N2 - Photometry of open star clusters

Remark: This article describes the modern data reduction and data analysis for the N2 experiment using an almost fully automated *Python* script for the reduction of flatfields and dark frames as well as the stacking and alignment of the star cluster images. The classical version, where the displacements are measured by hand and manually entered into a routine, is described in the article Photometry of an open star cluster (classical) (only in German). Alternatively, there is also a semi-automatic version based on *GDL*: Photometry of an open star cluster (GDL).

Task

Take photometry of two open star cluster (in two different filters) and create the color-magnitude diagrams for these clusters. The main task is the determination of the cluster ages. Look for suitable clusters in advance of the observation, e.g. at *Simbad* - a help page for the parameter searches can be found here.

Criteria to be fulfilled for the clusters are:

- High number of stars, at least 100 stars should be inside the cluster
- The star density should not be so dense that the diffraction discs of the stars merge with each other
- Field of view of the camera with the telescope covers most of the cluster
- Size of the cluster is not too small
- No strong brightness contrasts between the individual stars

Good example	Bad examples
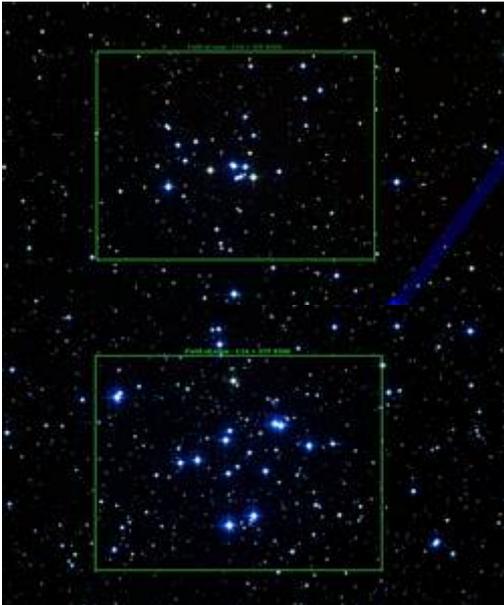
Observation

Night observations at the OST of the University of Potsdam (alternatively also at the 70cm telescope of the AIP) are required. Please refer to the checklist for night observations for preparation.

For the data reduction, flatfields and darks must be taken in addition to the actual images of the star clusters. Bias images (with zero exposure time) are not necessary, if for each set of star cluster images and flatfields also darks with the same exposure times are taken. At the OST, the flatfields can be taken with a flat field panel after observing the star clusters. If observing at the AIP, it is recommended to take the flatfields already at dusk (or against a white evenly illuminated wall). In this case, bias images are also required and should be taken with the cover closed. In any case, in absolute darkness and with the covers closed, the dark frames should be acquired. To minimize noise, each set of darks and flats should consist of at least 30 individual exposures.

After darkness has fallen, the images of the star clusters are taken. To get a sufficient number of counts also from fainter stars, the exposures should be long enough. Depending on the camera and the observed object the time span for single images can be between 20 seconds and several minutes. Especially if brighter objects are in the field of view or if the accuracy of the tracking is insufficient, it is useful to take several exposures and add them up later. With the typical seeing in Potsdam, the currently used cameras can be operated with 2×2-Binning or 3×3-Binning to further increase the signal to noise ratio. In any case, a total exposure time of at least 40 minutes should be achieved per filter.

Data reduction

Preparations

Get an overview - View pictures

The first thing to do is to log in to the laboratory computer. The next step is to copy the observation data (FIT files), including flatfield and bias/darkframe images from the directory ~/data/<date> to your own directory ~/data_reduction/. There are several tools to view the two-dimensional CCD images (data arrays) stored in the FIT format. For example by means of *ds9*:

```
ds9 filename.fit
```

opens the image `filename.fit` with *ds9*. After opening an image, you can vary the brightness and contrast by moving the cursor while holding down the right mouse button. Further options (zoom, false color display, rotate, mirroring etc.) can be accessed via buttons. The coordinates of the current cursor position are displayed in the upper left corner. It is also possible to open several images at once (buttons `Frame → new frame`, then open another file with `File → open`). With the blink option (buttons: `Frame → blink`) one can compare several images easily.

Alternatively, all images can be opened at the same time. For this purpose

```
ds9 *.fit
```

(but this is not suitable for a large number of files). In *ds9* the frames can be viewed as above or each frame can be viewed individually one after the other (button `Frame → Single Frame`). You can switch between the frames by pressing the tab key.

The **usable** frames are to be selected for further processing. For example, the stars should be seen as round discs. Images with oval-shaped stars are not to be used.

Install the pipeline

Some modules from the OST photometry pipeline are required for the data reduction and analysis. Python modules should always be installed in a virtual environment to reduce dependency issues. A virtual environment can be created using

```
mkvirtualenv ost_photometry
```

By doing so, we have named the virtual environment `ost_photometry`. The fact that we are in the virtual environment is indicated by the string '(ost_photometry)', which now precedes each terminal line. To leave the virtual environment, simply type

```
deactivate
```

If you want to reconnect, you can do so by typing

```
workon ost_photometry
```

This is also necessary if you reconnect to a12, e.g. after a break, and want to continue the data analysis.

The OST photometry pipeline can then be installed in the terminal using *pip* as follows

```
pip install ost_photometry
```

All necessary dependencies are also installed in this way.

Reduction pipeline: darkframes, flatfields, and image stacking

In order to cope with a larger amount of data, there is a *Python* routine that performs the corrections for darkframe and flatfield per filter, then adds up the images per filter and aligns them to each other. The routine does not perform any quality control of the images, so unusable observations must be sorted out beforehand, otherwise alignment problems may occur.

Copy the *Python* script `1_add_images.py` from the directory `~/scripts/n2/` into your local working directory. After that you should open it with a text editor of your choice to adjust the paths for the images. To be able to read and verify a larger amount of images, the program expects a separation of the data into different subdirectories (variables: `bias`, `darks`, `flats`, `imgs`). There should be one directory each for the images of the star cluster, the flatfields, and the dark frames. A possible directory structure could be:

```
/bias/
/darks/
/flats/
```

```
/imgs/
```

The *Python* script automatically detects the filters and exposure times used. Based on this, it arranges and classifies the files automatically without any further interaction. If you are sure that all `FIT`-`Header` keywords are set correctly, you can try to put all files into one directory. In this case only the path `rawfiles` must be set in the script. Otherwise, the paths to the subfolders for the flats, darks, etc. must be specified. Hence either `bias`, `darks`, `flats`, and `imgs` musst be specified or only `rawfiles`.

Configuration section of `1_add_images.py`:

```
######################### Individual folders ##########################
### Path to the bias -- If set to '?', bias exposures are not used.
bias = '?'

### Path to the darks
darks = '?'

### Path to the flats
flats = '?'

### Path to the images
imgs  = '?'

###################### Simple folder structure #########################
rawfiles = '?'
```

Once the path information and the name of the star cluster have been specified, the script can be executed with

```
python 1_add_images.py
```

The results are saved in a new subdirectory called `output`.

Data analysis

It is recommended not to execute all the following steps on the command line of *Python*, but to write a small script analysis.py (can also be named completely different). To do this, open the desired program file with a text editor (in the following case *Kate*):

```
kate analysis.py &
```

At the beginning of the *Python* script we first include the required modules. In our case these are Numpy, some Astropy modules, Astroquery, and some parts of our OST library:

```
import numpy as np
```

```
from astropy.coordinates import SkyCoord, matching
import astropy.units as u
from astropy.table import Table

from astroquery.vizier import Vizier

from ost_photometry.analyze.analyze import main_extract
from ost_photometry.analyze.plots import starmap, scatter
from ost_photometry.utilities import (
    find_wcs_astrometry,
    Image,
)
from ost_photometry.analyze.utilities import (
    clear_duplicates,
)
```

```
import warnings
warnings.filterwarnings('ignore')
```

The last two lines disable some warnings that unnecessarily clutter the console output.

Defining some variables

Next, some variables should be defined. These should be at least the name of the cluster (`name`), the directory where the results should be stored (`out_path`) and the paths (`V_path` and `B_path`) to the two images of the cluster in the considered filters (here V and B) obtained from the [reduction pipeline](#).

```
#   Cluster name (recognizable by Simbad/Vizier)
name = 'NGC7789'

#   Directory to save the data
out_path='output/'

#   Images
V_path = 'output/combined_filter_V.fit'
B_path = 'output/combined_filter_B.fit'
```

Note: The variable names given here and in the following are only examples and can be replaced by any other name.

Note: If the images are not in a subdirectory of the current directory, the path can also refer to the next higher level by means of `../`.

Reading in the images

We open the FIT files with image data by means of the `image` function provided by the OST library. This has the advantage that we do not have to worry about the details of the read-in process, and at the same time we have a *Python* object for each image, which we can use to store some of the results

obtained in the following steps. The `image` function has the following arguments: 1. index of the image (can be set to 0), 2. filter name, 3. path to the image file and 4. path to the output directory:

```
#   Load images
V_image = Image(0, 'V', V_path, out_path)
B_image = Image(0, 'B', B_path, out_path)
```

World Coordinate System

The images created by the OST are usually delivered without a so-called WCS. WCS stands for World Coordinate System and allows to assign sky coordinates to each pixel in the image. In *ds9* these coordinates will be displayed in the coordinates window of *ds9* when pointing with the mouse pointer on certain pixels or objects. This is very helpful if you want to compare the positions of stars in your own image with those in star catalogs. This could be quite helpful for the calibration of the stellar magnitudes later on 😉.

We will use the function `find_wcs_astrometry` to determine the WCS:

```
#   Find the WCS solution for the images
find_wcs_astrometry(V_image)
find_wcs_astrometry(B_image)
```

When a WCS solution is found,

```
 WCS solution found :)
```

is printed in green on the command line.

Localization of the stars

Finding the stars

The identification of the stars in the two images is performed using the `main_extract` function. This function takes as the first argument the `image` object. The second argument characterizes the size of the diffraction discs. This so called sigma can be determined from the images. But it is usually around `3.0`. As an optional argument, the extraction method can be selected (`photometry`). Here we specify `'APER'`, and thus select aperture photometry, where the flux of the individual objects and the associated sky backgrounds is read out within fixed apertures (here circular and ring-shaped, respectively). To specify these apertures, we have to give a radius for the circular object aperture (`rstars`) and two radii for the annular background aperture (`rbg_in` and `rbg_out`). In previous observations, the respective values were 4, 7, and 10, respectively. The radii are in arc seconds.

```
#   Extract objects
main_extract(
    V_image,
```

```
    sigma,
    photometry_extraction_method='APER',
    radius_aperture=4.,
    inner_annulus_radius=7.,
    outer_annulus_radius=10.,
)
main_extract(
    B_image,
    sigma,
    photometry_extraction_method='APER',
    radius_aperture=4.,
    inner_annulus_radius=7.,
    outer_annulus_radius=10.
)
```

In addition to the star coordinates (in pixels), `main_extract` also automatically stores all extracted fluxes in the `image` objects.

Check identified stars

The function `main_extract` has the nice feature that it marks the identified stars on a so called "starmap". This can be used to check if enough stars were identified. The starmaps are located in the output directory (variable: `out_path`) and there in the subdirectory `starmaps`. If not enough stars have been identified or noise has been falsely identified as stars, the `sigma` parameter should be adjusted in the call of `main_extract`.

Preparing the extraction results

For the subsequent steps in the tutorial we need the extracted fluxes and the star position preferably in the form of Astropy tables. These tables can be easily obtained from the `image` objects:

```
#   Get table
photo_V = V_image.photometry
photo_B = B_image.photometry
```

From these, in turn, the specific star positions (in pixels) can be easily extracted:

```
x_V = photo_V['x_fit']
y_V = photo_V['y_fit']

x_B = photo_B['x_fit']
y_B = photo_B['y_fit']
```

Cross correlation and sorting of results

Next we have to identified those stars that are present in both filters. This is done using the `astropy.coordinates` package, more precisely using the correlation functions for datasets

provided by this package. Since these functions work on the basis of celestial coordinates rather than pixel coordinates, we need to convert our previously determined pixel coordinates into a suitable coordinate system. For this purpose it is convenient that we have determined the WCS before.

First, we create `SkyCoord` objects for each of the datasets from the two filters. These objects, once defined, allow us to output the coordinates in a variety of already predefined coordinate systems. Furthermore, and even more convenient, these objects are also accepted as arguments by a number of *Astropy* functions and classes. For this reason, you usually don't need to worry about which coordinate system you are working in, since this is all handled internally by *Astropy*. We define our `SkyCoord` objects using the option `.from_pixel()`, which allows us to define them directly based on the pixel coordinates and the previously determined WCS (which we can take from the `image` object).

```
#   Create SkyCoord objects
coords_V = SkyCoord.from_pixel(x_V, y_V, V_image.wcs)
coords_B = SkyCoord.from_pixel(x_B, y_B, B_image.wcs)
```

These two `SkyCoord` objects can then be correlated with each other by means of the function `search_around_sky`. In addition to the two `SkyCoord` objects this function needs as third argument the allowed tolerance in the coordinates (below which two objects from both datasets are still recognized as the same). In our case we choose a generous 2 arc seconds. The unit is defined here by the `astropy.units` package that we loaded above using the abbreviation u.

```
#   Correlate results from both images
 id_V, id_B, _, _ = matching.search_around_sky(coords_V, coords_B,
2.*u.arcsec)
```

The successfully mapped stars get one entry each in `id_V` and `id_B`. These two lists (more precisely Numpy arrays) contain the index values that these stars had in the original unsorted datasets. This means that we can use these index values to sort the original tables with the fluxes and star positions in such a way that they only contain stars that were detected in both images and that the order of the stars in both data sets is the same. This assignment is essential for the further procedure.

Sorting is done simply by inserting the arrays with the index values into the tables. We select and simultaneously sort the stars identified in both images:

```
#   Sort table with extraction results and SkyCoord object
photo_V_sort = photo_V[id_V]
photo_B_sort = photo_B[id_B]

coords_objs = coords_V[id_V]
```

With the last line above we have also sorted one (which one doesn't matter) of the `SkyCoord` objects. This will be useful in the next but one step.

Conversion of fluxes into magnitudes

Since in the following we work in magnitudes, the fluxes must be converted accordingly. The

conversion can be done immediately on the basis of the tables extracted before (the fluxes are stored in the column `flux_fit`). The calculated magnitudes can also be added to the tables as a new column:

```
#   Calculate magnitudes
photo_V_sort['mag'] = -2.5 * np.log10(photo_V_sort['flux_fit'])
photo_B_sort['mag'] = -2.5 * np.log10(photo_B_sort['flux_fit'])
```

Note that the magnitudes are determined only to an additive constant as long as no calibration has been performed.

Calibration

The magnitudes are so far determined only up to a constant (the so-called Zeropoint). Calibration poses a significant problem without access to a database of comparison stars. Fortunately, the astronomical community offers such databases that we can use. We will use the **VizieR** database of the **Centre de Données astronomiques de Strasbourg** or obtain our calibration data from there. To acces this database we will use the `astroquery` package and from it the `Vizier` module.

Download calibration data

First we define the catalog we want to access. In our case, we use the **APASS** catalog, which runs under ID. `II/336/apass9`. Furthermore we define the columns we need. We limit ourselves here to the columns we really need to keep the download time short. The column names are partly catalog specific, so for another catalog other column names might have to be used.

```
#   Get calibration from Vizier
catalog = 'II/336/apass9'
columns = ['RAJ2000', 'DEJ2000', "Bmag", "Vmag", "e_Bmag", "e_Vmag"]
```

Then we define the `Vizier` object. We pass the catalog ID and the column definition to it and set the so-called `row_limit` to 10^6. The latter limits the table to be downloaded to 10^6 rows and thus the download volume. We do this to not run into a server timeout during the download.

```
v = Vizier(columns=columns, row_limit=1e6, catalog=catalog)
```

In the next step we can perform the actual download. For this purpose we use the function `.query_region`. We have to pass to it the coordinates and the size of the sky region to be queried. Fortunately, both are already known. We know the coordinates from the FIT headers of the star cluster images and the radius of the region is simply the field of view, which we already calculated above. Both values can be taken from the `V_image` object.

```
calib_tbl = v.query_region(V_image.coord, radius=V_image.fov*u.arcmin)[0]
```

The table `calib_tbl` now comprise all objects contained in the **APASS** catalog that are in our field of view with their B and V magnitudes.

⚠️ **Task:** Restrict the downloaded **APASS** catalog to all objects with V magnitudes in the 10 to 15 mag range. This will ensure that potentially overexposed as well as underexposed stars in our images are not used for the calibration.

Note: To accomplish this task, it might be helpful to learn a little about boolean masks, comparison operation, and boolean logic.

Alternatively to the **APASS** catalog, the 'Fourth U.S. Naval Observatory CCD Astrograph Catalog' (**UCAC4**) can be used for calibration, which has the ID `I/322A/`.

Cross correlation with the extracted data

The downloaded catalog must now be correlated with the star coordinates extracted above. For this purpose we create once again a `SkyCoord` object. This time for the calibration stars. Unlike above, we construct the `SkyCoord` object this time directly from the right ascension and declination coordinates, which we can take from the table `calib_tbl`. The right ascension values can be found in the column RAJ2000, whereas the declination values are in the column DEJ2000. Furthermore, the units for the coordinates must be specified. In our case these are degrees (`u.deg`). As the last argument (`frame`) the coordinate system should be specified. In our case we have to specify `icrs`.

```
#   Set up SkyCoord object with position of the calibration objects
coord_calib = SkyCoord(
    calib_tbl['RAJ2000'].data,
    calib_tbl['DEJ2000'].data,
    unit=(u.deg, u.deg),
    frame="icrs"
    )
```

As above, we correlate the calibration data with our results using the `search_around_sky` function from the `matching` module of *Astropy*. As arguments we pass the just defined `SkyCoord object` for the calibration stars, the `SkyCoord object` for the stars we found in both filters (`coords_objs`) and the maximum distance between stars in both datasets (below which they are still recognized as the same object).

```
#   Correlate extracted object position with calibration table
ind_fit, ind_lit, _, _ = matching.search_around_sky(
    coords_objs,
    coord_calib,
    2.*u.arcsec,
    )
```

In this way, we again obtain index values that we can use to select the calibration stars both from the datasets for the two filters and from the downloaded catalog:

```
#   Select data of the calibration stars
```

```
photo_V_sort_calib = photo_V_sort[ind_fit]
photo_B_sort_calib = photo_B_sort[ind_fit]

#   Select literature data of the calibration stars
calib_tbl_sort = calib_tbl[ind_lit]
```

Magnitude calibration

Now we are able to perform the actual calibration of the magnitudes. We calculate the so-called zeropoint by subtracting our extracted magnitudes from the magnitudes in the downloaded catalog for the calibration stars in each of the two filters. Then we can use the function `.ma.median` from the *Numpy* module to compute the median over all calibration stars:

```
#   Calculate zero points
ZP_V = np.ma.median(calib_tbl_sort['Vmag'] - photo_V_sort_calib['mag'])
ZP_B = np.ma.median(calib_tbl_sort['Bmag'] - photo_B_sort_calib['mag'])
```

Afterwards the calculated zeropoints have to be added to the magnitudes of the stars in the tables `photo_V_sort` and `photo_B_sort`. To guarantee reproducibility, the calibrated magnitudes should be added to the tables in a separate column:

```
#   Calibrate magnitudes
photo_V_sort['mag_cali'] = photo_V_sort['mag'] + ZP_V
photo_B_sort['mag_cali'] = photo_B_sort['mag'] + ZP_B
```

[Checking the calibration stars](#)

One way to check the validity of the calibration stars is to display them on a starmap (similar to what the `main_extract` above does automatically). But now we want to display the downloaded star positions as well as the stars that were actually used for the calibration later on. For this purpose the OST library offers a suitable function (`starmap`) which can create such plots. This function can be loaded via

```
from ost.analyze.plot import starmap
```

Since this function expects as input an astropy table, with the data to be plotted, we must first create it before we can plot the starmap. The position of the calibration stars are not yet available in pixel coordinates, because we got this information from the Simbad or Vizier database. Therefore, we need to generate these. At this point it is convenient that we have previously created a `SkyCoord` object for these stars. Using `.to_pixel()` and specifying the WCS of the image, we can easily generate pixel coordinates:

```
#   Calculate object positions in pixel coordinates
x_cali, y_cali = coord_calib.to_pixel(V_image.wcs)
```

Then we can use this data to create the new Astropy table:

```
tbl_xy_cali_all = Table(
```

```
    names=['id','xcentroid', 'ycentroid'],
    data=[np.arange(0,len(y_cali)), x_cali, y_cali]
    )
```

We now repeat the whole process for the `SkyCoord` object, which contains only the stars that were both in the database and identified on the two images (both filters) of the cluster:

```
x_cali_s, y_cali_s = coords_objs.to_pixel(V_image.wcs)

tbl_xy_cali_s = Table(
    names=['id','xcentroid', 'ycentroid'],
    data=[np.arange(0,len(y_cali_s)), x_cali_s, y_cali_s]
    )
```

After that we have everything ready and can plot the starmap:

```
starmap(
    out_path,
    V_image.get_data(),
    'V',
    tbl_xy_cali_all,
    label='Downloaded calibration stars',
    tbl_2=tbl_xy_cali_s,
    label_2='Identified calibration stars',
    rts='calibration',
    nameobj=name,
  )
```

Here, the first argument is our output directory, the second argument is the actual image (as a *Numpy* array), the third argument is the filter label, the fourth argument is the first table, `label` is the label to the first dataset, `tbl_2` is the second table, `label_2` is the label to the second dataset, `rts` is a description of the plot, and `nameobj` is the name of the star cluster.

Alternatively, you can also create the starmap directly with the help of `pyplot` from the `matplotlib` module. This is not much more complex but offers more possibilities to customize the plot. You load `pyplot` by means of:

```
import matplotlib.pyplot as plt
```

The plot window is created via

```
fig = plt.figure(figsize=(20,9))
```

Then the actual image can be loaded:

```
plt.imshow(image, origin='lower')
```

`image` is the actual image data and `origin=lower` makes sure that the overplotting of the pixel coordinates works. Afterwards the symbols which mark the star position can be plotted:

```
    plt.scatter(x_positions, y_positions)
```

x_positions and y_positions are the x and y star positions in pixels. .scatter offers a variety of configuration options such as the selection of the symbol, color, line width, and much much more. Please refer to the various documentation and tutorials on the Internet. Also regarding labels, titles, legends, and axis labels more than enough information can be found online. The plot can be saved via

```
  plt.savefig(filename)
```

Here, filename is the file name or the path to the file. Alternatively, the plot can also be displayed directly via

```
  plt.show()
```

However, in this case the backend may need to be changed before plt.show() is called:

```
  plt.switch_backend('TkAgg')
```

At the end of the plot it should be closed by means of:

```
  plt.close()
```

Saving the results

Once the calibration is done, we should still save our extracted and calibrated magnitudes. Since the tables photo_V_sort and photo_B_sort contain some data that we do not need for the creation of the CMD and we do not want to save them for the sake of conciseness, we create a new table that contains only the relevant data. The new table can be easily created using Table(). Then we add to this table the columns from the tables photo_V_sort and photo_B_sort that are relevant for us:

```
  #   Create new table for the CMD
  results = Table()
  results['id'] = photo_V_sort['id']
  results['x'] = photo_V_sort['x_fit']
  results['y'] = photo_V_sort['y_fit']
  results['B [mag]'] = photo_B_sort['mag_cali']
  results['V [mag]'] = photo_V_sort['mag_cali']
```

If the new table is filled, *Astropy* allows to save this table very comfortably with the command .write. As first argument we have to specify the path or file name under which the table should be saved. Furthermore we specify the format (we choose ascii) and set the parameter overwrite to True, so that if we run the script several times the current data will always be written to the file.

```
  #   Save table
  results.write(out_path + 'cmd.dat', format='ascii', overwrite=True)
```

Postprocessing

If you look at the images of the star clusters you will notice that the star clusters usually occupy only a part of the field of view. Mostly this area will be between 30% and 60% of the field of view. So we probably observe beside the star clusters a number of other stars, so called field stars, which actually do not belong to our star cluster. Generally there will also be some stars between us and the star cluster. Since these stars most likely did not form together with the star cluster, these stars will spoil our results concerning the age determination or make them more difficult to interpret.

> **Task:** Try to limit the selection of stars to the most probable star cluster members. You have two possibilities, which can be used alternatively or additively.
>
> 1. Limit the selection of stars to e.g. 10 arc minutes around the central coordinates of the star cluster.
> 2. Download the data from the Gaia archive (catalog ID: `I/350/gaiaedr3`) as demonstrated in the calibration above. From this data set, look in particular at the columns relating to the proper motion of the stars. Use this data to select the cluster members.
>
> Note: In any case, it is helpful to create starmaps (as described in the section "Checking the Calibration Stars") or similar plots that will help you to evaluate the results.

CMDs

Plot apparent CMD

For the creation of the CMD a *Python* script is available, in which only a few paths and a few further parameters have to be adjusted. This script also offers the possibility to plot isochrones. We will go into this in more detail below.

First you should copy the corresponding script `3_plot_cmd.py` from the directory `~/scripts/n2/` into the local working directory. Subsequently, the name of the star cluster (`nameOfStarcluster`) should be set and the path to the file saved above with the magnitudes should be added (`CMDFileName`).

The script `3_plot_cmd.py` can be called as follows

```
python 3_plot_cmd.py
```

This script creates a PDF file with the apparent CMD. The axes scaling is done automatically. Since this is not always ideal due to outliers, the plot range should be adjusted via the variables `x_Range_apparent` and `y_Range_apparent`. The quotation marks are simply to be replaced by the axis boundaries, such as `x_Range_apparent = [-0.5, 2]`.

```
###############################################################################
 ####            Configuration: modify the file in this section
####
###############################################################################

 #   Name of the star cluster
 nameOfStarcluster = "NGC7789"

 #   Name of CMD data file
 CMDFileName = "output/cmd.dat"

 ###
 #   Plot parameter
 #

 #   x_Range=[xRangeMin:xRangeMax] & y_Range=[yRangeMin:yRangeMax]
 #       -> x and y range to plot (change according to your data)
 #       -> The plot range is automatically adjusted, if range is set to ""
 #   Apparent CMD:
 x_Range_apparent=["",""]
 y_Range_apparent=["",""]

 #   Absolute CMD:
 x_Range_absolute=["",""]
 y_Range_absolute=["",""]
```

Note: In addition to these settings, there are a number of other configuration options, but we will not discuss them further at this point.

Reddening & absolute magnitudes

When comparing your apparent CMD to the literature, you will notice that the main sequence is likely to be shifted. This occurs due to the interstellar medium which is spread between the stars of our Galaxy. Like all other baryonic matter, it can be excited by light. It will reemit this energy usually at a longer wavelength. Therefore, this effect is called reddening (do not confuse it with redshift):

$(B-V)_{0} = (B-V) - E_{(B-V)}$

$V_{0} = V - A_{V}$

The reddening is mathematically described by the color excess $E_{(B-V)}$, the difference between the measured, uncorrected color $(B-V)$ (measured here) and the unreddened, "original" value $(B-V)_{0}$. The redding effects the magnitude, too. The correction term is A_{V}, which relates to $E_{(B-V)}$ by the reddening parameter R_V:

$A_{V} = R_V \cdot E_{(B-V)}$

In the solar neighborhood R_V usually is set to 3.1 (Seaton 1979). Find an appropriate value for $E_{(B-V)}$ for the line of sight to the observed cluster, i.e. in *VizieR* or in *Simbad* by means of the papers that are associated with your object. In any case, refer to the used catalog or paper in your

report! Apply this correction to your data and plot the CMD again.

Finally the apparent magnitudes should be converted into absolute magnitudes, so that later a comparison with isochrons is possible. For this, the corresponding distance modulus or the distance of the star cluster must be looked up in papers (publications) and the corresponding correction must be applied.

Absolute CMD plot

After the $E_{(B-V)}$ and the distance or distance modulus for the corresponding star cluster have been figured out, these can be entered at the corresponding variables in the script `3_plot_cmd.py`. `m_M` is the distance modulus. The remaining variables should be self-explanatory. If either `m_M` or `distance` is given, the script will create the absolute CMD as well as the apparent CMD. If it is necessary to adjust R_V this can also be done.

```
#   EB-V of the cluster
eB_V = 0.

#   R_V
RV = 3.1

#   Give either distance modulus of the cluster or the distance in kpc
m_M = '?'

distance = '?'
```

Note: In addition to these settings, there are a number of other configuration options, but we will not discuss them further at this point.

Plot isochrones

Some isochrones are already included in the OST library, although by no means all of them and some of them are incomplete. Therefore, especially if no suitable isochrones were found, you should search for further ones on your own. Stellar evolution calculations are performed by a number of working groups and researchers. The resulting isochrones are usually made available to the scientific community via web portals and can be downloaded from there.

Unfortunately, there is no uniform format for isochrones, which means that the script (`3_plot_cmd.py`) must be instructed to read these for each new "isochrone type" or each new "isochrone source". This is done using files in the so-called *YAML* format, which store the necessary configuration. For the isochrones contained in the OST library these configuration files can already be found in the script directory. An empty template file is also available there. In the script the selection of the respective "isochronous source" is done via the variable `isochrone_configuration_file`. Here the name or the path to the respective *YAML* file has to be entered.

[Do not display isochrones](#)

If no isochrones are to be displayed `isochrone_configuration_file` must be set to ""

For the [PARCEC isochrones](#) the configuration file looks like this:

```
---
#   PARCES isochrones (CMD 3.6)

#   Files
# isochrones:
'~/isochrone_database/parsec_iso/3p6/solar_0p2Gyr/iso_parsec_0p2Gyr.dat'
# isochrones:
'~/isochrone_database/parsec_iso/3p6/solar_0p5Gyr/iso_parsec_0p5Gyr.dat'
 isochrones:
'~/isochrone_database/parsec_iso/3p6/solar_1Gyr/iso_parsec_1Gyr.dat'

#   Type
isochrone_type: 'file'

#   Type of the filter used in CMD plots
#   Format:
#     'filter name':
#        - column type (single or color)
#        - ID of the filter if the column type is color, e.g., if the filter
is
#          R and the color is V-R, the filter ID would be 1. If column-type
is
#          single, the ID will be 0.
#        - name of the second filter, in the example above it would be V. If
#          column-type is single, the name can be set to '-'.
isochrone_column_type:
    'U':
      - 'single'
      - 0
      - '-'
    'B':
      - 'single'
      - 0
      - '-'
    'V':
      - 'single'
      - 0
      - '-'
    'R':
      - 'single'
      - 0
      - '-'
#   ID of the columns in the isochrone data file containing the magnitudes
#   and the age
isochrone_column:
```

```
    'U': 29
    'B': 30
    'V': 31
    'R': 32
    'AGE': 3
#   Keyword to identify a new isochrone
isochrone_keyword: '# Zini'

#   Logarithmic age
isochrone_log_age: true

#   Plot legend for isochrones?
isochrone_legend: true
...
```

isochrones points to the file with the isochrones. Here isochrone_type is set to file, which tells the script that all isochrones can be found in one file. An alternative is directory. In this case the script expects the isochrones to be found in individual files in a specific directory and that the variable isochrones points to that directory. With isochrone_column you can specify the desired column numbers. isochrone_column_type specifies whether the magnitudes are given as colors or as "single" magnitudes. See the format description above for more information. The basic options here are color and single. With isochrone_log_age you can specify whether the values in the age column are logarithmized or not. You can choose between True or False. If the isochrones are all in one file, the script needs a keyword to recognize when an isochrone ends and the next one begins. This can be specified with the variable isochrone_keyword. Finally, you can decide if you want to plot a legend for the isochrones. This is controlled by the variable isochrone_legend.

Tip: Usually there are isochrones from one source in different time resolutions and for different metallicities. These are then usually found in other files or folders. So it may be worthwhile to look in the database and adjust the entry for isochrones.

Note: Some additional information about the individual variables can be found directly in the *YAML* template.

Report

A usual report is to be handed in. See the general overview about the required structure and content here.

For this observation, the theoretical overview in the report should describe open and globular cluster with emphasis on the observed kind, and their differences to other accumulations and groups of stars. Explain Hertzsprung-Russell diagrams (HRD) and the color-magnitude diagrams (CMD) and the difference between them. *Shortly* describe the evolution of stars of different masses in the context of a HRD. Explain the concepts of isochrones and the turn-off point and how one estimates the age of a cluster using them.

In the methods section describe the observations and the data reduction, highlight points that deviate from general description in here and list all the parameters you set for the extraction. Further, include

all the plots of the data reduction in the report (a few in the text, most of them in the appendix). Also include any parameters for reddening, extinction, and distance that you adopt from the literature.

The results part presents the cluster CMDs and describes the observable features in it.

The analysis of the CMDs contains the estimation of the cluster age based on the turn-off point and an isochrone fit.

Finally, discuss your findings. Bring your results into a larger context and make a literature comparison when possible (i.e., for the cluster age). This also includes that you identify potential problems with the data, the data reduction, or the analysis (especially the isochrone fit) and possible solutions for them. Are their inconsistencies? Do you see specific and obvious features in the CMD you cannot explain, that do not match your expectations?

Note: *Due to the plots and images the report may not fit into an email appendix. You can upload your report to the University cloud system (BoxUP) or alternatively put it on the lab course computer and send us the path to it.*

Overview: Laboratory Courses