Contents

Light Curve Analyzer

1 Usage

The usage can be divided into two main steps:

1. Obtain parameters of filters
2. Systematic search in chosen database

There are a tool for every of these steps. The first helps to calculate the propriety of chosen filters and find optimal values for them. The second proceed searching into the database according to file of queries. Results are saved into a log file and can be easily managed by the tool for reading the log file.

- Input
    1. Light curves of searched type of stars
    2. Light curves of unwanted types of stars in order to learn what is not searched
    3. Specify filters which will be used, **it is possible to implement own**
    4. **Specify database connector for searching (can be even searching thru downloaded light curves in a folder) or implement own**
- Verify
    1. Input light curves
    2. Filters

3. Database connector

- Calculate parameters of the filters

  1. Get best parameters of the filters
  2. Get accuracy of this filters (according this user can decide if a filter will be used or estimate accuracy pf the method)

- Searching

  1. Specify extent of searching (e.g. by creating query file)
  2. Systematic search in the specified database
  3. Save matched light curves as files (by default - **can be changed)**
  4. **Create the status file (query file with new columns of searching status)**
  5. **In case of interuption, re-run searching according to last log**

2 GUI

2.1 Parameters estimation tool

During launching of the tool filtres are searched in src.stars_processing.filters_impl (All classes which inherits *BaseFilter* are considered as filters).

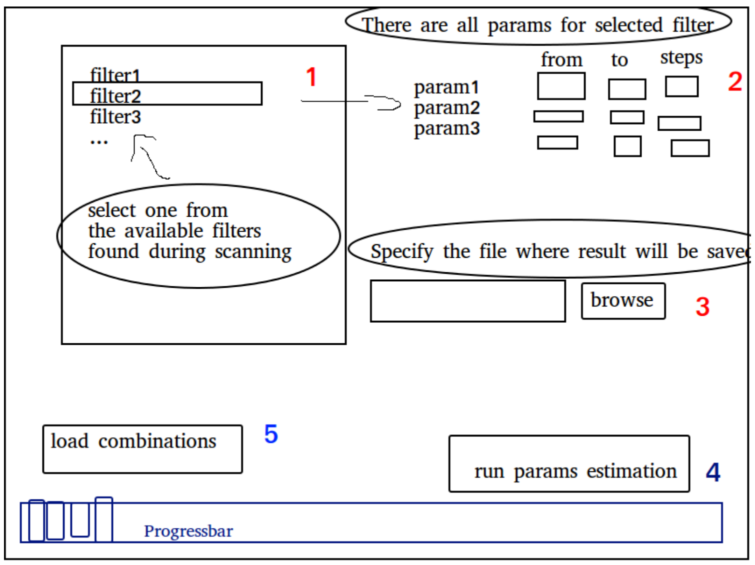
There are all params for selected filter
filter1
filter2
filter3
...
1
from
to
steps
2
param1
param2
param3
select one from the available filters found during scanning
Specify the file where result will be saved
browse
3
load combinations
5
run params estimation
4
Progressbar

Figure 1: So far just a draft of the GUI

There are available list of filters on the left panel 1(see picture 1). After one is selected all parameters with boxes for specifiation of the searching exension will apear on the right 2. Then config file for saving the result params need to be specified.

The second option is to press button 5 and load file which contains all combinations of parameters. Then is no need to specify ranges.

Finally searching for the most optimal parameters can be runned by the button 4.

2.2 Searching tool

During launching of the tool, packages are scanned in order to find all filters and database connectors. Database connectors are searched in src.db_tier (all classes which inherits *LightCurvesDb* or *StarsCatalogue* are considered as database connectors).

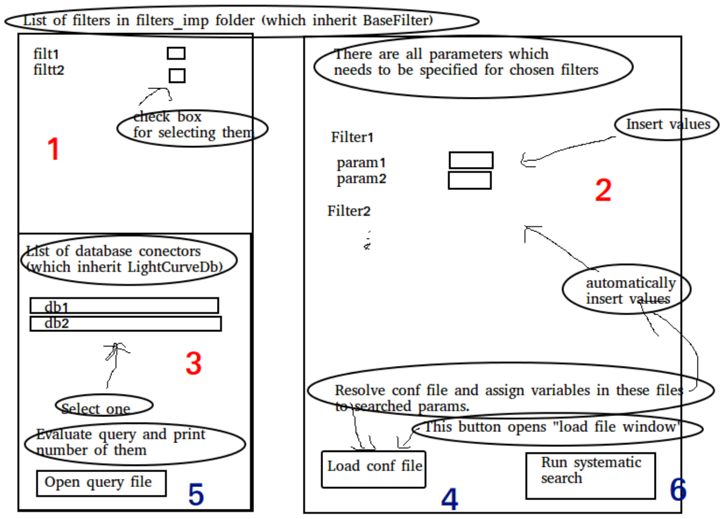
List of filters in filters_imp folder (which inherit BaseFilter)
filt1
filtt2
check box for selecting them
1
There are all parameters which needs to be specified for chosen filters
Insert values
Filter1
param1
param2
2
Filter2
automatically insert values
List of database conectors (which inherit LightCurveDb)
db1
db2
3
Select one
Evaluate query and print number of them
Open query file
5
Resolve conf file and assign variables in these files to searched params.
This button opens "load file window"
Load conf file
4
Run systematic search
6

Figure 2: So far just a draft of the GUI

Both filters and db connectors are then verified and showed on the left panel (on the image 2 - 1 and 3). After checking boxes at filters all parameters which need to be specified are showed up on the right in the panel 2. Values can be inserted manualy or by loading config file which contains these values (manualy created or by params estimation tool) by the button 5.

From the list of available database connectors just one can be chosen (by a click). The extension of the searching has to be specified in the query file which contains particular queries (see below) - loaded by the button 5. The file is also validated.

Finally searching can be started by the button 6.

3 Structure of the package

- *entities*
  - The most elemental classes of the program (e.g. *Star*, *LightCurve*, *Declination* etc.)
- *db_tier*
  - Classes which provide unitary data from the databases (Objects of *Star* type)
- *utils*
  - Modules for analysing time series, support methods for other classes, methods for visualizing data etc.
- *stars_proccesing*
  - Modules for proccesing light curves and sorting them by filters
- *commandline*
  - Main modules for executing the program and its tools
- *conf*
  - Configuration files containing information about structure of data folders and classes for estimating parameters of the filters
- *tests*
  - Test classes for checking performance of the program

3.1 Entities

Here is an example of creating a *Star* object containing a light curve. Anyway it is not necessary to create these objects manuly, because they are created by *StarProviders* (e.g. databse connectors).

```
import numpy as np
from entities.right_ascension import RightAscension
from entities.declination import Declination
from entities.light_curve import LightCurve
from entities.star import Star

#Coordinate values and its units
ra_value, ra_unit = 5.5 , "hours"
dec_value, dec_unit = 49.1, "degrees"

#Identifiers of the star
db_origin, identifier = "ogleII", {"field": "LMC_SC1", "starid": 123456,
    "target":"lmc"},

#Data for the ligh curve
time_data = np.linspace(245000, 245500, 500)
mag_data = np.sin(np.linspace(0,10,500)) + np.random.rand()
err_data = np.random.rand(500)


#Creating coordinate objects
ra = RightAscension(ra_value, ra_unit)
dec = Declination(dec_value, dec_unit)

#Creating light curve object
lc = LightCurve([time_data, mag_data, err_data])

#Creating star object
star = Star({db_origin: identifier}, ra, dec, {"v_mag" : v_mag, "b_mag" :
     b_max})
star.putLightCurve(lc)
```

LightCurve

This the most fundamental class of the program, however it is not accessed directly but just like the attribute of *Star* object (see below). It contains data about the light curve (times, magnitudes and optionally errors).

Star

This the most fundamental objects of the program. Star object contains information about the inspected astronomical body – identificators, coordinates, light curve etc..

One of the most importent attributes of the *Star* object is the indetifier. It is taken into account that there are many possible names of the stars, so name of the database has to be specified for every identifier. For example star which has identifier in both ogle and macho db:

```
ogle_ident = {"field": "LMC_SC1", "starid": "1"}
macho_ident = {"starid": "12.345.678"}

star = Star(ident = {"ogle":ogle_ident, "macho": macho_ident})

assert star.ident["ogle"] == ogle_ident
assert star.ident["macho"] == macho_ident
```

In case that "name" is not specified for identifier in certain database, it is automatically created according other subidentifiers as is shown below:

```
ogle_ident1 = {"field": "LMC_SC1", "starid": "1"}
ogle_ident2 = {"field": "LMC_SC1", "starid": "1", "name": "LMC_SC1_1"}
star1 = Star(ident = {"ogle":ogle_ident1})
star2 = Star(ident = {"ogle":ogle_ident2})

print star1.name
--> ogle_field_LMC_SC1_starid_1

print star2.name
--> LMC_SC1_1
```

The second most importent attribute are coordinates (see below). Adittional information about the star is hiden in *more* dictionary. There are some naming convention the keys in *more*, but there can be anything. In case of writting new filters or db connectors is necessary to keep in mind how certain variables was named. For example from OGLEII db color indexes are also obtained and stored into star objects by convention "*v_i*", "*b_v*" ... for $V - I$, $B - V$ ..., so color index filter has to access these values by this convention. Moreover in case of new db connectors or othe color index filters, it is advised to keep the convention.

Last but not least there is the attribute *starClass* which carry the information about type of the star (quasar, cepheid etc.).

AbstractCoordinate, Declination and RightAscension

The *AbstractCoordinate* is the common abstract class which is inherited by both *Declination* and *Right Ascension*. These two classes differ just in used units and its restrictions. Coordinate unit can be specified and values are then checked whether they corespond with coordinate restrinctions (see example code at the beginning the subsection).

Exceptions

There are all exceptions which can be raised by the program

3.2 Db tier

In the example below stars in certain area in OGLEII database are obtained.

```
from entities.right_ascension import RightAscension
from entities.declination import Declination
from db_tier.stars_provider import StarsProvider

db_key = "ogle"
query ={
        "ra":RightAscension(5.56, "hours"),
        "dec":Declination(-69.99),
        "delta":3,
        "target":"lmc"
         }
```

Database connector class is resolved by *db_key* (see *StarsProvider* below). For example for loading stars from folder input would look like this:

```
db_key = "file"
query = {"path" : path_to_the_folder_of_light_curve_files}
```

Moreover *OgleII* client class supports query via field-starid-target or starcat-target. The type of query is resolved automatically by keys in *query* dictionary.

```
db_key = "ogle"
query ={"field": "LMC_SC1", "starid": 12345, "target": "lmc"}
```

The second part of the example (below) is common for every database and every type of query (beause of *StarsProvider* interface).

```
stars_prov = StarsProvider().getProvider(obtain_method = db_key,
                                         obtain_params = query)
stars = stars_prov.getStarsWithCurves()
```

StarsCatalogue

Common abstract class for every database connector which provides *Star* objects. Every inherited classes need to impelement method *getStars*.

LightCurvesDb

The *LightCurvesDb* is common class for all connector classes to the databases which contain light curves data. It also inherit *StarsCatalogue* abstract class. That means that these connectors provide *Star* objects enchanted by *LightCurve* attributes.

StarsProvider

All database connectors are not accessed directly, but by this interface. It reads from config file which contains keys and name of the database connector module in db_tier package. That allows to call db connectors by keys (see above in example).

CrossmatchManager

NEED TO BE UPGRADED

OGLEII

Connector class for OGLEII database thru web interface

FileManager

Manager for obtaining *Star* objects from the light curve files. Names of the stars are resolved according to file names (without suffix).

TapClient

Common class for connectors to the databases via TAP protocol. The class can be accessed directly. In example below complete light curve of the star "MyStar1" is obtained in the database "http://my_db.org/tap" from table "light_curve_table".

```
tap_params = {
"URL" : "http://my_db.org/tap",
"table" : "light_curve_table",
"conditions" : ("ident1": "MyStar", "ident2": 1),
"select" : "*"}

lc = TapClient().postQuery(tap_params)
light_curve = LightCurve(lc)
```

MachoDb

TODO: NEED TO BE UPGRADED

3.3 conf

The idea of conf files is about having all parameters in one place. The configuration file can be created like an ordinary text file in format *variable_name* = *variable_value* per row or it can be created automatically by tools for estimating the most optional parameters.

glo

Global parameters of the program. There are strucure and paths to the data folders (e.g. light curve folder of certain stars), verbosity level etc.

DefaultEstimator

Estimator class which is used by *ParamsEstimation* class (see below) by default. As all estimator classes it has to inherit methods of *GridSearch* library also it has to implement *fit* and *score* methods. The first method learns to recognize input stars according to the input sample and the second one calculate precision of examined combination.

ParamsEstimation

The class for calculating best filter parameters according to input sample of searched and undesired stars. It is also needed to specify combination of parameters for the testing and the estimator (see above). In the following example the *AbbeValueFilter* is tested. The result is saved into the file and printed.

```
tuned_params = [{"abbe_lim": 0.35},{"abbe_lim": 0.4}, {"abbe_lim": 0.45}]
es = ParamsEstimation(quasars, stars ,AbbeValueFilter , tuned_params)
es.fit()
```

filters params

There are configuration files for certain projects (e.g. filters parameters for quasars searching) in this subpackage.

3.4 utils

data analysis

Mainly there are functions for processing time series (e.g. calculate histogram, normalize, PAA etc.).

commons

There are decorators for validating input/output of functions. Usage is shown in example below.

TODO: NEED TO BE CHECKED

```
class Foo(object):
    @mandatory_args(("param1"),("param2","param3","
    @default_values(param5 = 1, param6 = 11)
    @args_type(param1 = (list ,), param2 = (str , int
        int , float), param4 = (str ,float))
    def __init__(self,*args,**kwargs):
            ...
```

Every *Foo* object has to be initialized with one or three parameters. In case of one it has to be list in case of three params - first has to be string or int etc. Also if *param*5 and *param*6 is not specified their value would be set by default decorator.

helpers

There are support functions (e.g. progress bar)

output process modules

Functions for serializing objects. Especially saving/loading them into/from the files.

Stars

Methods for managing lists of *Star* objects. For instance *get_sorted_stars* takes list of stars and returns the dictionary where keys are types of the stars (e.g. quasars, cepheids etc.) and its values are lists of the stars of the certain type.

Also there are methods for visualizing stars.

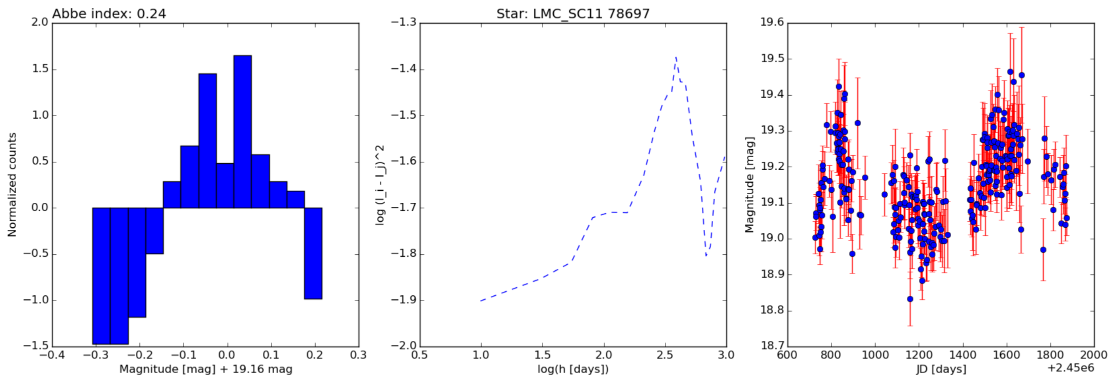

Figure 3: Output graph of *plotStarsPicture* method

3.5 stars processing

This package contains three subpackages:

- filters tools
  - Support modules for particular filter implementations
- filters impl
  - Star filters implementations
- systematic search
  - Modules for systematic searching of databases and filtering its results

BaseFilter

Every filter has to inherit this class which ensures that every filter class has *applyFilter* method.

filters

Filter classes are located in src.stars_processing_filters_impl package (program will be searching for them in this location) and they have to inherit *BaseFilter* class then is necessary to impelement *applyFilter* method which returns list of *Star* objects passed thru filtering. There is a convention to use decorators in order to ensure correct type flow as is shown in the example below.

```
from utils.commons import returns,accepts
from stars_processing.filters_tools.base_filter import BaseFilter

class MyFilter(BaseFilter):
        ...
    @returns(list)
    @accepts(list)
    def applyFilter(self,stars):
        '''
        Filter stars according to ...

        @param stars: List of star objects (containing light curves)
        @return: List of star-like objects passed thru filtering
        '''
        ...

    @accepts(Star)
    def _filterStar(self,star):
                '''
                @param star: Star type object
                @return: True if star passed, else --> False
                '''
                ...
```

There are several filters which are already implemented:

- AbbeValueFilter
  - It filter stars according to Abbe value of their light curves
- ColorIndexFilter
  - It filter stars according to their color indexes
- ComparingFilter
  - It takes reference stars as filtering parameters and sorting is done according to comparing subfilters which decide how filtering will be done. So far

there are *CurvesShapeFilter*, *HistShapeFilter* and *VarioShapeFilter*. They let pass just stars which light curves/histograms/variograms shape (in these implementation symbolic representation is taken) is similar as the reference stars.

- CurveDensityFilter
  - It filter stars according density of sampling data in light curves in order to get rid of inappropriate curves.

There some examples of initializing filters. Let's start with Abbe value filter which denies stars with Abbe value of their light curves higher then *ABBE_LIM*:

```
abbe_filter = AbbeValueFilter(abbe_lim = ABBE_LIM)
```

For many filters is their creation one-line business. Anyway for *CompraingFilter* is needed to create subfilters and decision functions which defines treshold for dissimilarity of subfilters.

```
#Prepare comparative sub filters in order to load them into comparing
    filter
cf = []
cf.append(HistShapeFilter(days_per_bin = HIST_DAYS_PER_BIN,
                          alphabet_size = HIST_ALPHABET_SIZE))
cf.append(VariogramShapeFilter(days_per_bin = VARIO_DAYS_PER_BIN,
                               alphabet_size = VARIO_ALPHABET_SIZE))

#Define decision function which decides whether a star
#will pass thru filtering (according its histogram
#and variogram distance from template
def dec_func_t(distances):
    hist_dist, vario_dist = distances
    return hist_dist < HIST_TRESHOLD and vario_dist < VARIO_TRESHOLD


#Load comparative sub filters, template stars
#of quasars and decision function
comp_filt = ComparingFilter(cf, quasars, dec_func_t,
                            search_opt="closest")
```

FilteringManager

This class manages filtering of stars. It loads all filters, verify and apply them. Once filters are initialized is easy to load them into *FilteringManager*:

```
#Load inspected stars and filters
filteringManager = FilteringManager(stars)

filteringManager.loadFilter(comp_filt)
filteringManager.loadFilter(abbe_filter)

#Perform filtering and return stars passed thru filter
result_stars = filteringManager.performFiltering()
```

StarsSearcher

Master class for searching stars in databses and their filtering. It loads filters and list of queries for specified database. Results of searching is saved into the status file.

```
#Prepare queries (stars in OGLEII galactic bulge in the first field from
    starid 1001 to 1008)
queries = []
for i in range(1001,1009):
        queries.append({"starid":i, "field_num":1,"target":"bul"})

#Prepare filters
abbe_filt = AbbeValueFilter(0.7)

searcher = StarsSearcher([], SAVE_PATH = "test_light_curves/", SAVE_LIM =
     1, OBTH_METHOD="ogle")
search.queryStars(queries)
```

There is *matchOccur* method which is called if any of inspected star passed thru all filters. By default the light curve of the star is saved into *SAVE_PATH* location. Anyway is possible to change the behavior by overwritting this method as is shown in example below for ogleII searcher.

```
class OgleSystematicSearch(StarsSearcher):
    '''
    This is systematic searching and filtering class for OGLEII.
    '''

    OBT_METHOD = "ogle"

    def __init__(self, filters_list, SAVE_PATH = None, SAVE_LIM = None,
        UNFOUND_LIM = None):
        '''
        @param filters_list: List of filter type objects
        @param SAVE_PATH: Path from "run" module to the folder where
            found light curves will be saved
        @param SAVE_LIM: Number of searched objects after which status
            file will be saved
        @param UNFOUND_LIM: Number of unsuccessful query to interrupt
            searching
        '''
        StarsSearcher.__init__(self, filters_list, SAVE_PATH, SAVE_LIM,
            UNFOUND_LIM,OBTH_METHOD = self.OBT_METHOD)


    def matchOccur(self, star, query):
        '''
        What to do with star in case of passing thru filtering

        @param star: Star type object
        @param query: Dictionary of query for this object
        '''

        verbose(star,2, VERBOSITY)
        path = "%s%s_%s" % (self.save_path, query["target"],query["
            field_num"])
        create_folder(path)
        star.saveStar(path)
```

StatusResolver

This class manages status files - it reads them, exctracts desired information etc. Queries for *StarsSearcher* can be read from status file-like file. In the following format:

```
#first_query_param      second_query_param      ...  ...
a_value1      a_value2     ...              ...
b_value1            b_value2      ...       ...
           ...
```

Header begins with "#" symbol and then there are column names separated by the tabulator. This line is resolved as the query keys. During searching the query file is transforming into the status file by adding new columns with status as is shown below:

```
#param1          param2        ...       ... found     filtered     passed
 a_value1        a_value2      ...       ... True      True      False
 b_value1        b_value2      ...       ... False     False     False
 c_value1        c_value2      ...       ... True      True      True
 d_value1        d_value2      ...       ...
```

This status file says that first star was found in the database, filtered but it was not passed. Second star was not found, the third one passed thru all filters and the last one has not been queried so far.

In case of interupted query *StatusResolver* is able to find where quering ended according to status information (the last three columns).

3.6 commandline

There are executable modules.