is still unclear how the linear function would be determined. For example, a list for all the runs, filters, camcols and frames could be allocated each time a Results object is. However, because trails are a rare phenomenon on images, I am expecting that the actual number of Result objects found would be a very small subset of all images. To open such a large array at each new Results object instantiation would certainly be very memory inefficient. On the other hand just using the Pythons dictionary container is a bad practical setup. The way end-users would have to loop through it[6] to retrieve all the results would prove too time costly to be a viable solution.

Current solution offers users choice between using listr and dictr, at least until such a time I can properly override the access methods and provide my own that would make that decision without the users knowledge. listr

[6] for each run in runs, for each filter in run, for each camcol in run, for each frame in run, is a quadruple nested for loop!

can provide them with a simple single loop mechanisms for processing all results, as well as slices, counting operators, mapping functions etc. dictr can provide them with a $O(1)$ lookup time, because dictionaries in Python are hashed objects,and directed selection operators such as "just that runs i filter".

Unfortunately current solution is not the best either. Users would have to know by hearth that the dictr keys order is [run][filter][camcol][field] and in an ideal case they just should not be bothered by having to understand the internal workings of a class. It is also impossible to select just the desired camcol of a run, or runs, because run and filter keys have to be specified beforehand. Additionally there is an memory overhead of $O(n)$ because of the additional dictionary requirements. Memory overhead is a bit larger than that because I do not have a single dictionary, but a series of multiply-nested ones and it would appear that there are certain additional information stored per Python dictionary thus slightly ruining the $O(n)$ space complexity.

I have recently tried a several attempts at using numpy's void type[7] which can help me facilitate use cases I desire, but it is a hard type to get around with. There is a lot of operators that would have to be added in order for this object to function as intended and they are not all always straight-forward. Predominantly I have issues with the inability to provide a field value not field name as a basis on which to return fields that match the sent field value. It is possible I might have to write extensive code in C++ and wrapping it to get the desired functionality as a python object.

Each Result object is a composition of AstroTime, Frame and Line objects from astrotime, frame and line modules located in the *results* module (not listed in figure-2.4, they are not contained in additional subfolders but are single files). I chose to use a composition instead of inheritance because in reality AstroTime, Frame or Line are in fact subsets of information that define a single Result. By that logic Result is a superclass of them. However in order to code this in a sensible manner, I would have to make Results inherit the three classes and then override therein defined method if necessary. This makes no sense since it implies Results is a subclass of each of the three classes. Main purpose of composition is to create "wholes" out of "parts" and therefore solves this logical conundrum elegantly. The composition model is shown on figure-2.6.

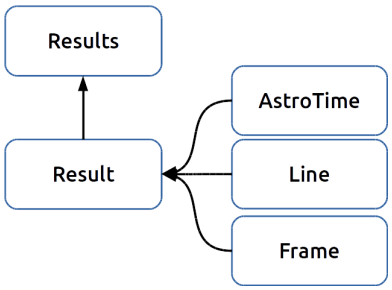


Figure 2.6: Composition diagram of Results class.

[7] A must read http://docs.scipy.org/doc/numpy/reference/arrays.dtypes.html

AstroTime handles conversions from the SDSS tai to mjd and iso times. Note that SDSS tai times are a "tad bit weird" and conversion to mjd is given with mjd = tai/(24*3600). Line class holds line parameters from DetectTrails results. It can calculate m and b line parameters and will likely contain functions that calculate line position on a resized image. This feature would prove useful in *imagechecker* GUI. Frame holds all frame related parameters.

It is also of interest, because the result are kept in a CSV file, to define the layout of the data so it can be written and read across the modules uniformly. It is questionable if the data should be represented by a CSV file or should I have used a proper module, i.e such as JSON, which handles data descriptive files more generally. This is a remnant of the old program that will likely be subject to change in the future depending on the usage of the LFDS. Data is written in the space separated CSV format shown in listing-2.15.

Listing 2.15: Results format

```
run field camcol filter tai crpix1 crpix2 crval1 crval2 cd1_1
    cd1_2 cd2_1 cd2_2 x1 x2 y1 y2
```

First 4 values are the frame file identifiers. Tai represents an Mean Julian Date (MJD) of the moment when the first line of the image was read on the CCD (see section-1.2.1). CRPIX values are the (x, y) coordinates of a reference pixel and are accompanied by the CRVAL values that represent equatorial (ra, dec) coordinates of that same reference pixel. CD1_1 and CD1_2 describe the change of ra per column and row pixel, respectively, while CD2_1 and CD2_2 describe how much does dec change per column and row pixel, respectively. Result object can be instantiated though a string formatted as in listing-2.15, or through a dictionary containing the necessary keywords.

2.2.4.1 imagechecker

imagechecker allows users to keep false positive detections minimal. User is currently expected to go through the output of *detecttrails* "by hand", separating actual trails from false detections visually.

Listing 2.16: Running imagechecker.

```
>>> from results import imagechecker
>>> imagechecker.run()
```

To run *imagechecker* GUI, shown on figure-2.7, user has to execute *imagechecker* module's run command from within Python IDLE as shown in listing-2.16. At the start of the GUI user is faced with two "Select a folder" dialogues. One selects the image folders, while the second one selects the folder containing the output of *detecttrails*.

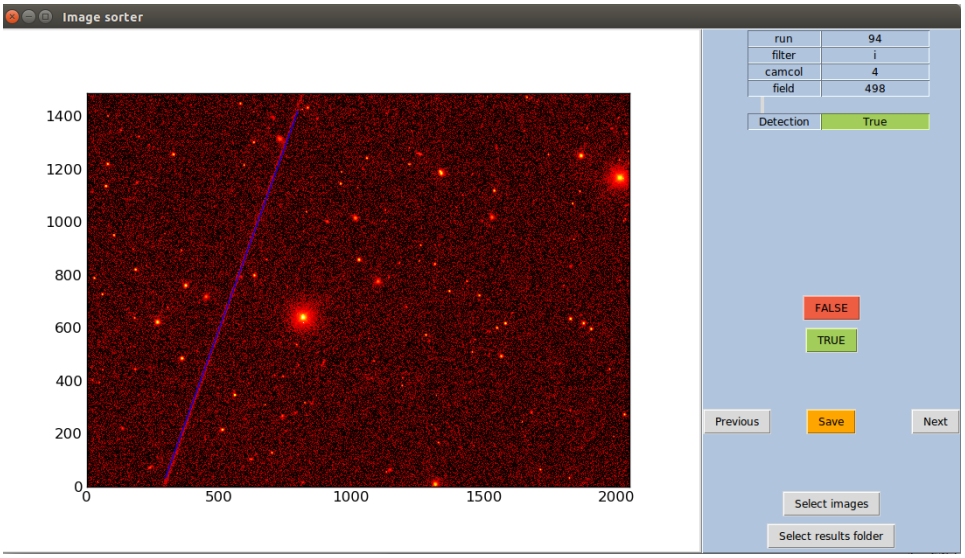


Figure 2.7: imagechecker GUI design.

Images in the image folder do not yet have a firmly established structure. So far I have been working with hand-made test case scenarios. Single main issue with these images is that GUI can only be run locally, Fermi cluster does not have a OS GUI. That means user has to download them to a local PC. As explained in the section-1.3 there are approximately 1.1 million images per filter. If we assume that my program will filter out 99% of the images, that would still leave 15 000 images. A 2048x1489px large png image is about 2.5MB large, if we were to convert all those 15 000 images, assuming a 99% filtering rate, we would have to download approximately 37GB of images to a local computer.

This implies that the images will have to be resized. This is where the problem becomes twofold. Figure-2.8 shows aftermath of a simulated resizing test on details in the image. Original 2048x1489px image with a set of 5 lines with different widths (in order: 100, 50, 25, 12, 6, 3 pixels) was resized to 4 smaller resolutions. We see that along with the expected reduction in the widths of the lines we also incur a loss of line intensity in groups 3 and 4. This seemingly happens when resizing produces a thinner-than-pixel widths in which case, rounding preserves the line but at a cost of its intensity.

Losing details when downsizing is inevitable, however this loss can be minimized if a resampling technique is chosen smartly. Resampling is a process in which the image is convoluted with an operator to produce a new image. If the operator is chosen carefully, that is to say such that the produced image is more suitable for resizing, the end resized image will have less artefacts than the resized image without resampling. Choosing

Figure 2.8: How resizing affects line width and intensity.

a resampling technique, however, is proving very difficult, with variety of choices between interpolation methods (point, box, cubic, bilinear, Hermite, LaGrange, Catmull-Rom...), blurring methods (variations of Gaussian blur) and so called windowing (apodization) filters (Lanczos, Blackman, Bohman, Hann ...). Each approach comes with its advantages and disadvantages when it comes to downsizing. Downsizing will become a part of LFDS once I have sufficient knowledge to confidently write such script(s) and make these decisions. This detail loss is not something that will "erase" the majority of detected lines, but I still maintain that it is not a problem that should be ignored if we want to achieve maximal detection rates.

As mentioned, this is the latest module to be added and therefore is not yet finished. It can preform what is, at this stage, necessary and expected of it but still needs a lot of polishing. I can create the images from the *detecttrails* output, match them with their Result object and display the parameters, I can browse fast through the images to check for trails visually and I can export my edits in a file that matches the Result format described in listing-2.15. However, I have still to display all the Result parameters in a readable manner, add left and right mouse button events that can interactively edit the line parameters, create mosaic script that can download the surrounding frames and display them as one image (written but not yet incorporated in GUI itself) and of course solve the burning issue of detail loss when resizing.

2.2.5 detecttrails

One of the goals of LFDS is to be as generic as possible. But to enable batch image processing for SDSS I rely on SDSS's par and photoObj files (see section-1.2.3). Therefore, there are certain aspects of this module that are only applicable to SDSS folder and file structures. In order to still be able to, later on, add the extra functionality for processing of databases that do not follow the same model, *detecttrails* module is split into *removestars*, *processfield* and *sdss* (see section-2.1.1) modules. Their inter-dependance is shown on figure-2.9.

detecttrails module itself consists of convenience class DetectTrails and a process_field function. DetectTrails holds various execution parameters and

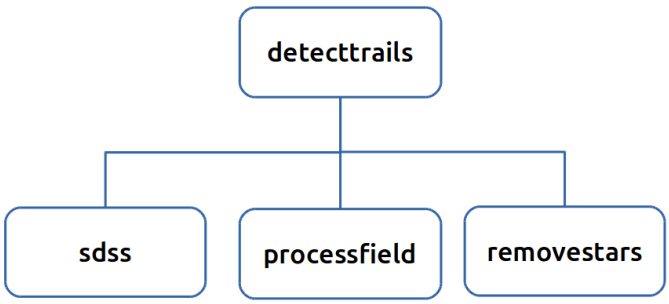


Figure 2.9: detecttrails module layout.

processes frames selected by initial parameters. DetectTrails usage should feel familiar because we have already shown its capabilities in listings in section-2.2.2. First usage case shown in listing-2.17 shows by example how to specify the processing data.

Listing 2.17: DetectTrails instantiation, case 1

```
1 foo = DetectTrails(run=2888)
2 foo = DetectTrails(run=2888 camcol=1)
3 foo = DetectTrails(run=2888, filter='i')
4 foo = DetectTrails(run=2888,camcol=1,filter='i')
5 foo = DetectTrails(run=2888,camcol=1,filter='i',field=139)
6 foo = DetectTrails(camcol=1, field=139)
```

DetectTrails class relies on SDSS par files and folder structure to deduce what data user selected for processing. In hindsight *detecttrails* module should have been coded with a base class that contains the bare minimum (parameters, files...) to be able to execute the detection algorithm, and from that class the current DetectTrails class should be inherited containing additional file handling logic specific to SDSS, preferably under a new name i.e. SDSSDetectTrails. This would enable for faster scripting of different file logic for different databases and would eliminate the need for process_field function which could then be coded as a method of this new class. process_field handles the linear feature detection logic as well as results and error writing logic. LFDS detection logic follows a simple pattern:

1. remove all stars on the images. As described in the section-2.1.3.6 Hough line detection algorithm is very resistant to "hollow" objects but very susceptible "filled" objects. Unfortunately stars and galaxies are such "filled-in" objects. In fact some star are so bright they tend to completely dominate the image as shown in figure-2.10. In order to reduce their impact on our detections it is necessary to completely remove them, or at least partially block them (to "hollow" them out).

2. Search for bright trails. Often bright trails do not need a lot of preprocessing that is required to make dim trails stand out. In fact often

very bright trails tend to get damaged by unecessary processing. Time cost of this search is small enough, according to section-3.1 3.1 times smaller than searching for dim trails, that it pays out to check for them every time before ever continuing further.

3. Search for dim trails. If no bright linear features were found then faint features on the image are boosted in hopes one of them is a trail.

Except the data selection shown in listing-2.17 user can also specify processing parameters as shown in listing-2.18. There are three parameter groups that user can specify, one for removal of stars, one for bright trail detection and one for dim trail detection. Each group is prefixed with a `params_` after which follows the name of the group. Each params is just a python dictionary containing specific keys that are then passed, through process_field, into *processfield* module where those values are used in the detection itself.

Listing 2.18: DetectTrails instantiation, case 2

```
1 foo.params_bright["debug"] = True
2 foo.params_removestars["filter_caps"]["i"] = 20
```

To be able to use DetectTrails efficiently users have to have intimate knowledge of all processing parameters, what they represent and what changing them implies. More detailed explanations are presented in the section-2.2.5.1 and in table-2.4 where I have shown all adjustable execution parameters. Most of the parameter values, except pixscale perhaps, were determined by a heuristical approach where an optimal solution would be guessed on an increasing sample size until the current parameters were determined. They provide the most satisfactory solution, one that contains most of actual trails while minimizing false positive detections (see section-3.2), therefore changing them might have unexpected consequences.

To start the processing of the selected data, with the selected parameters, user must call the process method of DetectTrails instance.

Parameter	Member of params	Description
dilateKernel	bright and dim	Default: 4x4 kernel of ones for bright, 9x9 kernel for dim. See 2.1.3.1
erodeKernel	dim	Default: 3x3 kernel of ones, see 2.1.3.1
contoursMode	bright and dim	Default: RETR_LIST, see 2.1.3.4
contoursMethod	bright and dim	Default: CHAIN_APROX_NONE, see 2.1.3.4
minAreaRectMinLen	bright and dim	Default: 5 for bright, 3 for dim. Once contours have been returned minimal area rectangles are fitted. Length of sides is determined. If either side is shorter than this value, in pixels, rectangle is dismissed as faulty.
lwTresh	bright and dim	Default: 5. Lengths of sides are sorted such that l is always the longer side. If the ratio l/w of sides lengths exceeds the lwTresh rectangle is dismissed as faulty.
houghMethod	bright and dim	Default: 1. Remnant of old OpenCV backend, mandatory but meaningless.
nlinesInSet	bright and dim	Default: 3. Once contours have been returned minimal area rectangles are fitted. Length of sides is determined. If either side is shorter than this value, in pixels, rectangle is dismissed as faulty.
thetaTresh	bright and dim	Default: 0.15. nlinesInSet number of lines are averaged by their thetas, in radians, for both Hough lines fitted. If either set of lines exceeds thetaTresh detection is dismissed as false.
linesetTresh	bright and dim	Default: 0.15. If sets of lines pass thetaTresh but their respective set averages are set apart by more than linesetTresh detection is dismissed.
dro	bright and dim	Default: 20. If respective r averages of line sets are set appart for more than dr than detection is seen as false.
minFlux	dim	Default: 0.02. For bright processing, all negative values are set to 0 before processing. For dim processing, all values of pixel brightness above minFlux are set to zero.
addFlux	dim	Default: 0.5. All other pixels above min_flux are increased by addFlux value. This increases contrast and improves histogram equalization results.
debug	bright and dim	Default: False. Verbose tests output and step by step images are displayed.

pixscale	removestars	Default: 0.396. SDSS pixel scale is 0.396 arcseconds/pixel. Used to convert the petrosian radius to pixel dimensions.
defaultxy	removestars	Default: 20. Half of the length, in pixels, of the side of a square that is drawn over stars. Used if there is no petrosian radius or if the half-length of square side is larger than maxxy.
maxxy	removestars	Default: 60. Maximal allowed half-length of square side. Maximal covered area on the image is 120x120px.
filter_caps[ugriz]	removestars	Default: 22 for all filters, prone to change. Stars on the image that are dimmer than filter caps will not be removed.
magcount	removestars	Default: 3. Maximal allowed number of filters in which magnitude difference is larger than maxmagdiff.
maxmagdiff	removestars	Default: 3. Maximal allowed difference in magnitude between two filters

Table 2.4: All adjustable processing parameters and their description.

2.2.5.1 processfield

processfield module is **the** work-horse of LFDS. This module is written to be as generically applicable as possible, therefore it operates on the smallest subset of data possible. In SDSS terminology that subset is called a frame, but in any astronomical observation, even in those that do not use the drift scan method, it is still the single field imaged through the telescope with a CCD. The example we will be using is just the i filter of the composite image shown in figure-2.10.

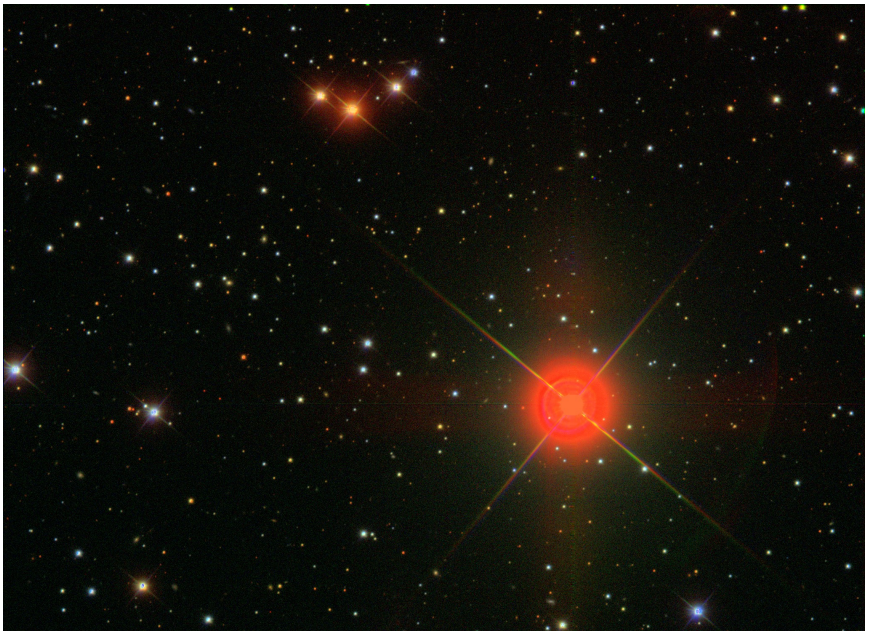


Figure 2.10: frame-irg-002888-1-0017, a composit image made out of i, r and g filters.

processfield module is consisted of a series of functions but only two of them are important to users: process_field_bright and process_field_dim. They are generally expected to always be called in the same sequential manner as described in section-2.2.5 but in some cases user might want more flexibility therefore neither of them is hidden. Both bright and dim functions take in an image in the form of a numpy array, to avoid the issues attributed to different file formats, and a params dictionary. First thing both functions do is to convert the image into a 8 bit single channel grayscale image. There are important differences in how they do it though. Function process_field_bright converts all negative elements to zero before convertScaleAbs function is applied as shown in listing-2.19. Function process_field_dim converts all values smaller than minFlux value to zero after which addFlux value is added to all remaining nonzero pixels as shown in listing-2.20.

Listing 2.19: process_field_bright, step 1

```
1 img[img<0]=0
2 gray_image = cv2.convertScaleAbs(img)
```

Listing 2.20: process_field_dim, step 1

```
1 img[img<minFlux]=0
2 img[img>0]+=addFlux
3 gray_image = cv2.convertScaleAbs(img)
```

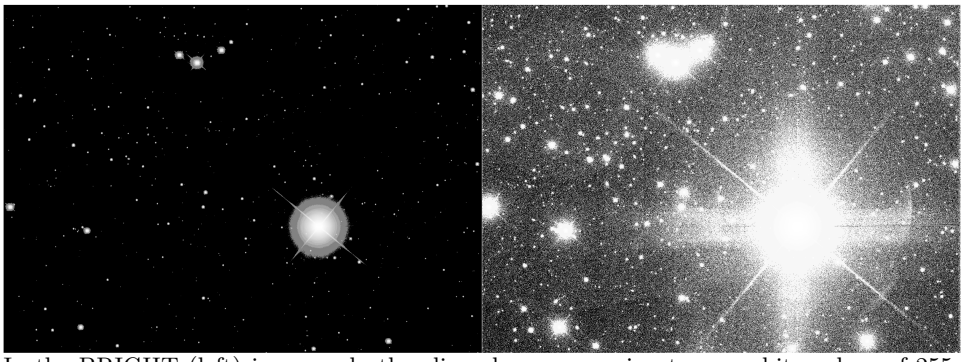
convertScaleAbs function from OpenCv performs the following calculation:

$$dst(I) = saturate\_cast < uchar > (\alpha * src(I) + \beta)$$

This function scales the original image pixels intensity by the optional factor of α, then scales the original image pixel intensities by the optional additive factor β, after which it performs a cast of any type into a unsigned 8 bit corresponding type. In my case factor α is 1 and β is 0, therefore the only conversion that happens is from a 32 bit float type into an unsigned 8 bit integer type. This also means that all the values outside the 8 bit range, that is 255, are clipped. In essence, all values over 255 become 255. This conversion from 1 channel 32 bit float image to 1 channel 8 bit integer image, known as grayscale image, is necessary because OpenCv's functions often have limitations on the type of images they can work on. This also means that pixels are binned from a 32 bit float representation, which can convey a lot more information, to 8bit int representation which can not.

Following the conversion to a grayscale image, both functions preform histogram equalization. Due to the saturate_cast call a lot of the low brightness pixels (those with brightness value less than 0.5) will get rounded down to zero. Histogram equalization for process_field_bright will never act on dim objects unlike in process_field_dim. Because we added addFlux, not a small value in this context, a lot of low brightness level pixels will now be rounded to higer values and will be candidates for the histogram equalization. Since we have "tipped" the histogram in process_field_dim to the darker side, histogram equalization will brighten the entire image by a considerable amount. Unfortunately this also means that noise is potentiated as well. As a result of aforementioned processing, grayscale and histogram equalization, images shown in figure-2.11 are recovered.

Operations such as Hough line, Canny edge, contours detections etc. are per-pixel operations at best. A lot of active pixels in the image, as is the case with figure-2.11, means a lot of processing time. To combat this time requirements and to increase the chances of detecting the actual line, in process_field_dim, noise is removed. This same step is not done for process_field_bright. For detecting bright trails image is additionally dilated (see section-2.1.3.1) in order to even further accentuate the bright objects. Contrary to process_field_bright for dim trails the entire image is eroded before it is dilated (see section-2.1.3.1). This is usually known as a closing operator, however unlike with the closing operator the erosion and dilatation kernels are not symmetrical. It is important to remember that we still have not done much to "thicken" our dim objects and that they could still be destroyed by erosion. This is why the erosion kernel must remain small. At the same time, logic mandates, that a perfect case of noise would be a solitary pixel, a pixel that is not surrounded by a single neighbouring pixel. This provides us with the lower limit for our kernel, a 3x3 matrix anchored



In the BRIGHT (left) image only the clipped areas remain at pure white values of 255. Lowest brightness areas are "pushed" completely to black, so called shadow-clipping. This destroys all low brightness details while mid-tones and high-tones are preserved and "stretched" by histogram equalization beyond information they can convey, thus producing these cascading areas of same gray tone around the pure white areas. In the DIM image (right) highlights get clipped, destroying the information on the bright end of the histogram but preserving and enhancing the information of the low tonal histogram region. End result is a tonally compressed overexposed image which shows the dim features more prominently than a shadow clipped image.

Figure 2.11: Processing result of BRIGHT on the left and DIM on the right.

in the center. A fairly large dilatation kernel (9x9) is used after that to dilate the image and exaggerate all objects that remained. This value has an heuristically determined top limit to how much an image could be dilated before a significant amount of false positives start appearing. In the case of process_field_bright dilatation kernel is a modest 4x4 matrix. This second step in processing consists of histogram equalization, erosion and dilatation for process_field_dim and just dilatation for process_field_bright. Those are simple commands in OpenCV module shown in listings 2.21 and 2.22.

Listing 2.21: process_field_bright, step 2

```
1 equ = cv2.equalizeHist(gray_image)
2 equ = cv2.dilate(opening, dilateKernel)
```

Listing 2.22: process_field_dim, step 2

```
1 equ = cv2.equalizeHist(gray_image)
2 opening = cv2.erode(equ, erodeKernel)
3 equ = cv2.dilate(opening, dilateKernel)
```

After this processing BRIGHT image will look a lot like the original, except all objects are now expanded by 2 pixels in each direction. DIM image, however, consists of closed and overly exaggerated objects. If, after erosion, a solitary pixel would somehow manage to survive, it would currently be expanded to a 9x9 square due to dilatation. At this point all of these objects would appear similar to rectangles, squares, or, if they were circular before, "blobish squares". This is demonstrated on the figure-2.12.

Figure 2.12: Zoomed in view of the remains of an elongated object after erosion (left) and comparison with that same area after dilatation (right).

This would of course mean that any image that was not completely cleaned of all the stars and objects would likely produce a detection. Comparison of the DIM image from figure-2.11 with the DIM image from figure-2.13 shows how much objects are expanded by the large dilatation kernel. Even if stars would have been removed before-hand near perfectly, any remains would be dilated enough to produce a false detection.

Mechanisms that can adequately make the distinction between such remains of objects and actual detections are needed. Because the object remains are rectangles, squares or generally "blobish squares", if we tried to escribe rectangles to these remains they would all look like an upright rectangle whose sides would be of similar lengths; that is to say a square. Take the elongated object remains on figure-2.12 as an example. On the eroded image (left), if we were to try to escribe rectangles over the remains we would require a lot of small and elongated rectangles. Some of these rectangles would perhaps be only 1 or 2 pixels in height, but 15-17 in width, as is the case with the horizontal remains at the top-center of the left image in figure-2.12.

Once we have dilated these remains, a lot of them have merged back into a single object. The most elongated rectangle that we can fit now is located at the bottom-left on the right image in figure-2.12. However, the width of that rectangle now is approx. 80 pixels while its height is approx. 175px.

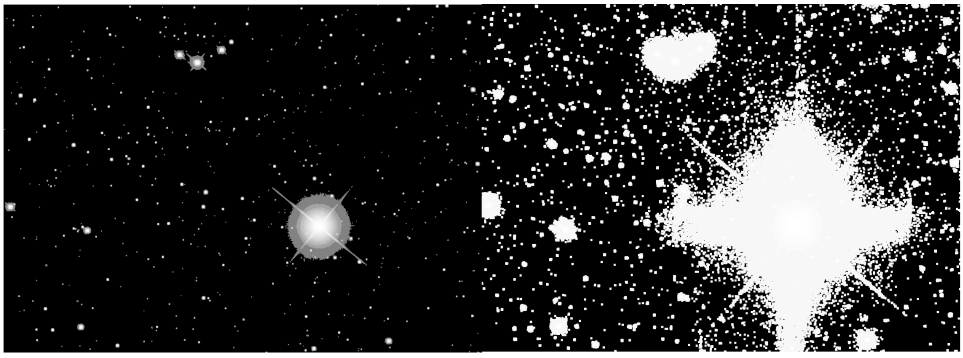


Figure 2.13: Aftermath of erosion and dilatation on DIM (right) and dilatation operator on BRIGHT (left)

Unlike the first case, where width to height ratio was 17, the ratio of height to width this time is just a factor of 2. This holds true if we attempt to produce a minimal area rectangles for any object, especially stars, present on images in figure-2.13. Even in worst case scenarios for large elongated objects, such as galaxies, I have not been able to produce an length to width ratio larger than 4. Since I have decided to take the ratios of longer over smaller values from this point onwards I will refer to them as length for larger value, and width for lower value.

The same logic can be applied to the case of a relatively elongated object. Let us for the sake of simplicity say we have an object exactly 100 pixels long and 10 pixels wide. After erosion kernel operated on that object, its width would be reduced to just 8 pixels while its length would stay approximately the same. Dilatation would expand this trail back to 16-18 pixels wide, depending on the geometry of the trail, while its length would be expanded to 108-110 pixels, length after dilatation can vary 1-2 pixels depending on the geometry at hand. Even in the worst case scenario this is still a length/width ratio of 6 which is significantly larger than length to width ratio of 2.

As I have described in the section-1.1, expected lengths of fully imaged meteors are around 400 pixels long. If we follow the same logic described above and find the length to width ratio for a series of widths, it is easy to see that, for this case, the trails would have to be 100 pixels wide, or more, to come near the ratios that we can attribute to object remains or noise. In reality, most often meteor trails are not wider than 20 pixels. Following the two examples given on the remains of an elongated object (figure-2.12) and an imaginary 100px long elongated object we see that this provides us with an important selection criteria that can distinguish between actual linear features and processing side-effects. Selecting only those objects which can be escribed by a minimal area rectangle with length to width ratio larger than a lwTreshold (set as a high value) can distinguish between noise and actual trails.

This, 3^{rd} step in detecting the existence of linear features, is carried out by the function _fit_minAreaRect, shown in listing-2.23. It finds all edges using Canny edge detection method, then all contours are found to which only the minimal area rectangles that pass the lwTresh criteria are fitted. Figures 2.14, 2.15 and 2.16 display these three steps in the case of process_field_dim only but the same applies to process_field_bright.

Listing 2.23: process_field_dim/bright, step 3

```
def _fit_minAreaRect(img, contoursMode, contoursMethod,
    minAreaRectMinLen,lwTresh, debug):
    detection = False
    box_img = np.zeros(img.shape, dtype=np.uint8)
    canny = cv2.Canny(img, 0, 255)
    contours, hierarchy = cv2.findContours(canny, contoursMode,
         contoursMethod)

    boxes = list()
    for cnt in contours:
        rect = cv2.minAreaRect(cnt)
        if l>minAreaRectMinLen and w>minAreaRectMinLen:
            if (rect[1][0]>rect[1][1]):
                l = rect[1][0]
                w = rect[1][1]
            else:
                w = rect[1][0]
                l = rect[1][1]
            if (l/w>lwTresh):
                detection = True
                box = cv2.cv.BoxPoints(rect)
                box = np.asarray(box, dtype=np.int32)
                cv2.fillPoly(box_img, [box], (255, 255,255))
    return detection, box_img
```

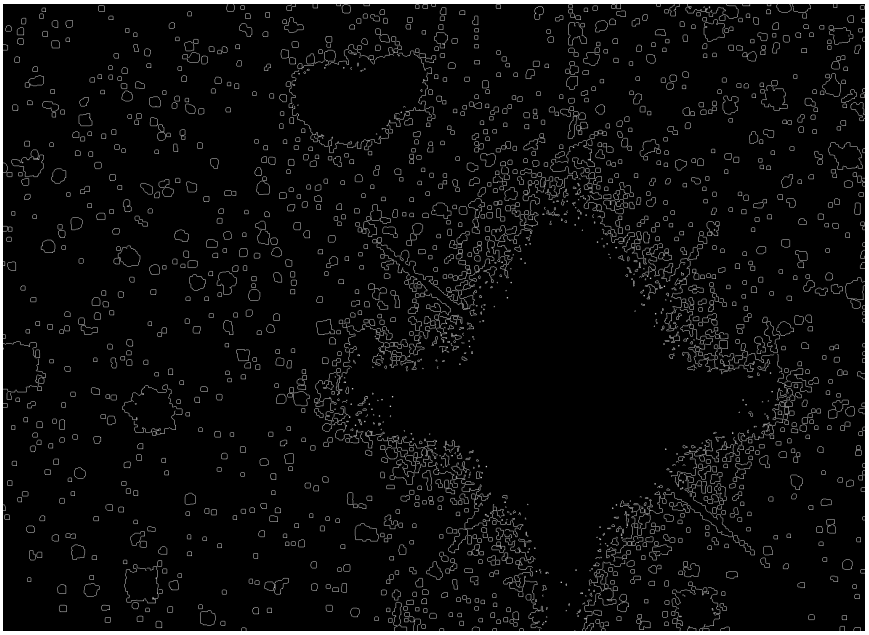
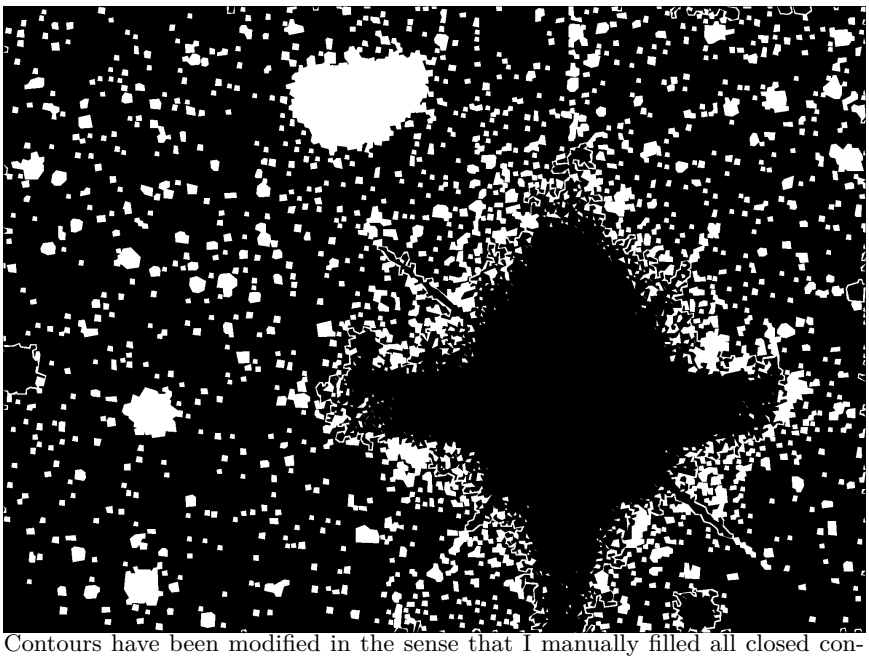


Figure 2.14: Canny edges found on our example image.

Observing figure-2.16 one has to wonder why did OpenCv fit the large rectangle in upper-center area. That rectangle obviously belongs to the large blob, visible in DIM image on figure-2.13, created by merging of multiple stars located in that area. It is obviously a single closed contour as visible on figure-2.15 and the fitted minimal area rectangle should encompass that entire contour. In fact it should never even be plotted because such a rectangle would obviously fail the lwTresh criteria.

Otherwise finding minimal area rectangles has proven as a successful strategy in ignoring large objects that exhibit certain linear features such as the radial lines of Fraunhofer's diffraction visible around the large star (Padgham, 1967). Apart from the bottom right radial line being detected, the other 3 radial lines were successfully ignored. Big success is that the star itself was ignored as well. This would not have been the case if we were to just search for Hough lines on the DIM image in figure-2.13 as was demonstrated in section-2.1.3.6. However, it is still clear that we can not just blindly believe the lines fitted over minimal area rectangles only, as demonstrated by the figure-2.16.



Contours have been modified in the sense that I manually filled all closed contours. This was done because contours are not returned in the form of an image but in a data structure that is also capable of describing their mutual relationships and therefore are not trivial to draw on an image. Only these filled contours will be considered for minimal area rectangle fitting.

Figure 2.15: Contours found among Canny edges.

Figure 2.16: Minimal area rectangles that passed the lwTresh condition.

This brings us to the final step, step 4, in the linear feature detection algorithm. In this step we try to correlate the actual image with the results of minimal area rectangles. One set of lines is fitted over the original image and a secondary set of lines is fitted over the minimal area rectangles. Each set contains nlinsInSet number of lines which are checked for internal consistency within the set, after which both sets are checked for mutual consistency. This is done in the function _check_theta shown in listing-2.24.

Internal consistency is checked by comparing the difference of maximal and minimal θ value of the lines from the same set with the thetaTresh value given by user. Difference of maximal and minimal θ value is observed, instead of their averages, because Hough line detection algorithm tends to place them symmetrically around a possible linear feature. If we had set thetaTresh value to 15° and observed a set of 2 lines, one at -30° and the other at 30°, they would've passed the test even though in reality they are sepparated by 60°. Comparing the difference of maximal and minimal values of θ's in the set makes more sense since in a way that is what determines the "spread" of the set. It is expected that if the linear feature is long and pronounced this "spread" will be a small value.

To check for inter-set consistency both Hough space coordinates (r, θ) are observed. Unlike for θ, averaging r values makes sense because, as I said, Hough line detection algorithm tends to place the lines symmetrically around the linear feature. Since r determines how far "left" or "right" from the linear feature will the line be located, averaging r would produce a more accurate position of the linear feature. Because I will be checking how r coordinates compare between two different sets of lines, one fitted to the processed image and the other to reconstructed minimal area rectangles image, it is better if there were no large deviations caused by a single outlier. If the difference between the two averages is larger than dro parameter, detection is deemed false. Same procedure is applied to compare θ coordinates of the two sets. Averages of the θ's in both sets of lines are subtracted and if that difference is larger than linesetTresh parameter detection is false. It makes sense to average *theta* values of each set individually and then compare the θ values between the sets because we are not interested in how "spread" out each individual set of lines is, but in which direction are the sets of lines facing.

Listing 2.24: process_field_dim/bright, step 4

```
1 def _check_theta(hough1, hough2, navg, dro, thetaTresh,
      linesetTresh, debug):
2     ro1 = np.zeros((navg,1))
3     ro2 = np.zeros((navg,1))
4     theta1 = np.zeros((navg,1))
5     theta2 = np.zeros((navg,1))
6     for i in range(0, navg):
7     try:
```

```
8           ro1[i] = hough1[0][i][0]
9           ro2[i] = hough2[0][i][0]
10          theta1[i] = hough1[0][i][1]
11          theta2[i] = hough2[0][i][1]
12      except:
13          pass
14
15      if abs(np.average(ro1)-np.average(ro2))>dro:
16          return True
17
18      dtheta1=abs(theta1.max()-theta1.min())
19      if dtheta1> thetaTresh:
20          return True
21
22      dtheta2=abs(theta2.max()-theta2.min())
23      if dtheta2> thetaTresh:
24          return True
25
26      dtheta = abs(np.average(theta1-theta2))
27      if np.average(dtheta)> linesetTresh:
28          return True
```

These checks are incredibly useful. In the case of our example image, Hough lines fitted to the actual image, DIM image figure-2.13, will correspond more to the Fraunhofer's diffraction lines emanating from the star itself while in the case of the minimal area rectangles, figure-2.16, they would be fitted to the two longest minimal area rectangles drawn which do not correspond to the radial lines of the star. The two linesets are shown in figure-2.17.

Considering that the two linesets are separated by approximately 70 pixels in the r coordinate this image is declared as a false detection based on the dro criteria. The two lines exhibit very similar θ coordinate confirming the suspicions that even such cases, of different r but almost the same θ coordinates, can occur.

Even though I am expecting that the trails will be very long objects (see section-1.1), it is not a promise that they will always be captured in full. Trails could enter the field under such an angle that they exit the field from either left or right side of the image, or vice-versa entering from the sides they could exit through the top/bottom side of the image. This would limit their length to $\sqrt{x^2+y^2}$ where x is the entry point coordinate and y the exit point coordinate. Such trails could possibly be too short to pass the lwTresh criterion from the start, therefore it is important to set the lwTresh as low as possible. But as noticed, setting the lwTresh too low would start fitting minimal area rectangles to more and more noise and other object remains.

Following this logic, it is a valid observation that comparing line parameters is a double edge sword. What if a short line, perhaps a line cutting through a corner of the image, is present in an image with a large star or

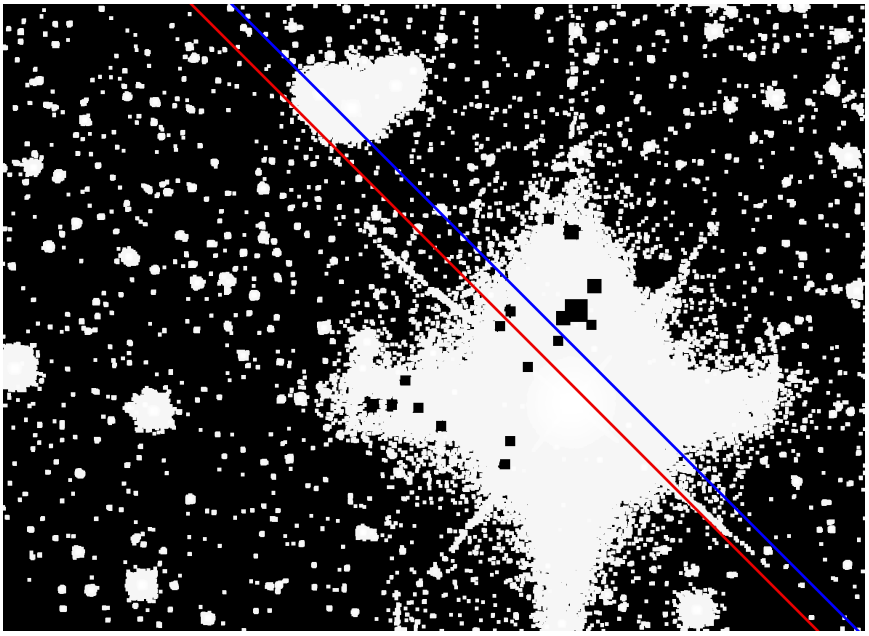


Figure 2.17: Two sets of hough lines fitted to the image. Minimal area rectangle line set is in red while the lineset in blue belongs to lines fitted to DIM image.

galaxy? In that case it is possible that the lines found in the actual image might be fitted to that object instead of the actual line, even though minimal area rectangles have only been fitted to the line itself. Or an opposite case in which a short line is present in an image, without large linear-like objects? In that case it is possible that _fit_minAreaRect would not produce valid rectangles, but the lines fitted in the actual image would still get fitted on the actual line in the image. That line would then be rejected because no valid rectangles could be found at all, or even if they were found they would not correspond to the actual line and the line would be rejected either by θ or r criteria.

Experimentation shows that these checks are essential in isolating images with actual lines from those that have false detection generators such as large galaxies or bright stars. Solution to these issues must be found elsewhere. Best solution would be to perfectly clean the image of all objects we know of that should be there. This would enable us to completely reduce the lwTreshold to nearly the factor of 2.5-3. Consequently this would mean that the minimal area rectangles could be fitted to even the shortest of trails for which the length is comparable to the width, but would still fail in cases where only known objects exist on the image because no rectangles could be found. If we could remove all objects we are not interested in we would not have to worry about lines fitting on elongated objects in the actual image

either. This would still hold true even if I did not remove all known objects from the image perfectly but instead only sufficiently "hollowed" out known objects. We know this because I have shown in section-2.1.3.6 that Hough line detection algorithm is incredibly resiliant to all objects that do not exhibit linear features such as circles and hollowed out discs. So far all the processing has been done on images that have not had any pre-processing of that sort done on them.

2.2.5.2 removestars

The purpose of *removestars* module is to remove as precisely as possible all known objects on the images. It uses photoObj files described in section-1.2.3. As mentioned before, these files contain the full photometric data for every object on the frame. From them I extract pixel coordinates, photometric data and objects shape data in a photoObj_read function and subsequently use this data to blot out these objects on the image itself in a remove_stars function, before image is ever sent to process_field_bright and process_field_dim.

ptotoObj_read function interprets the photoObj file header and extracts the information shown in table-2.5 in the form of 4 lists. Lists row, col, petro90 and psfMag have corresponding indexes, meaning that each object's data is stored in each of the lists under the same index. A more natural way would have been to store these 4 lists in a structure, a class or a dictionary, in such a way that they could not be altered and that objects would be accessed by their unique identifiers. This would remove all ambiguity from the code. Unfortunately in a single image there can be a large number of detected objects, upwards of a thousand. Because I have to keep the performance times small and because looping over a dictionary in python wastes access time on the hash table lookup, I have decided against it.

Function remove_stars "blots" out stars and other objects on the image by drawing black squares over them. Currently the remove_stars function does not use the full extent of data read out by the _read_photoObj function. I have experimented a lot with various data such as exponential fits, object type probabilities, PSF fit parameters, in cases when obcjtype is equal to 3 (galaxy) I tried using de Vaucouleurs fit and elipticity data to perform as close as fit as possible and various other combinations. Even though I have reached a certain level of reliability, at this time I have not yet been able to always reconstruct the original image as well as I would have liked so I have left a lot of this read-out data in the code to avoid re-coding it until such a time I write a satisfactory remove_stars function.

The issue lies in the way SDSS photo pipeline works. Photo pipeline

[8] http://www.sdss3.org/dr10/algorithms/magnitudes.php#mag_psf
[9] http://www.sdss3.org/dr10/algorithms/magnitudes.php#mag_petro
[10] http://www.sdss.org/dr12/algorithms/classify/#photo_class

List name	Data description
row	"y" pixel coordinates. Each entry is a dictionary {u:, g:, r:, i:, z:}
col	"x" pixel coordinates. Each entry is a dictionary {u:, g:, r:, i:, z:}
psfMag[8]	Magnitude as determined by a Gaussian fit of PSF function. Each entry is a dictionary {u:, g:, r:, i:, z:}.
petro90[9]	Radius, in arcsec, that contains 90% of Petrosian flux. Each entry is a dictionary {u:, g:, r:, i:, z:}.
objctype[10]	Type classification. Integer.
types[10]	Type classification per filter band. Each entry is a dictionary {u:, g:, r:, i:, z:}.
nObserve	Number of times this position was observed. Integer.
nDetect	Number of times object was detected in the observed position. Integer.

Table 2.5: Data model of usefull photoObj data tables.

detects all pixels whose counts are above the background sky noise. All such pixels that are in contact one with another are organized into objects found in fpAtlas files. Deblender, if possible, separates these clumps into objects. Separated parts are, when possible, constrained by parent-child relationship. This makes it possible to detect, for example, stars within sufficiently large galaxies.

However this is bad news for me. This implies that all trails will also be registered as objects in the pipeline itself just because their pixel counts are above the background sky noise. Furthermore because trails are not uniformly bright along their length, it implies that trails will most likely be deblended as a combination of a series of objects, some perhaps marked as galaxies, some as large stars. Immediately, this, rules out sorting based on object type. When reconstructing a frame using nothing but photoObj file data this problem becomes obvious. Such reconstructed frame is displayed on the figure-2.19. For comparison, figure-2.18 displays the original frame. Original frame underwent only histogram equalization; therefore a lot of low-brightness objects are not visible. Reconstructed frame used all data available in the photoObj file without any filtering.

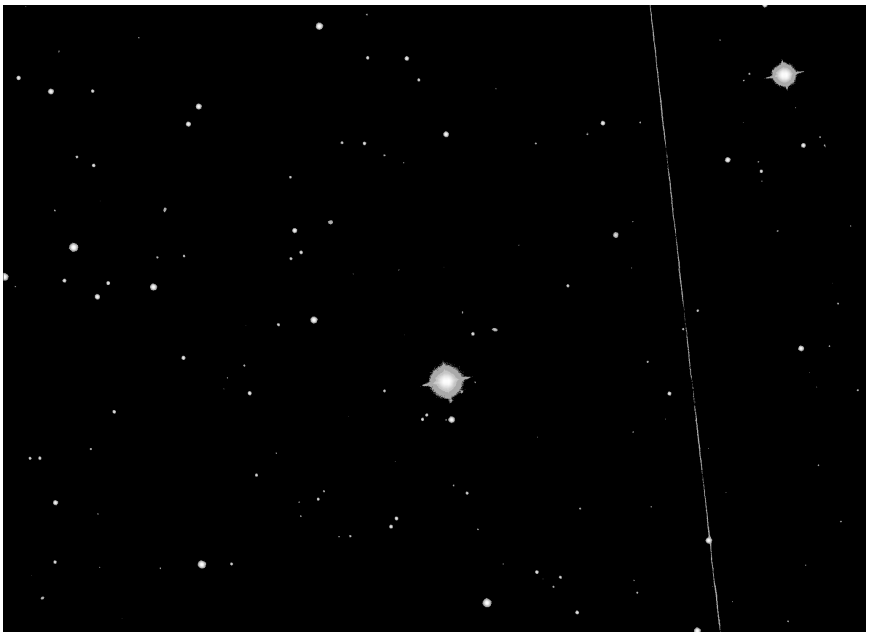


Figure 2.18: Original frame-i-002888-1-0139 data.

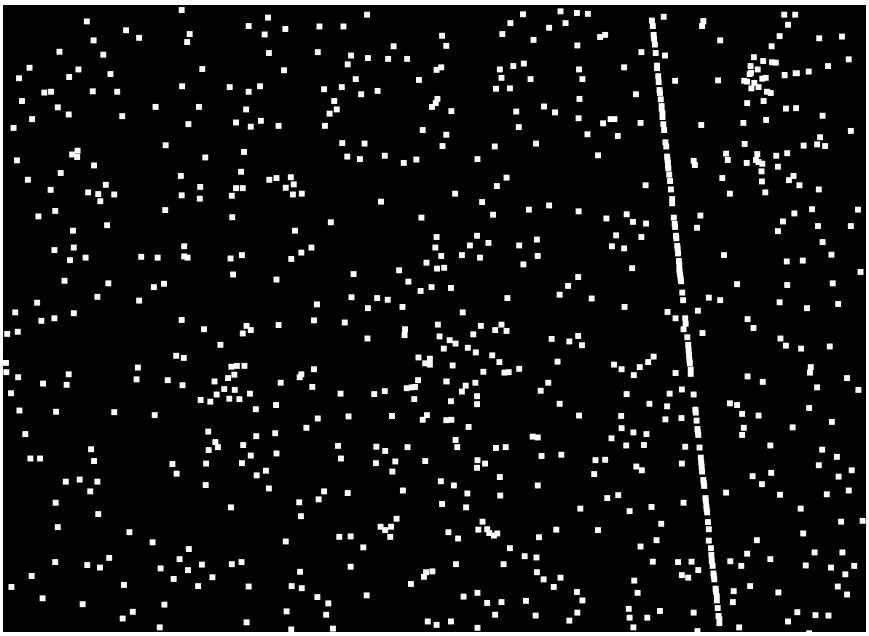


Figure 2.19: Reconstructed frame using photoObj-002888-1-0139 data.

As noticeable on the reconstructed frame in figure-2.19 the trail is visible as well. This means I can not just use all the data I read from photoObj files but have to sort through them for objects I can confidently confirm are actually objects I want removed and not a product of photo pipeline. Otherwise I would be deleting my own trails.

Before a square is actually drawn over the object, size of the square is, if possible, determined with petro90 radius. Petrosian radius is given in arcseconds and has to be converted into pixel value based on the pixscale factor provided by the user. If the calculated radius is less than zero a default square size is used. If the calculated side length is longer than maxxy parameter, a default square size is used. Default square size is determined by the user-set dxy parameter. Reconstruction of the frame-i-002888-1-0139 including variable square sizes is shown on figure-2.20. As we can see, this reconstruction bares more resemblance to the original image than figure-2.19 although we are still masking our trail.

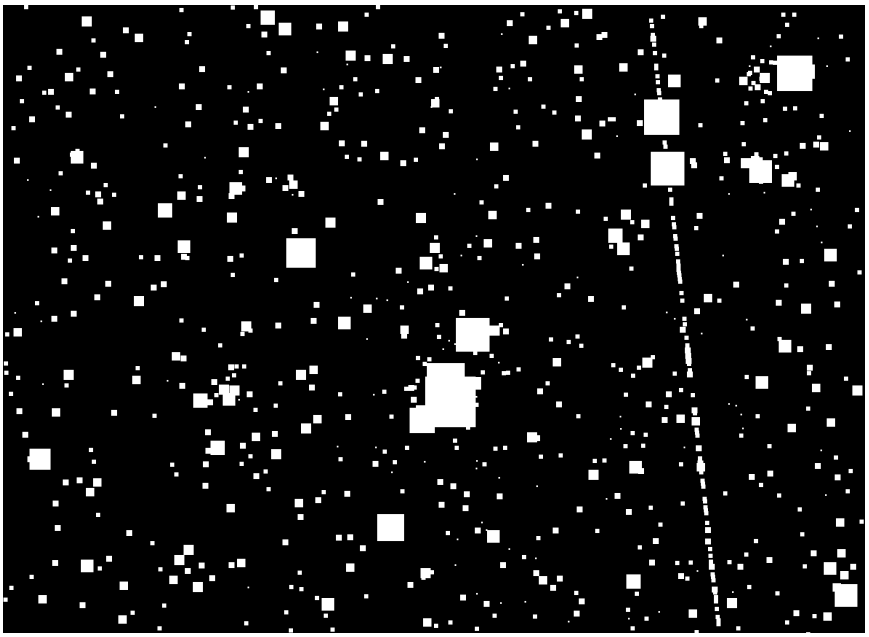


Figure 2.20: Reconstructed frame-i-002888-1-0139 with variable square sizes.

In order to stop masking the trails, first I select only those objects that are brighter than a certain apparent magnitude. These magnitude "caps" are controlled by the filter_caps parameter. According to Stoughton et al. (2002) SDSS has 95% completeness limits for stars at magnitudes 22.0, 22.2, 22.2, 21.3, and 20.5 in five filters [ugriz] respectively. These magnitude limits do not actually affect if SDSS can or can not record magnitudes deeper than that, authors just remark that at those magnitudes SDSS has successfully recorded 95% of all existing objects. It is worth to remark that errors grow

past these magnitudes and parameters measured for these objects grow more unstable. In any case, sizes of stars dimmer than these limiting magnitudes are small enough that they will not interfere with Hough line detection algorithm thus, I have decided to take magnitude 22 as the bottom limiting magnitudes for all filters. In the case of the z filter, however, I might decide to reduce the limiting magnitude once I get the necessary images and start testing LFDS on that filter as well. Currently I have only been able to determine these magnitudes work well on r and i filter, whose data I have. Once this condition is applied on the figure-2.20 we get the following mask shown on figure-2.21.

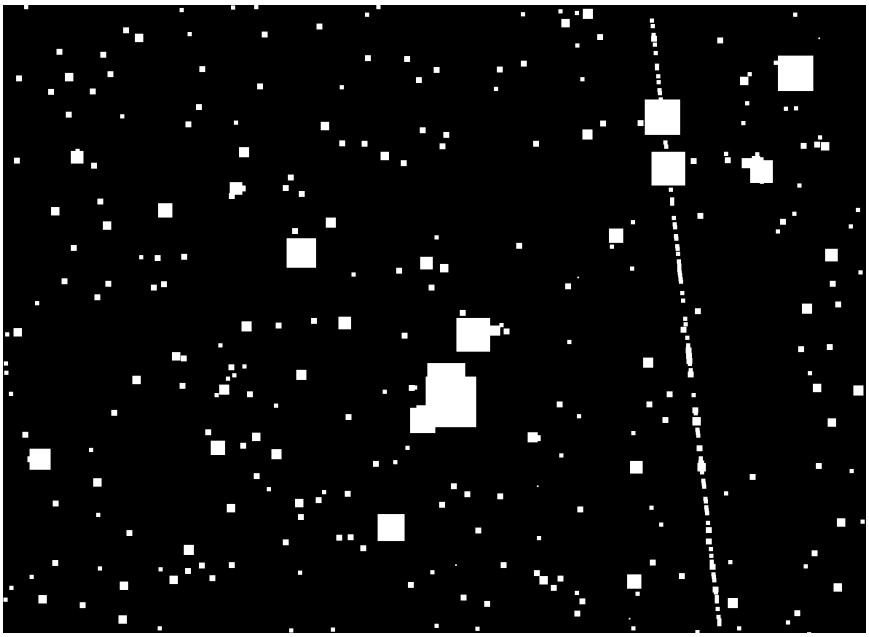


Figure 2.21: Star mask with variable square sizes and filter caps.

Still we see that this did not filter out the trail objects in photoObj file. This is to be expected because trail in the example image is not exactly very dim. We did get rid of a lot of small squares on the mask that could not be attributed to any object visible on figure-2.18. This is a positive step, blotting out such large parts of the image risks interfering destructively with the trail.

If there is an object in the image, for example a star, we should expect that we are able to see it in other filters as well. Each stationary object has one image in each filter, taken at 71.7 second intervals (see section-1.2.1). Any transient phoenomena, not moving exactly in parallel to the apparent motion of the sky, would enter and exit only in that single filter. If there is a fast moving transient phoenomena, such as a meteor or a satellite, it would never even be recorded on any other filters. If the star is a red giant it

should be the brightest in the r and i filters but it should still be measurable in other filters as well, albeit dimmer. Therefore we can expect that objects moving with the same angular speed as the sky will be recorded in all filters, while objects not moving with the same angular speed as the sky would have relatively high magnitudes recorded only in a single filter and extremely dim magnitudes, corresponding to a null-detection, in others.

It is not clear how much that difference amounts to or how much dimmer can we expect a star of certain class to appear in other filters. I have not found a function, method or a rule of thumb governing this, so this will most certainly have to be determined by trial and error approach. Still if we subtracted each filter magnitude from others and compare if those differences are larger than a user set threshold, with some experimenting, we should be able to determine which objects were actually measured in all filters and which were not. We could even take it a step further and say that if differences in measured magnitudes of all filters are larger than set threshold, in more than a user set number of filters[8], then the object is not a stationary one. Parameter maxmagdiff controls what is the maximal allowed difference of magnitudes, while parameter magcount controls in how many filters is the maxmagdiff allowed.

For example, if we set maxmagdiff to 5 and magcount to 3 the following objects would not pass the test: (2,8,8,8,8). Obviously, this object is very bright in just one of the filters and not in others, therefore is ruled out. Object with magnitudes (2,2,8,8,8) would fail as well. This object is bright in two filters, but not in others. Difference of magnitude of the first filter and all other filters gives us (0,6,6,6). The differences in magnitudes are bigger than maxmagdif in 3 different filters. Per condition, differences in magnitude are bigger than maxmagdiff in at least magcount filters, therefore this is not considered as an actual object.

The following objects would pass this test for the same maxmagdiff and magcount parameters. Object with magnitudes (5,6,7,8,9). First magnitude produces the following differences (1,2,3,4). Because all differences are smaller than maxmagdiff this is considered an actual object. This object is brightest in the first filter, i.e. filter i, and then gets progressively dimmer through the r, g, u and z filters. This is how I would imagine a yellow star behaves. Object with magnitudes (3,3,7,8,9) would pass as well. Magnitude differences for this object are (0,4,5,6) therefore only one magnitude difference is larger than maxmagdiff. This condition has to be broken in at least magcount filters for it to fail. If we take the first value to be the i filter and the second r filter then this would be how I imagine a very red star behaves.

As noticeable, this condition can be thought of as sorting the magnitudes from lowest to highest value and then counting the number of occurrences when the difference of first magnitude with all other magnitudes is larger

[8]The actual used condition states: in more than or equal to a user set number of filters.

than maxmagdif. If this occurred more than or equal to magcount times object is not considered “real”. Image displayed on figure-2.22 shows the frame reconstruction under this condition.

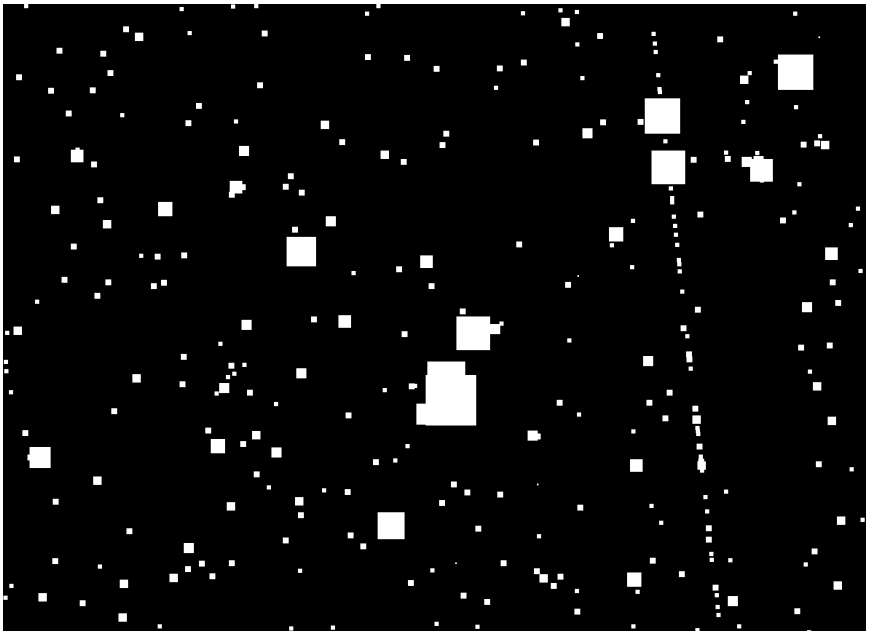


Figure 2.22: Star mask with variable square sizes, filter caps and magnitude difference condition.

Former condition managed to get rid of at least half of the objects from our trail. The last condition we can apply is the simplest one. It states that the number of times this area was observed must match the number of times this object has been detected. There is a single important downfall to this criterion; to the best of my knowledge SDSS has not imaged all areas twice. If an area has not been imaged at least twice, this criterion is always true by default. On figure-2.23 I have shown the reconstructed mask for the frame-i-002888-1-0139 including the latest criterion. As noticeable the trail is not visible on the mask this time.

If we subtract the final mask from the starting image (figure-2.18) as shown on figure-2.24, we can see that it does a fairly good job at removing stars from the image and does not excessively interfere with the trail.

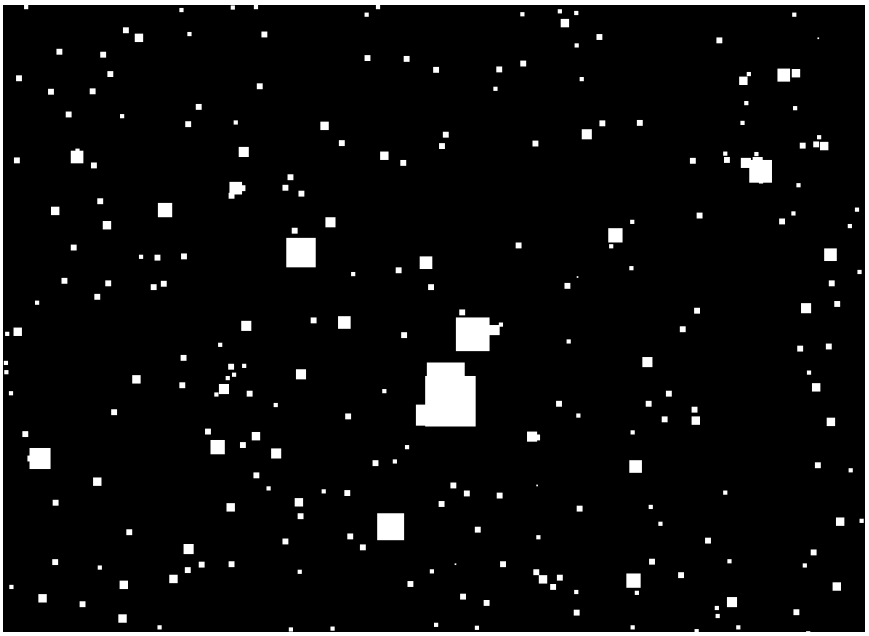


Figure 2.23: Final mask used for removal of known objects from the image.

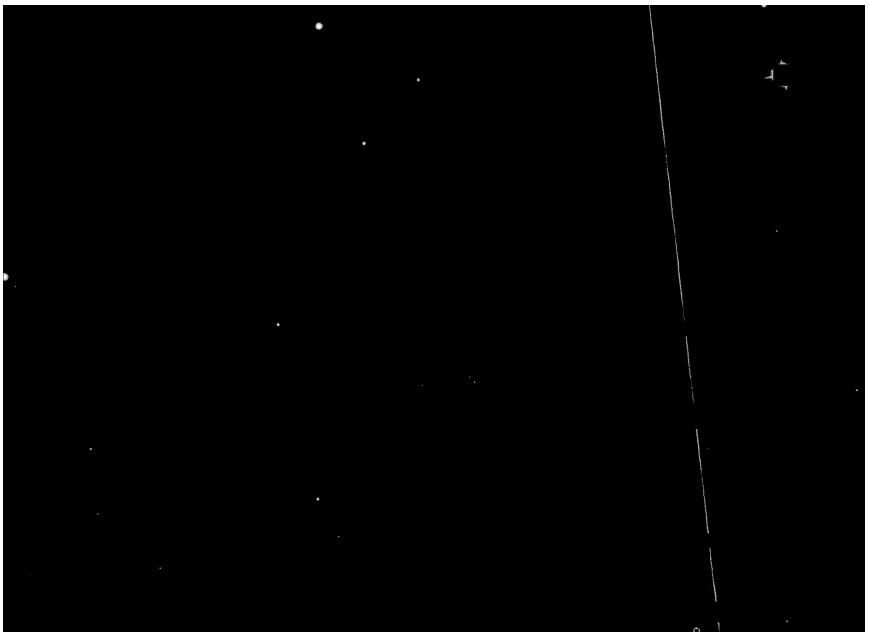


Figure 2.24: Final mask subtracted from the original image.

Unfortunately this is not always the case. For example using the frame-i-002888-1-0017 (BRIGHT image on figure-2.11) we will not get as good as results as we did with frame-i-002888-1-0139. photoObj-002888-1-0017 file contains only 96 entries and they are mostly concentrated around the bright star on the image. Figure-2.25 displays a mask created by the same methods described so far. On figure-2.26 result of mask subtraction from the frame-i-002888-1-0017 is shown. As we can see the star removal works excellent. It has removed all small to medium sized stars and has "hollowed" out the large star, just as requested in the conclusion of section-2.2.5.1. Unfortunately it is strictly contained to the vicinity of the large star producing less than impressive results.

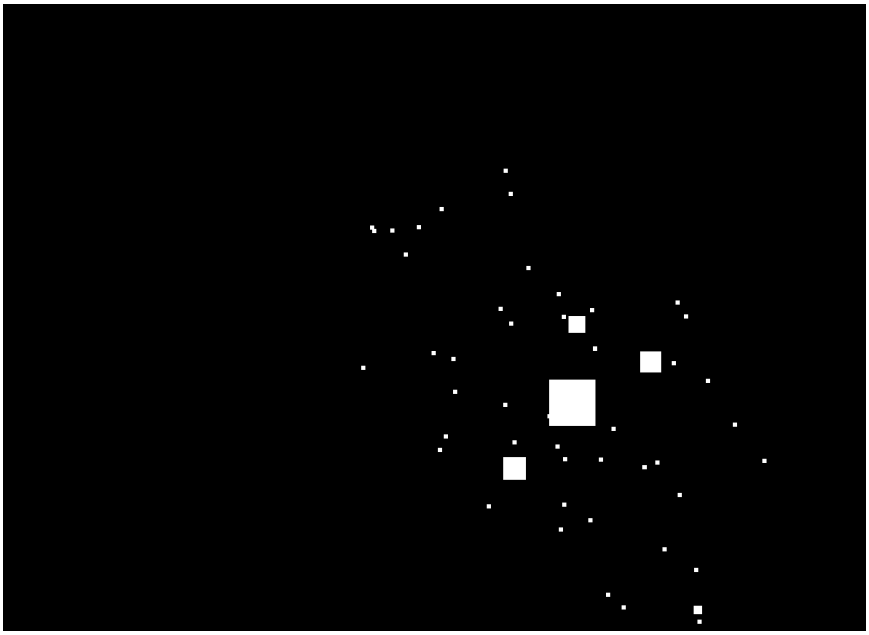


Figure 2.25: Final mask for frame-i-002888-1-0017

These issues are solvable. Both photoObj and frames files are chock-full of data that could potentially shine a light on this situation. Of special interest are the object flags[9] entries in the photoObj files that provide additional information about the object such as: if there were issues during deblending, if the object has a stable Petrosian/de Vaucouleurs/exponential fit, if they are saturated, binned or near the edge etc. All this information can be used to determine what kind of object is it and subsequently to use the best parameters possible to remove them (Petrosian radius for bright objects, de Vaucouleurs profiles for galaxies, PSF fits for dimmer stars etc.). The Main issue here is the fact that a lot of these parameters require a

[9]Best described on Robert Lupton's webpage http://www.astro.princeton.edu/~rhl/flags.html

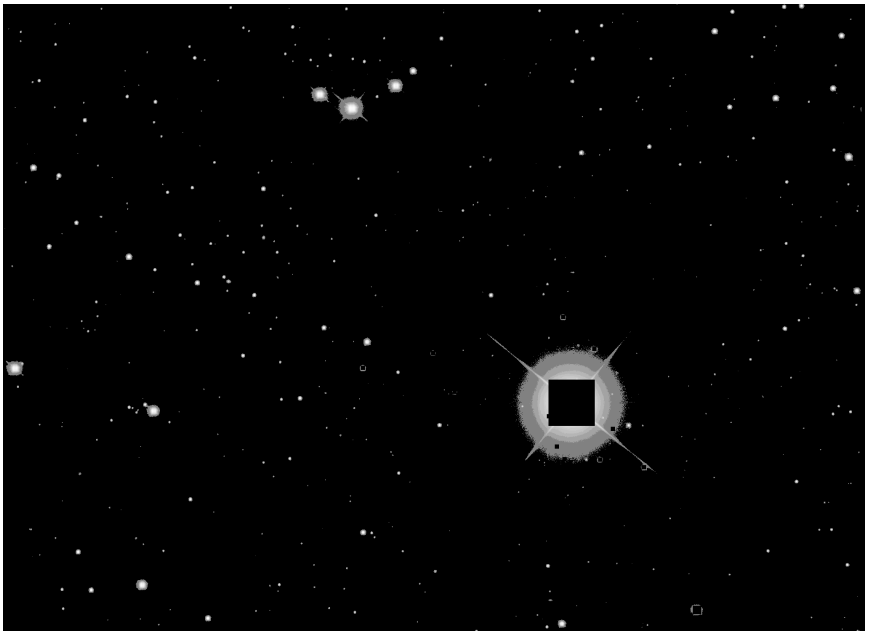


Figure 2.26: Image recovered by subtracting appropriate mask from the original frame-i-002888-1-0017.

level of applied/theoretical knowledge and this slows down the progress in developing new algorithms.

This is not the only issue with *removestars* module. Considering that I have in several occasions stressed the importance of writing a generic package that could be run on different databases, having such a key module as *removestars* that is purely SDSS oriented is a bad decision. *removestars* module should be expanded to be able to interpret different catalogue information, such as UCAC2, USNO A and B, NOMAD, GSC etc... If LFDS is to ever be run on any other database alongside SDSS, a fast and reliable way to find all objects within a selected field is still needed.

2.3 LFDS recap and processing examples

To recap, in short, the entire processing procedure I will use three different images. One without a trail and two with trails. For the example without trails we will use frame-i-00787-1-0045 which presents a similar problem as the example frame used in section-2.2.5.1 (see figure-2.10). Steps of both process_field_bright and process_field_dim functions are displayed on figure-2.27. For examples of detection of linear features I use frame-i-002888-1-139 for bright detection on the left side of figure-2.28 and frame-u-005973-3-0128 as a borderline dim detection on the right side of figure-2.28. Only the steps of the function that detected the trail is shown this time.

In the case LFDS was run on the Fermi cluster user would have to create a job as described in the section-2.2.2. In the case LFDS was run locally user would have to run the `start.sh` script located in the home folder of LFDS package. To run the frame through the detection process in both cases the same command would be used, the only difference would be that the processing on Fermi would be run through the TORQUE and MAUI scheduler while locally user can run these commands through the python-idle opened by the startup shell script. Command used in processing all example images is shown in listing-2.25.

Listing 2.25: Command used to run LFDS

```
1 d = dt.DetectTrails(run=7787, camcol=1, filter='i', field=45)
2 d.params_bright["debug"]=True
3 d.params_dim["debug"] = True
4 d.process()
```

detecttrails module first searches for the frame in the path described by `PHOTO_OBJ` environmental variable. This variable is controlled by setting the `BOSS` variable in thestartup script, or in the generic template in *createjobs* module. In that path LFDS expects to find a frame file compressed by the bz2 format. Frame will be extracted to the path set by `FITS_DUMP` environmental variable. FITSIO will open the fits file and load image header and data header. A partial result string is formed from the data header while the image header data gets set to *removestars* module for processing. *removestars* module does, as the name implies, star removal from the images by drawing black squares over filtered set of all objects in the image detected by photo pipeline. These objects are extracted from photoObj file. Additionally to stars, *removestars* module removes galaxies and other knows objects as well.

The result of this processing is passed in the detect_bright function. detect_bright first sets all pixels less than zero to 0. It then equalizes histogram of such image. Results of the equalization are then "buffed" by dilating the image. Minimal area rectangles are fitted on the dilated image. If not one rectangle that match the lwTresh and minAreaRectMinLen criteria is found, function terminates as a False detection. If any rectangles that pass the criteria are found, detect_bright continues. detect_bright function then finds lines on the equalized dilated image and on the image constructed from the fitted minimal area rectangles.

Lines fitted have their r's and θ's compared both between the two sets of lines fitted on equalized dilated image and minAreRect image and inside the set itself. This comparison is done over the nlinesInSet strongest lines detected. Thresholds that determine if the comparison is valid or not are controlled by following parameters: dro, thetaTresh and linesetTresh. If the lines pass this check frame is marked as a True detection and the function terminates. If the lines do not pass this check, detection continues.

The *detecttrails* module then calls the detect_dim function of *processfield* module. This function takes the original image returned by *removestars* module and sets all pixels with values less than minFlux, a user-set parameter, value to zero. Pixels that remain after have their values increased by the addFlux parameter. Histogram equalization is then done.

This equalized image is then eroded to destroy all small single solitary pixels and objects that have now become visible due to the artificial brightness increase. To restore and boost the remaining objects on the image dilatation with a large dilatation kernel is used. Following that is the same process done in detect_bright. Minimal area rectangles are fitted. If none are found, function terminates. If rectangles are found, lines are fitted over them. Lines are fitted over the equalized, eroded and then dilated image as well. If these lines pass the tests function returns a True detection and parameters. If the lines do not pass the tests, function terminates as a False detection.

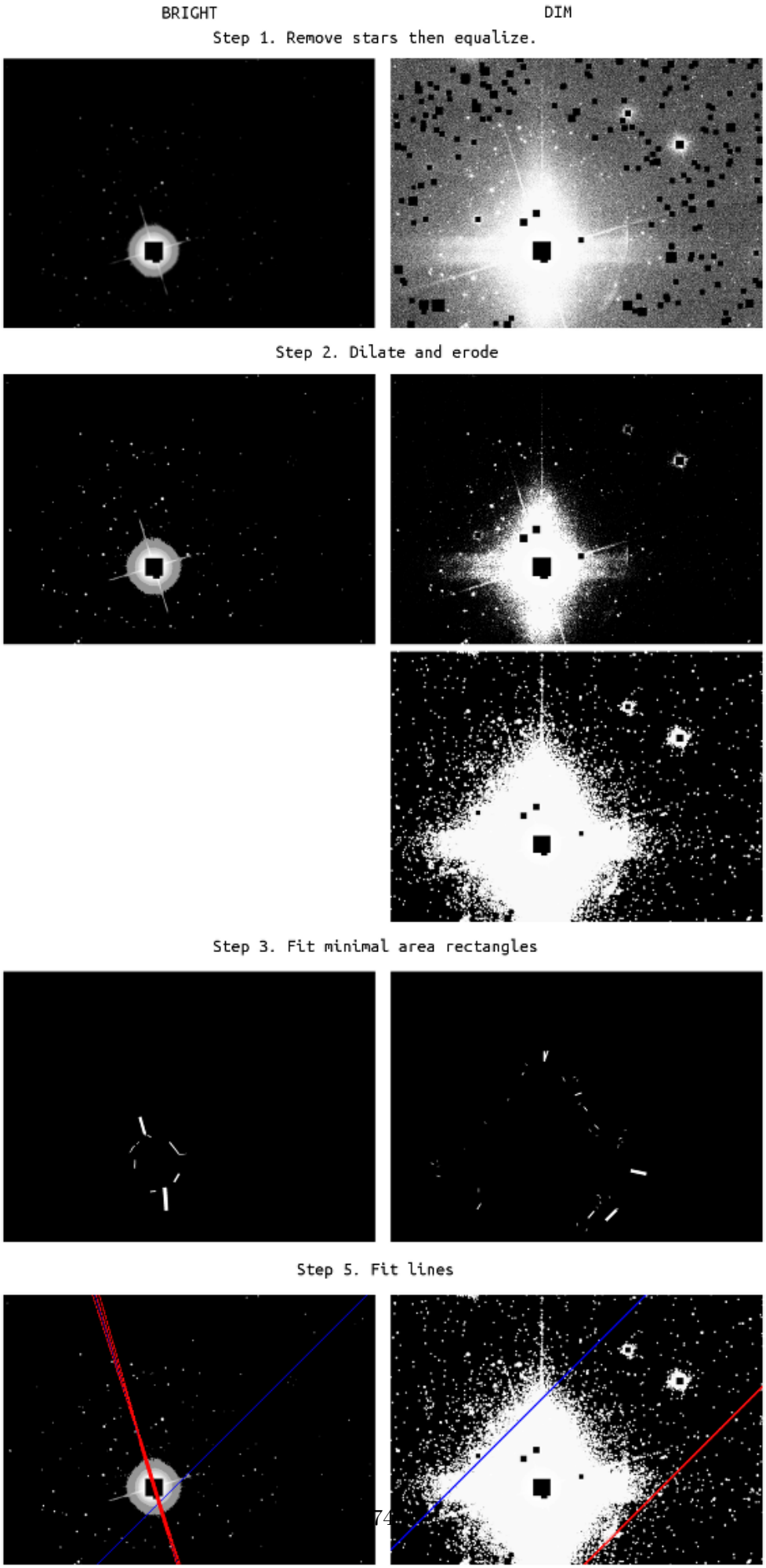


Figure 2.27: frame-i-00787-1-0045 processing steps displayed visually for the detect_bright (left) and detect_dim functions (right)

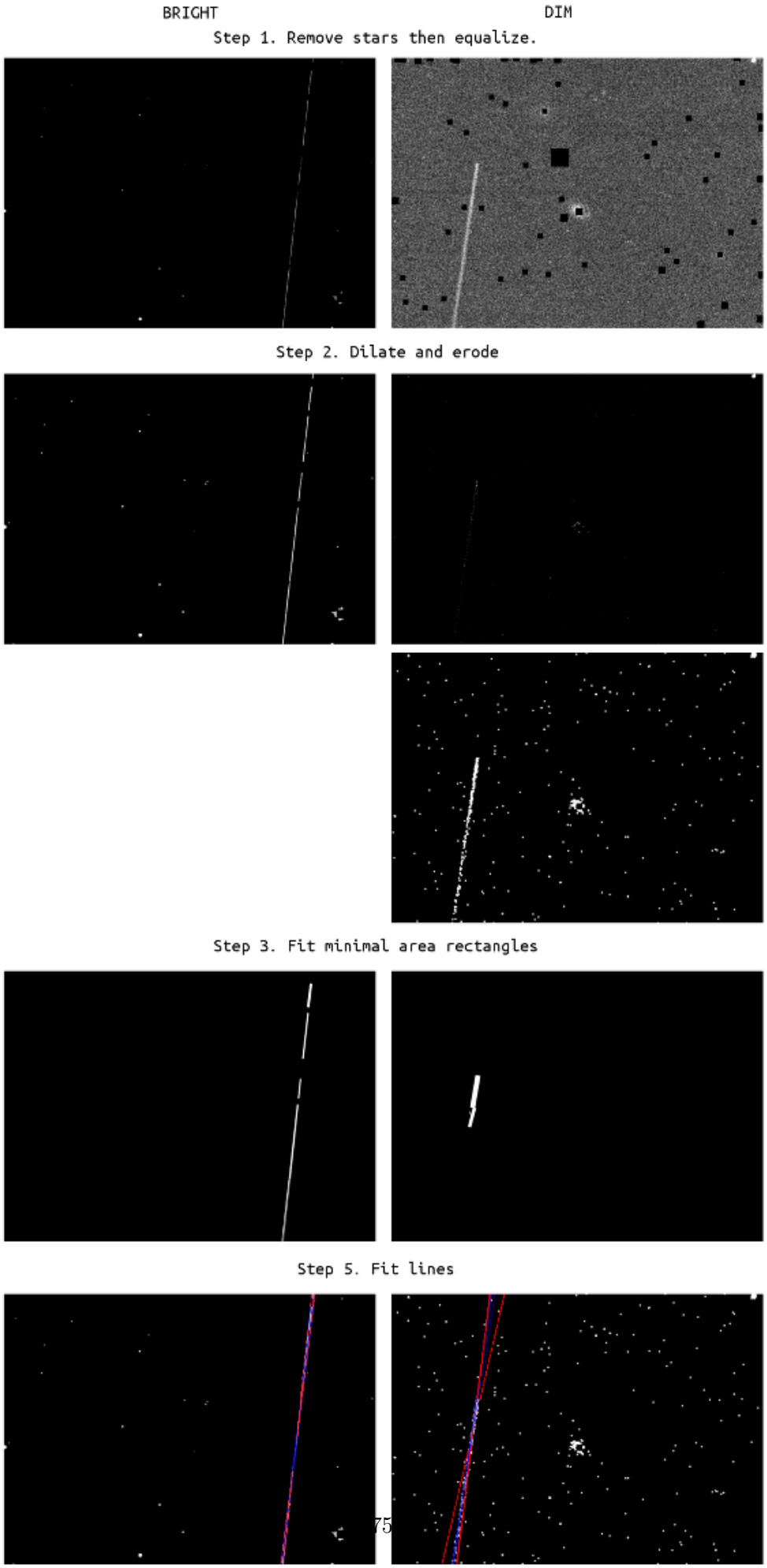


Figure 2.28: Processing steps that produced output for frame-i-002888-1-139(left) and frame-u-005973-3-0128 (right).

Chapter 3

LFDS benchmarking

So far LFDS was run primarily on the images in i filter. All the parameters and steps described in chapter-2 were determined on that subset of data as well. Therefore, it is important to acknowledge that it is likely that the current version of LFDS is not necessarily best-suited for detection of linear features on all filters. Changing the parameters can have influence on the detection rates as well as performance times of LFDS. In this chapter I will present the performance of current version of LFDS software with the parameters set at the values described in table-2.4 and compare it with previous attempt of batch processing.

The first results we recovered were presented at LSST@Europe conference (Cikota et al., 2013). Dataset then analysed consisted of 1 876 092 frames primarily in i and r filter. There were a total of 420 000 potential detections recovered which were hand-checked for false positive detection. From the potential detections, after sorting, we recovered 15 063 actual trails, 7 768 of those belonged to the i filter. Approximate execution time per frame was 30 seconds. Because the first program used the Random sample consensus (RANSAC) algorithm (Fischler and Bolles, 1981) for detection of linear features it was very susceptible to noise. Number of false positives was overwhelming and required a team effort over a span of couple of weeks to recheck all the images by hand. The nature of the detection algorithm was such that we can confidently claim we detected a high percentage (>95%) of existing lines on the processed images. Data recovered by this processing is used as a benchmark for analysis of detection confidence levels of LFDS.

3.1 Execution time

I have measured execution times of remove_stars, detect_bright and detect_dim functions. Tests were executed on 8^{th} of September 2015 on Fermi. Only data for run 94, camcol 1, i filter were considered. Out of 535 frames existant in run 94, none failed or produced errors of other nature during test-

ing. Histogram on figure-3.1 shows the total execution times for all three functions. Only 7 frames had remove_stars execution time over 0.5 seconds while the same is true for only 5 frames of detect_dim function. Histogram on figure-3.2 shows execution times of functions with upper x-axis limit on 0.5 seconds.

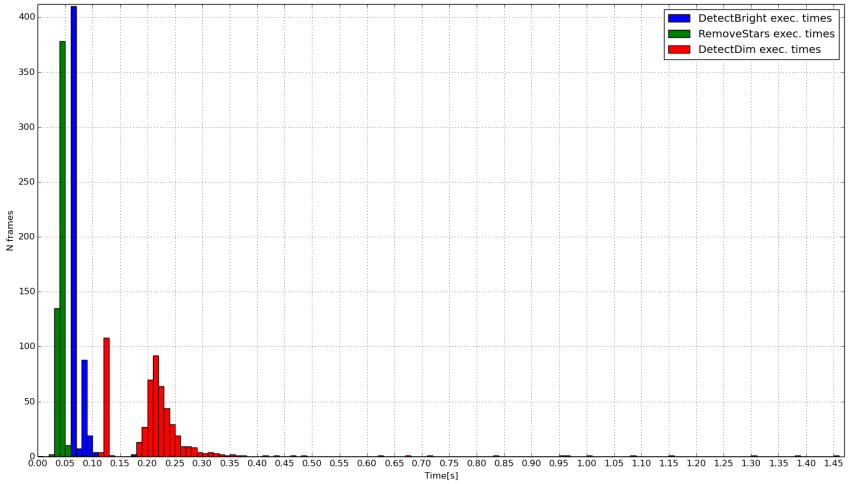


Figure 3.1: Execution times of functions remove_stars, detect_bright and detect_dim.

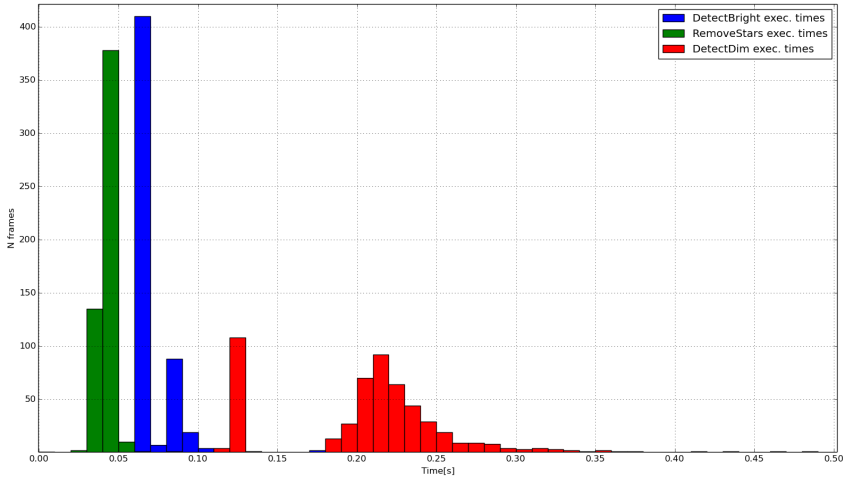


Figure 3.2: Execution times of functions in more details.

What is noticeable from figure-3.2 is that detect_bright and detect_dim functions display two peaks. detect_bright has one peak at execution times of 0.06 to 0.07 seconds and the second peak at 0.08 to 0.09 seconds. First peak can be attributed to an early return of the function, presumably be-

cause no minimal area rectangles could be found. detect_dim displays the same behaviour with first peak at the 0.12 to 0.13 seconds time range which happens for frames for which no minimal area rectangles could be found. The second detect_dim peak is more spread out and ranges from 0.18 to 0.34 seconds. This is due to the fact that not all images are of the same quality. Because the _fit_minAreaRect and Hough line detection functions run several operations per pixel, noisier images will consume more time. remove_stars function is mostly executed in the 0.02 to 0.06 seconds, however there are several occurrences of this function that take up to 1.3 seconds to execute. This is, most likely, because photoObj files for those frames contain a large amount of recorded object through which I loop over in Python. As explained in section-2.2.5.2 looping in python is very costly. In table-3.1 I present average and mean times of execution times per function.

Function name	Average exec. time [s]	Mean exec. time [s]
remove_stars	0.0573	0.0573
detect_bright	0.0687	0.0687
detect_dim	0.2126	0.2126

Table 3.1: Contours mode options.

A separate test was done to measure total execution time of the program. Alongside the execution times of remove_stars, detect_bright and detect_dim functions this test help measure the cost of unpacking time from bz2 to fits format, results and errors saving times, function calls and time spent on algorithm logic. Total execution time average was 0.8647 seconds and coincided with its mean indicating that there are no extreme execution time outliers. Execution times per number of frames is shown on figure-3.3.

Immediately noticeable from the figure-3.3 is that the total execution times are concentrated mostly around 0.9 seconds. This 0.6 seconds larger than the sum of the execution times remove_stars, detect_bright and detect_dim functions. Most of this time difference can be attributed to the long unpacking time from bz2 archive to the local fermi-node. What is visible as well are that the histogram has 3 peaks, one ranging from 0.65 to 0.69 seconds, one ranging from 0.73-0.8 seconds and the final one ranging from 0.82 to 0.97 seconds. Presumably, the lowest execution time peak can be attributed to situation where both searches for minimal area rectangles failed. Second peak can be attributed to function detect_bright succeeding. Third peak belongs to the situation where both detect_bright and detect_dim functions acted on the image.

Current version of LFDS outperforms both the Cikota et al. (2013) and Bekteševi ć (2013) in the time execution benchmarking by a factor of 30. Current execution times are dominated by the bz2 decompression step, therefore still leaving the opportunity to increase the performance by factor of 2 if

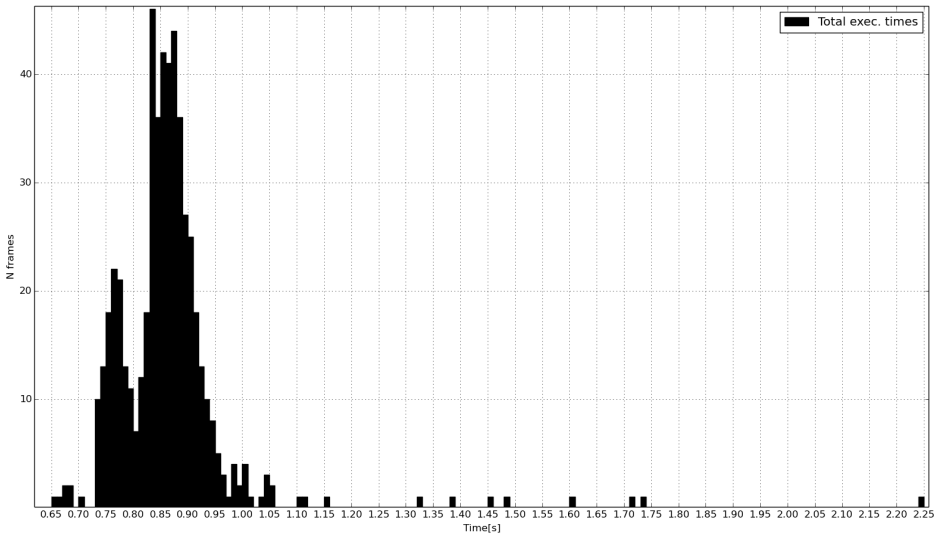


Figure 3.3: Total execution time per frame.

more HDD space is allocated. If a faster permanent memory was used, i.e. SSD disk, file access times would be reduced even further. Tests performed on 240GB Kingston SSD (SHFS37A240G) show, on a 1000 samples, 10MB each, the average read rate to be 458.1 MB/s, with average access time being 0.15 ms. Compared to the used HDD, file access times would be reduced by a factor of 4. However, the best approach would be pipeline design. A clever file management script that spools the data before and during execution itself could increase the performance drastically. However such a script would be extremely hard to code. Before the job is executed it is run through a scheduler and TORQUE which means jobs are not always executed in the same order they are received. This mandates a continuous communication with the scheduler as well as with the program execution to predict which data will be needed next. Issues related to preventing race conditions and deletion of files are also present. Considering the level of my knowledge I have deemed that spooling approach is too complicated and error prone to be reliable.

Processing time of an entire filter, currently, is a little over 4 days. Assuming a stable processing times for all filters we can extrapolate that for the entire set of nearly 5.6 million images we would need approximately 20 days of processing which is tolerable. In a pipelined processing approach the same amount of data could easily be processed in half the current necessary time. Current LFDS performance enables large data analysis to be done by just one person. Running filter per filter, and then analysing each output with *imagechecker* (section-2.2.4.1) during the execution of the next filter would enable the retrieval of final results in optimal time.

3.2 Detection rates

Entire i filter has been processed with LFDS and results retrieved were compared with the benchmark results. Out of 1 122 264 images LFDS returned 17 058 images as possible detections. Compared to benchmark results, number of preliminary results returned by LFDS is a significant improvement as well. There are 25 times fewer frames that need to be checked by hand in LFDS than there were in benchmark results. Total rejection rate, ratio of the number of processed frames and the number of returned frames, is 98.5%. Assuming that total rejection rate remains approximately constant through all filters we can expect that for the full dataset of 5 611 320 images, returned number of possible detections amounts to approximately 85 thousand images. From experience I know that this is manageable even for a single person within a span of 2 to 3 days. Considering the optimal approach where user would analyse smaller batches of preliminary results during the processing of other filters, this after-processing that has to be done manually would present a negligible time delay between preliminary and final results. Processing smaller batches of preliminary results in this way would not present a time consuming or overly stressful job either. However smaller number of frames in the preliminary results does not necessarily mean a good detection rate. Cross-referencing frame names in LFDS preliminary results and benchmark results reveals that, out of the total 17 thousand images returned, only 5 391 actually contain trails. Detection rate, ratio of the number of frames actually containing trails with the number of frames containing trails detected by LFDS, is 69.4%. For certain parameter settings I can get a higher detection rate, but at a cost of a drastical increase in number of false positive detections. Maximally achieved detection rate was 75% at a cost of 35 thousand potential trails detected on the i filter. Including other filters and post-processing time requirement estimates I have decided that this is not a beneficial trade-off.

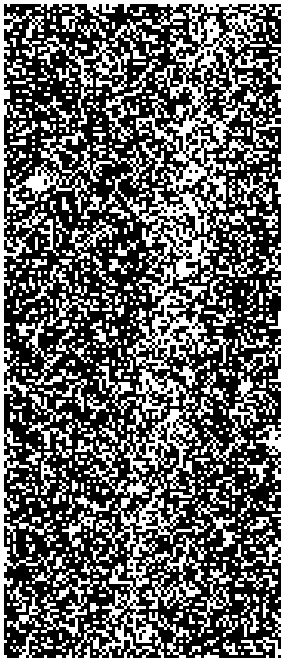


Figure 3.4: Zoomed section of transparent trail.

Main issues that prevent the detection of more trails lie in the fact that a lot of trails are semi-transparent. Once recorded on the image a human eye can see them just fine as object whose intensity is several times over that of the background. Unfortunately, the pixels constituting that object, even if more luminous, are not more "dense" in comparison to the surrounding noise. If zoomed in sufficiently, to the single-pixel

level, such trails can be indistinguishable from the surrounding noise. One such trail is shown in figure-3.4. Since current algorithm relies heavily on finding objects by using the relationships between the positions of pixels of an object, to avoid a lot of false detections it is prudent to perform noise removal beforehand[1]. Noise removal is done by eroding the image before dilating it. Unfortunately because erosion does not consider the intensities of surrounding pixels, merely checks if there is a pixel of lesser intensity in its surrounding (section-2.1.3.1), it often deletes such trails completely, or at least partially. Since reducing the erosion kernel any further would not make any impact on the image, and we have to reduce the noise for detection and performance benefits, this step cannot be cut.

The only other way to boost the detection rate is to loosen the detection parameters and allow for a lot more false positives. Then additionally provide a more processing intensive and time consuming post-processing step that would be centered at the assumption that all images have a line and that we know where it is on the image. The general idea would be to take the preliminary results, cut out the section of the image where the trail is supposed to be and run a special detection algorithm processing only that part of the image. Benefits of this approach are twofold. Firstly, because we have only cut a selected part of the image we have automatically reduced the number of pixels we have to perform operations on and thus allowed ourselves to be less drastic when it comes to noise removal. Secondly, we can allow ourselves to be less constrained by execution time demands due to the decreased data set we need to process. In that situation LFDS would provide a fast decision making algorithm if such additional step would be done on the image or not.

[1] this step also helps improve the performance of the program by removing active pixels from the image

Chapter 4

Conclusion

I have shown that meteor trails, when imaged, are expected to be dominant features in the SDSS images (section-1.1). This makes meteors well suited cadidates for fast, scalable, automatic detection. Still, issues related to analyzing tens of terabytes of data (section-1.3) proved to be quite restricting in practice.

The largest practical obstacle to overcome is the file access latency and bandwidth. For a modern HDD with average read rate of a 79MB/s extracting the file occupies 0.8 seconds per file. The average execution time of functions is 0.06 seconds for *removestars*, 0.07 seconds for detect_bright and 0.2 seconds for detect_dim. Cumulative average time of function executions is 0.33 seconds which indicates that only 36% of the average total execution time is spent on actual frame processing. File access mandates additional restrictions on the global execution rates as well. Currently used production environment, Fermi cluster, can run a total of 144 concurent processes. However, using more than 36 will put such a strain on fermi-stor nodes, responsible for data access, that they enter swap mode and reduce the total execution time even further. Total predicted execution time for the full SDSS data set with 36 concurent processes is approximately 20 days.

File access issues plague the detection rates as well by limiting the amount of processing time that can be dedicated to a frame. Overstepping the 1 second average execution time per frame increases the total execution time for the full SDSS data set nonlinearly. Currently, using a selected set of parameters, approximately 70% of existing trails are detected in the i filter. Due to the robustness of used algorithm, I expect these rates to hold for other filters as well.

File access issues are solvable by focusing on a more pipelined execution approach, by increasing the total HDD space availible, or by using SSD disks. In a pipeline approach the expected speedup range varies from factor 1 to factor 2. If we had enough HDDs to be able to extract the entire SDSS data set, we would effectively avoid the need to extract each frame individually

thus, gaining a speedup factor of ~2. In the same scenario but with modern SSD disks, due to their low latency and larger bandwith, speedups up to a factor of 4 can be expected.

In conclusion, our LFDS is a success. Even if the detection rates could be higher, the performance is satisfactory. Under the assumption that performance and detection rates measured for i filter hold for other filters as well, we could detect approximately 75 000 trails left by meteors or satelites which would be the largest such collection of trails detected so far in astronomical setting. Further reduction of the resulting data set, by using meteor-satellite separating criteria described in section-1.1, would accurately provide us with a set of trails left behind only by meteors. Providing a full statistical analysis of the reduced set would shine the first light on the fainter end of the luminosity distribution of meteors.

Bibliography

Ahn, C. P., Alexandroff, R., Allende Prieto, C., Anderson, S. F., Anderton, T., Andrews, B. H., Aubourg, É., Bailey, S., Balbinot, E., and et al. (2012). The Ninth Data Release of the Sloan Digital Sky Survey: First Spectroscopic Data from the SDSS-III Baryon Oscillation Spectroscopic Survey. *The Astrophysical Journal Supplement*, 203:21.

Alam, S., Albareti, F. D., Allende Prieto, C., Anders, F., Anderson, S. F., Anderton, T., Andrews, B. H., Armengaud, E., Aubourg, É., et al. (2015). The Eleventh and Twelfth Data Releases of the Sloan Digital Sky Survey: Final Data from SDSS-III. *The Astrophysical Journal Supplement*, 219:12.

Bektešević, D. (2013). Star removal on SDSS images. Bachelor thesis availible at http://vinkovic.org/Projects/MindExercises/radnje/2013_Dino.pdf.

Canny, J. (1986). A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, Institute of Electrical and Electronics Engineers Transactions on*, PAMI-8(6):679–698.

Cikota, A., Bektešević, D., Cikota, S., Jevremović, D., and Vinković, D. (2013). Meteor science with survey telescopes - the case of sdss meteors. In *Proceedings of the* 1^{st} *LSSTEurope: The Path to Science conference (Cambridge, United Kingdom, Septembed 2013)*. Institute of Astronomy, University of Cambridge.

Dawson, K. S., Schlegel, D. J., Ahn, C. P., Anderson, S. F., Aubourg, É., Bailey, S., Barkhouser, R. H., Bautista, J. E., Beifiori, A., Berlind, A. A., et al. (2013). The Baryon Oscillation Spectroscopic Survey of SDSS-III. *The Astronomical Journal*, 145:10.

Fischler, M. A. and Bolles, R. C. (1981). Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395.

Gray, J., Slutz, D., Szalay, S. A., Thakar, A. R., VandenBerg, J., Kunszt, Z. P., and Stoughton, C. (2002). Data mining the sdss skyserver database. *Microsoft Corporation Technical Report MSR-TR-2002-01*.

Gunn, J. E., Carr, M., Rockosi, C., Sekiguchi, M., Berry, K., Elms, B., de Haas, E., Ivezić, Ž., Knapp, G., Lupton, R., et al. (1998). The Sloan Digital Sky Survey Photometric Camera. *The Astronomical Journal*, 116:3040–3081.

Gunn, J. E., Siegmund, W. A., Mannery, E. J., Owen, R. E., Hull, C. L., Leger, R. F., Carey, L. N., Knapp, G. R., York, D. G., Boroski, W. N., et al. (2006). The 2.5 m Telescope of the Sloan Digital Sky Survey. *The Astronomical Journal*, 131:2332–2359.

Hough, P. V. C. (1959). Machine Analysis Of Bubble Chamber Pictures. In *Proceedings, 2nd International Conference on High-Energy Accelerators and Instrumentation, HEACC 1959*, volume C590914, pages 554–558.

IMO (1995a). Angular velocities of meteors. http://www.imo.net/visual/minor/shower/velocity. Accessed: 2015-09-08.

IMO (1995b). Angular velocities of meteors. http://www.imo.net/visual/minor/shower/length. Accessed: 2015-09-08.

IMO (1997). Angular velocities of meteors. http://www.imo.net/glossary#letterm. Accessed: 2015-09-16.

Ivezić, Ž., Tabachnik, S., Rafikov, R., Lupton, R. H., Quinn, T., Hammergren, M., Eyer, L., Chu, J., Armstrong, J. C., and SDSS Collaboration (2001). Solar System Objects Observed in the Sloan Digital Sky Survey Commissioning Data. *The Astronomical Journal*, 122:2749–2784.

Iye, M., Tanaka, M., Yanagisawa, M., Ebizuka, N., Ohnishi, K., Hirose, C., Asami, N., Komiyama, Y., and Furusawa, H. (2007). SuprimeCam Observation of Sporadic Meteors during Perseids 2004. *"Publications of the Astronomical Society of Japan"*, 59:841–855.

Jenniskens, P. (2006). *Meteor Showers and their Parent Comets*. Cambridge University Press.

McLeod, Norman, M. (1993). *Suggestions for Visual Meteor Observations*. American Meteor Society, Ltd. Revised in 1995.

Morris, R. V., Shelfer, T. D., Scheinost, A. C., Hinman, N. W., Furniss, G., Mertzman, S. A., Bishop, J. L., Ming, D. W., Allen, C. C., and Britt, D. T. (2000). Mineralogy, composition, and alteration of Mars Pathfinder rocks and soils: Evidence from multispectral, elemental, and magnetic data on terrestrial analogue, SNC meteorite, and Pathfinder samples. *Journal of Geophysical Research*, 105:1757–1818.

Padgham, C. A. (1967). The points on stars. *Physics Education*, 2(5):252.

Rubin, A. E. and Grossman, J. N. (2010). Meteorite and meteoroid: New comprehensive definitions. *Meteoritics and Planetary Science*, 45:114–122.

Solontoi, M., Ivezić, Ž., West, A. A., Claire, M., Jurić, M., Becker, A., Jones, L., Hall, P., B., Kent, S., Lupton, R. H., et al. (2010). Detecting active comets in the {SDSS}. *Icarus*, 205(2):605 – 618.

Stoughton, C., Lupton, R. H., Bernardi, M., Blanton, M. R., Burles, S., Castander, F. J., Connolly, A. J., Eisenstein, D. J., Frieman, J. A., et al. (2002). Sloan Digital Sky Survey: Early Data Release. *The Astronomical Journal*, 123:485–548.

Suzuki, S. and Keiichi, A. (1985). Topological structural analysis of digitized binary images by border following. *"Computer Vision, Graphics, and Image Processing"*, 30(1):32 – 46.

Teh, C. H. and Chin, R. T. (1989). On the detection of dominant points on digital curves. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(8):859–872.

Toussaint, G. (1983). Solving geometric problems with the rotating calipers. In *Proceedings of the Mediterranean Electrotechnical Conference (Athens, Greece, May, 1983)*. Institute of Electrical and Electronics Engineers.

Trigo-Rodriguez, J. M., Rietmeijer, F., Llorca, J., and Janches, D. (2008). *Advances in Meteoroid and Meteor Science*. Springer.

Vokrouhlický, D. and Farinella, P. (2000). Efficient delivery of meteorites to the Earth from a wide range of asteroid parent bodies. *Nature*, 407:606–608.

York, D. G., Adelman, J., Anderson, Jr., J. E., Anderson, S. F., Annis, J., Bahcall, N. A., Bakken, J. A., Barkhouser, R., Bastian, S., and SDSS Collaboration (2000). The Sloan Digital Sky Survey: Technical Summary. *The Astronomical Journal*, 120:1579–1587.