



The University of Manchester

PHYS30762 - OBJECT-ORIENTED PROGRAMMING IN C++

UNIVERSITY OF MANCHESTER

DEPARTMENT OF PHYSICS AND ASTRONOMY

Star Catalogue

Authors:

Andrea Sante (ID: 10305586)

Date: May 21, 2022

Abstract

Modern astronomical surveys are both an opportunity to unravel the mysteries of the universe, and a scientific challenge because of the large amount of data produced in each observation. Highly-efficient data analysis codes need to be implemented to extract all the information available. In this report I describe the implementation of an object-oriented program for the elaboration of astronomical observational data. Three classes describing galaxies, stars, and planets were considered. The functionality of the code is shown considering two data sets; one contains manually created galaxy and star objects, while the other consist of data from an open-source exoplanets catalogue¹. The program allows the user to collect data from different observations and perform data manipulation by finding, deleting and sorting specific objects. Functions for data visualisation and for calculating physical quantities, such as the apparent magnitude of stars, are also implemented. The program also allows the user to collect and save the derived properties of the objects into external files.

¹source: <http://exoplanet.eu/catalog/>

1 Introduction

In a white paper submitted to the Astro2020 Decadal Survey on Astronomy and Astrophysics [1], Siemiginowska et al. highlighted the crucial importance of statistics and informatics for analysing the huge amount of high-quality data observed by modern astronomical surveys. For instance, the Square Kilometre Array telescope is predicted to produce $0.5\text{-}1\text{ TBs}^{-1}$ of observational data [2] which, considering a standard observational length of six hours, will translate to a volume of data impossible to be elaborated manually. Object-oriented programming (OOP) represents an elegant and efficient solution to this challenge. By focusing on data types (objects) rather than functions or logic, the OOP approach uses polymorphism and inheritance to create software that are flexible and easily maintained even for large amount of data. In this report I describe how OOP could be employed to create a program that stores and elaborates observational data from astrophysical objects. The program is structured as a catalog and, as such, it groups together elements with common properties. The user is able to populate the catalog by reading-in experimental data from multiple files as well as inserting directly from the keyboard. Several data manipulation functions are implemented such as input validation, searching and deleting objects, and sorting containers based on the properties of its elements. The user can also perform astrophysics-related operations to gain insights on the data hold. For the sake of simplicity, only three classes (one for galaxies, stars and planets respectively) were implemented to give an example of the functionality of the code, however, the code can be extended easily to other types of astrophysical objects as an abstract base class acts as an interface to all the non-member functions. Details on the code design are provided in Section 2, whereas a practical application of the program is described in Section 3.

2 Code design and implementation

The code was implemented in order to be open to extensions and close for modifications. An abstract base class representing a general celestial object is used as an interface of derived classes describing specific astrophysical objects. The properties in common to all celestial objects such as the ID in the catalogue, name and equatorial coordinates are encapsulated in the abstract class as well as counters to the number of objects present in the scope. The specific properties of each derived class are also private and accessible only through class member functions. Functions whose behaviour differs for each astrophysical object are implemented as virtual functions of the base class and overridden in the derived classes. These involve mainly the creation and validation of new objects, the output of one object's information, and the calculation of the flux and distance to the object which are core properties of every astronomical system but that cannot always be directly measured through observations.

Each object can be created either manually inputting the parameters from the keyboard

Object Type	Name	RA (deg)	Dec (deg)	Hubble Class	Flux (Jy)	Diameter (Kpc)	Redshift
galaxy	NGC 4284	185.05	58.09	SBc	0.054	60.37	0.01402
...

Object Type	Name	RA (deg)	Dec (deg)	Flux (Jy)	Spectral Class	Mass (M_{\odot})	Radius (R_{\odot})	Proper Motion (mas/yr)	Transverse Velocity (km/s)
star	Altair	297.7	8.87	1853	A7	1.8	18	661	16.1
...

Object Type	Name	RA (deg)	Dec (deg)	Mass (M_J)	Radius (R_J)	Period (days)	a (AU)	e	i (deg)	Distance (pc)	Year of discovery
planet	K2-181 b	127.5	10.9	-	0.253	6.9	0.07	-	-	-	2018
...

Figure 1: Tables representing the order followed by the observational data of galaxies (top), stars (middle), and planets (bottom) stored in the external files that can be correctly read by the code.

or by reading it from data files with different extensions. In both cases, the object is dynamically initialised as a unique smart pointer in order to automate the process of memory allocation and deletion. When an object is created from the keyboard, the value of its parameters are validated immediately after the insertion; if a value is out of its physical bounds, an exception is thrown and caught allowing the user to re-insert the parameter until correctly defined. For astrophysical objects whose parameters are store in an external file, the construction requires the object type and file delimiter to be specified; the former is deducted from the file line containing the information of the object whereas the latter has to be inputted by the user, with the exception of csv files for which is automatically recognised. Although the delimiter can be any arbitrary key, the order of the object parameters in the file needs to be specifically as shown in Figure 1. The parameters of each object are read and saved into a stringstream object which is used together with the delimiter to assign the value to each corresponding member of the object class during the parameterised construction. The validation of the parameters is not part of the initialisation process and needs to be called explicitly by the user through the specific member function of the object of interest. All objects constructed from the data of a single file are collected into a vector of shared smart pointers; these are preferred to unique pointers as the ownership of each object can be shared with other functions in the main. Data from different files can be read during one run and the respective objects can be saved in the same vector of shared pointers.

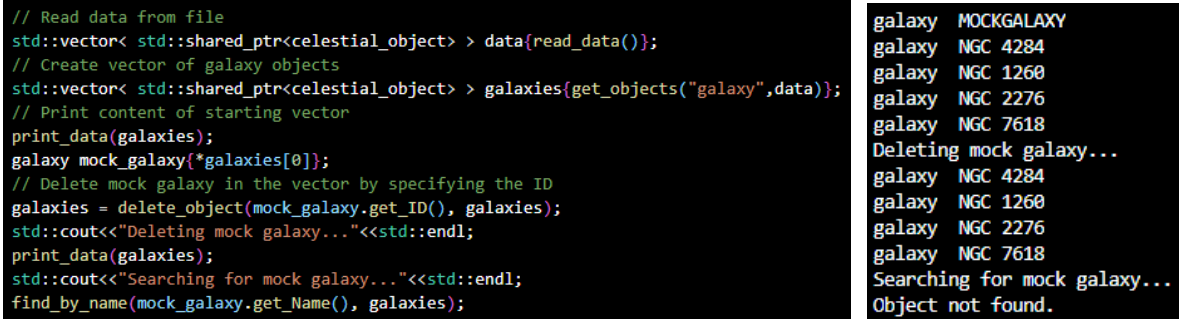
A set of member and non-member functions has been created to enable the user to manipulate the observational data from astrophysical objects. Based on the operations performed, these can be organised in three classes:

- **output and visualisation** functions, which allow the user to print information on the individual objects and on their containers in scope. Detailed information on a specific astrophysics objects can be displayed on screen and saved as a stringstream object using the member functions 'info()' and 'get.info()' respectively. The template function 'print_data()' allows the user to get an idea of the type and amount of data held by a vector of shared pointers of any type. As shown in Figure ??, it returns the total number of objects in scope breaking it down in the specific classes as well as a list of the type and name of the first ten objects, which could be expanded from the keyboard based on the user's desire. As a plotting package is not included in the Standard Library [3], the functions 'save_arrays_2D' and 'save_arrays_3D' are implemented to enable the user to save two or three vectors as columns of a csv file in order to be easily accessible with an external plotting software. These functions allow the user to choose the file name and check if the indicated file is already present in the working directory; if it is, the user can choose whether to overwrite or append the new content.
- **data manipulation** functions, which consent the user to organise and handle the data available in scope. Objects of the same type can be recognised in a vector of shared pointers and collected into a separate vector of the same type using the function 'get_objects'; this performs a deep copy of the objects of the original vector ensuring that no two pointers point towards the same object. It is also possible to find and delete objects in vector by specifying either the corresponding name or catalogue ID. The object of interest is searched using iterators and the function 'find' from the Standard Library; an exception is thrown when the desired object is not found. Both the find and delete member functions are coded as template functions to extend their applications to all classes of astrophysical objects. Template functions were also implemented to sort vectors by name (alphabetically), type (alphabetically), distance (closest to furthest), and flux (brightest to dimmest). As planets do not have flux information, the flux value is set to 0 by default.
- **astrphysics-related** functions, which elaborate the available data providing physical insights for the different objects. Hubble law ($v = cz = H_0 d$, assuming $H_0 = 68$ km/s/Mpc) [4] is used to explicitly calculate the distance (d) in Mpc to the observed galaxies from its redshift (z) allowing a comparison with other objects in the code. The distance from the observed stars is calculated using the relation between transverse velocity (v_T) and proper motion (μ): $v_T = 4.7 \times 10^{-3} d \mu$. The apparent (m) and absolute (M) magnitude of each star object can also be calculated. The former is derived from considering the apparent magnitude of Vega as reference: $m_{Vega} - m_* = -2.5 \log_{10}(F_{Vega}/F_*)$; the latter is calculated from the

distance modulus relation: $m - M = 5 \log_{10}(d/10pc)$. The derived parameters of star objects are used to produce the Hertzsprung-Russell (HR) diagram for the observed stellar population. Apart from the function specific to each derived class, functions to validate and convert the values parameters to SI units are implemented as virtual functions to ensure they are overridden adapting to each object.

3 Results

The code hereby reported is used to elaborate observational data from two different files: 'stars.txt', which comprises 17 objects between galaxies and stars; 'exoplanets.csv'², which contains information of 5036 exoplanets and planetary systems. At first, all objects are initialised and collected into a single vector of shared pointer. Then, the user is prompted to add a further object from the keyboard. After that, all the objects in scope are divided into three vectors based on the type. The galaxy vector is then used to test the operation of the validation and data manipulation functions, as it contains one object whose parameters were assigned incorrectly on purpose. After the validation of the galaxy objects, the mock galaxy is deleted from the vector using the delete by id function; the outcome of the operation is then checked calling the 'find_by_name' function as shown in Figure 2.



```
// Read data from file
std::vector< std::shared_ptr<celestial_object> > data{read_data()};
// Create vector of galaxy objects
std::vector< std::shared_ptr<celestial_object> > galaxies{get_objects("galaxy",data)};
// Print content of starting vector
print_data(galaxies);
galaxy mock_galaxy{*galaxies[0]};
// Delete mock galaxy in the vector by specifying the ID
galaxies = delete_object(mock_galaxy.get_ID(), galaxies);
std::cout<<"Deleting mock galaxy..."<<std::endl;
print_data(galaxies);
std::cout<<"Searching for mock galaxy..."<<std::endl;
find_by_name(mock_galaxy.get_Name(), galaxies);
```

```
galaxy  MOCKGALAXY
galaxy  NGC 4284
galaxy  NGC 1260
galaxy  NGC 2276
galaxy  NGC 7618
Deleting mock galaxy...
galaxy  NGC 4284
galaxy  NGC 1260
galaxy  NGC 2276
galaxy  NGC 7618
Searching for mock galaxy...
Object not found.
```

Figure 2: Input (left) and output (right) of the piece of code used for deleting and searching a specific galaxy object.

The star vector is considered for showing the kind of physical insights that could be provided by the code. The vector is first sorted by distance and flux to determine the closest and brightest star to Earth; the former is found to be Sirius, which has an apparent magnitude of -1.44 , while the latter is Alpha Centauri C, which is calculated to be 1.3 pc away. Figure 3 shows the input and output of sorting the star vector in these two ways. The absolute magnitude, luminosity, spectral class, and surface temperature of each star object is also calculated and used for plotting the two versions of the HR diagram shown in Figure 4.

²source: <http://exoplanet.eu/catalog/>

```

// Sort stars by flux
sort(stars.begin(), stars.end(), sort_by_flux<celestial_object>);
print_data(stars);
std::cout<<"The brightest star in the catalogue is "<<stars[0]->get_Name()<<"!"<<std::endl;

// Sort stars by distance
sort(stars.begin(), stars.end(), sort_by_distance<celestial_object>);
print_data(stars);
std::cout<<"The closest star in the catalogue is "<<stars[0]->get_Name()<<"!"<<std::endl;

```

```

star Sirius
star Canopus
star Beta Orionis
star Altair
star Zeta Puppis
star Mintaka
star Epsilon Pegasi
star Delta Virginis
star Gamma Piscium
star Lacaille 9352
... Do you want to load more data? ('y'/'n') n
The brightest star in the catalogue is Sirius!
star Alpha Centauri C
star Barnard's star
star Sirius
star Lacaille 9352
star Luyten's star
star Kapteyn's star
star Altair
star Gamma Piscium
star Delta Virginis
star Canopus
... Do you want to load more data? ('y'/'n') n
The closest star in the catalogue is Alpha Centauri C!

```

Figure 3: Input (left) and output (right) of the piece of code used for sorting the star objects by distance and brightness.

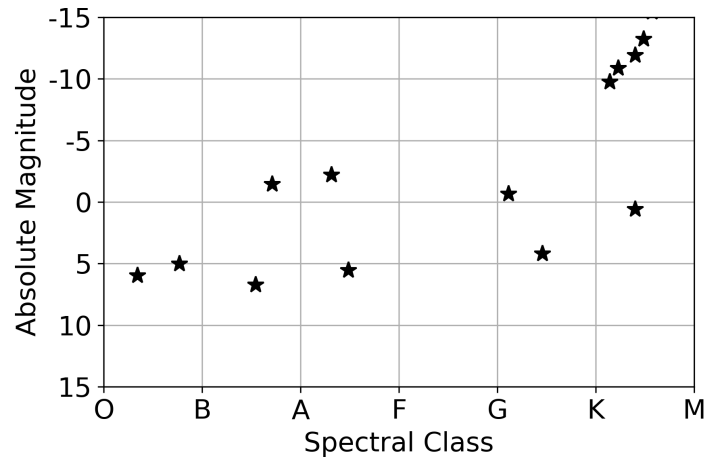


Figure 4: HR diagram produced calculating the absolute magnitude and spectral class of the star objects elaborated by the program.

The planet vector is considered to prove the possibility of extension of the code functions to files of larger size. The planet objects are first sorted by distance to determine the closest exoplanet from Earth; this is found to be Proxima Centauri b, which is distant 1.23 pc. Then, a random exoplanet is searched in the vector using the 'find_by_id' function' as shown in Figure 5.

Eventually, the equatorial coordinates of each exoplanets are saved and used to plot the exoplanet chart in Figure 6.

```
// Find a random exoplanet in the vector (ID: ASTRO+4352)
planet exoplanet(*find_by_id(4352, planets));
exoplanet.info();
```

```
-----
Planet: OGLE-2013-BLG-0132L b
-----
ID: ASTRO+4352
RA: 270
DEC: -28.4
Radius: 0
Mass(M_Jupiter): 0.29
Period (days): 0
Semi major axis (AU): 3.6
Eccentricity: 0
Inclination (deg): 0
Distance (pc): 3.9e+03
Year of discovery: 2017
-----
```

Figure 5: Input (left) and output (right) of the piece of code used for searching a planet in the data.

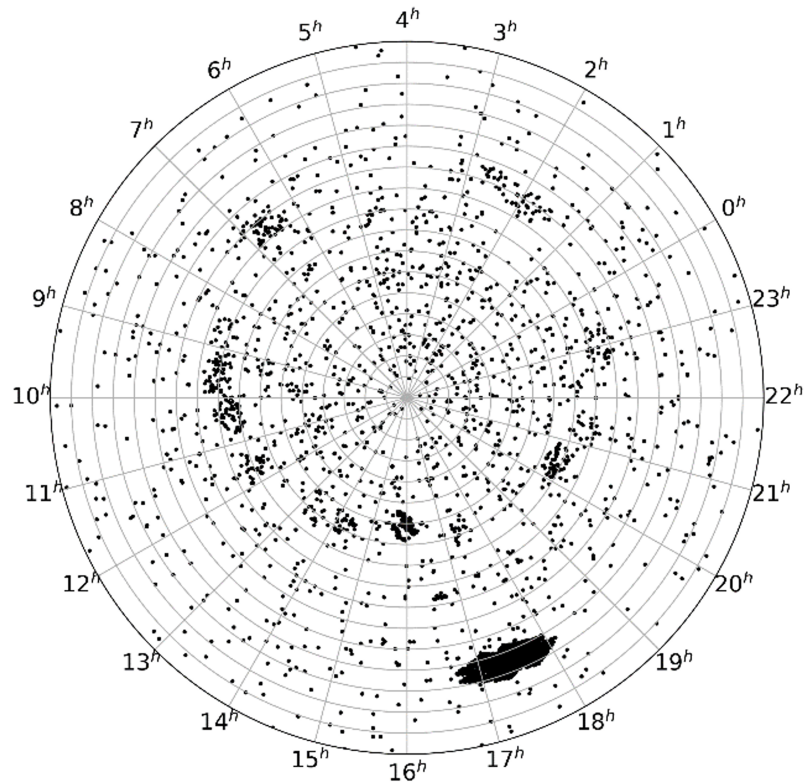


Figure 6: Position in the sky based on the equatorial coordinates of each of the exoplanets elaborated by the program.

4 Discussion

The program hereby reported successfully read and elaborates data from galaxies, stars, and planets. The operations performed are, however, generic and the user is recommended to implement functions specific to the insights interested in gaining from the analysis. Moreover, classes describing other astrophysical objects (such as nebulae, black holes, etc.) could be added to the code expanding its applications. In a similar way, objects sub-types, like pulsars and white dwarfs, could be implemented as derived classes of the existing classes making the data analysis more detailed. In terms of functionality, the code is limited by the fact that only data files with a precise structure and order can be read. Hence, the data entry pipeline could be made more general by allowing the user to store different type of properties for the same type of objects; this would probably mean to encapsulate the object parameters into an associative container (e.g a map) that can change depending on the type and amount of data entered. The implemented delete and find functions are particularly useful when working with large amount of data; however, these apply only to objects whose name or catalog ID are known. Hence, the functionality of find the delete could be extended to search and delete single or multiple objects satisfying certain boundary conditions on their parameters.

References

- [1] Aneta Siemiginowska et al. “The Next Decade of Astroinformatics and Astrostatistics”. In: 51.3, 355 (May 2019), p. 355. arXiv: 1903.06796 [astro-ph.IM].
- [2] A. M. M. Scaife. “Big telescope, big data: towards exascale with the Square Kilometre Array”. In: **Phil. Trans. R. Soc. A.** 378.2166 (2020).
- [3] Bjarne Stroustrup. **Programming: principles and practice using C++**. eng. Pearson Addison Wesley, 2014. ISBN: 9780133796735.
- [4] James Binney and Scott Tremaine. **Galactic Dynamics: Second Edition**. eng. REV - Revised, 2. Princeton: Princeton University Press, 2011. ISBN: 9780691130262.