Physics and Astronomy, School of Physical and Chemical Sciences,
University of Canterbury, Christchurch, New Zealand

Comparing Methods of Stellar Luminosity Function Construction Through Interpolation of Stellar Isochrones

Alex Goodenbour

UNIVERSITY OF CANTERBURY
Te Whare Wānanga o Waitaha
CHRISTCHURCH NEW ZEALAND

PHYS391 Project 2020

Supervisor: Dr. Chris Gordon

Date: 05/01/2020

Abstract

The construction of a stellar luminosity function is a problem that is often an important step in the analysis of astronomical data. This report describes and compares two methods for luminosity function construction. The more common Monte-Carlo method based on random sampling, and a more analytic approach involving a statistical change of variables. These methods are presented in the context of a consistency check to the *semi-analytic* method of luminosity function construction presented in the Galactic bulge investigation of Paterson et al. (2019) [1] but the strategies employed may be useful for the problem of stellar luminosity function construction in general. There is a particular emphasis on the computational details involved in implementing these strategies. We conclude that the preferred method of stellar luminosity construction is a *semi-analytic* method with confirmation using a Monte-Carlo method. The Monte-Carlo method alone is unnecessarily inefficient due to its need for heavy sampling.

Contents

1. Introduction

Investigations into Galactic structure often rely on a luminosity function, a distribution which describes the spread of luminosities of stars in a population, where luminosity refers to the electromagnetic power radiated by a star or related object. As the distribution of luminosities between different populations of stars remains relatively constant, a luminosity function can be calculated independently of the data being studied and acts as an assumption that is useful in finding the spatial distribution of a population of stars. Estimates of the distance to stars can be made through parallax or by the use of standard candles (For reference see Carroll & Ostlie (1996) [2]). The construction of a luminosity function is an important step in using standard candles to estimate stellar distances.

Previous investigations into Galactic structure, such as Wegg & Gerhard (2013) (WG13) [3] and Simion et al. (2017) (S17) [4], have made use of a Monte-Carlo method of luminosity function construction involving repeated sampling of a set of initial distributions. WG13, for example, fitted a parametric model to the distribution produced by the Monte-Carlo method while S17 used the GALAXIA tool for synthetic survey generation which also makes use of a Monte-Carlo method for luminosity function construction.

The investigation into structure in the Galactic bulge of Paterson et al. (2019) (P19) [1] took a more analytic approach involving a statistical change of variables. This approach shows promise as an alternative to the established Monte-Carlo method as the Monte-Carlo method involves heavy sampling of an arbitrary distribution and is therefore particularly inefficient while the *semi-analytic* approach is much more concise as it acts on expressions for the probability density functions rather than on samples from these distributions.

This report describes these two approaches to luminosity function construction including interpolation of isochrone tables with a particular emphasis on details of computation. These methods are presented in the context of a consistency check of the *semi-analytic* approach used in P19 but the procedures used may be useful for the construction and checking of stellar luminosity functions in general. Section 2 describes the process of interpolating from an isochrone table while Sections 3 and 4 describe the Monte-Carlo and *semi-analytic* methods of luminosity function construction respectively. Finally, Section 5 contains a comparison of these two methods in the context of Galactic structure.

1.1 Isochrones

For the purposes of this report, the term isochrone refers to a table of values that defines a general relationship between the metallicities, initial masses, and absolute magnitudes of an arbitrary population of stars at a given age. Note that metallicity is a quantity that describes the abundance of elements in a star heavier than hydrogen or helium (see Equation 1.1) and absolute magnitude is a measure of the luminosity (or radiated electromagnetic power) of a star on an inverse logarithmic scale. For reference, see Carroll & Ostlie (1996) [2].

$$[\mathrm{Fe/H}] = \log_{10}\left(\frac{N_{\mathrm{Fe}}}{N_{\mathrm{H}}}\right)_{\mathrm{star}} - \log_{10}\left(\frac{N_{\mathrm{Fe}}}{N_{\mathrm{H}}}\right)_{\mathrm{sun}} \tag{1.1}$$

The isochrone values provide a method for estimating an absolute magnitude given an initial mass and metallicity of a star, or conversely, estimating probable initial masses given a star's absolute magnitude and metallicity. As the isochrone values come in the form of discrete points, interpolation methods are needed. This process and details of computation are described in depth in Section 2.

1.2 Luminosity Functions

In this report, a luminosity function is defined as a distribution that describes the number density of stars (or related objects) at a given absolute magnitude, and so constructing a luminosity function is the process of finding the distribution of absolute magnitudes of some arbitrary population of stars. The luminosity and thus the absolute magnitude of a star can be estimated given its initial mass, its age, and its metallicity.

If it is assumed that only initial mass m, metallicity z, and age influence a star's absolute magnitude M_{K_s}, then, once the isochrone values are available, construction of a luminosity function becomes a statistical problem. Given two random variables m and z with known distributions, a constant age, and a third variable $M_{K_s} = \theta(m, z)$ given by the isochrone relation denoted θ, how does one determine the distribution of the third variable? There are two distinct approaches to solving this problem. The first is a Monte-Carlo method in which N samples of each of the two known distributions are taken and substituted into the function θ to get N samples of M_{K_s} distributed according to the resulting luminosity function. The second approach, designated the *semi-analytic* method, uses a statistical change of variables to find the distribution of the third variable M_{K_s} given expressions for the probability density functions of the initial variables.

It is important to break down the assumption that initial mass and metallicity can be used to estimate a star's luminosity. If a star is treated as a perfect black-body, then its luminosity will be determined solely by its radius and temperature. Initial mass is highly correlated with radius while temperature is heavily dependent on the chemical composition of a star which can be described using metallicity. So, if we assume black-body radiation, a star's luminosity can be estimated using these two quantities. Of course, stars are not perfect black-bodies and many other factors influence their luminosity but for the purpose of stellar luminosity function construction, this is a reasonable assumption.

2. Isochrone Interpolation

To construct the luminosity function as in P19, mass-absolute magnitude-metallicity relations from the PARSEC+COLIBRI isochrones [5] were used for an age of 10 Gyr using 39 metallicity bins spaced linearly in the range $-2.279 < [Fe/H] < 0.198$. More details on the specifications of the isochrone data can be found in Appendix A.

Isochrone values come in the form of a table of discrete data points each containing an initial mass, a K-band magnitude, and a metallicity among several other variables. The data is tabulated at constant values of metallicity and so the isochrone table can be thought of as a set of 39 mass-absolute magnitude relations each corresponding to a value of metallicity. Figure 2.1 provides a visualisation of this data. Flags in the data allow the grouping of points corresponding to different stages of stellar evolution. It is advisable to perform the following methods for luminosity function construction separately on each of these branches of the isochrone as there are discontinuities between stages that may result in artefacts in the final luminosity function.

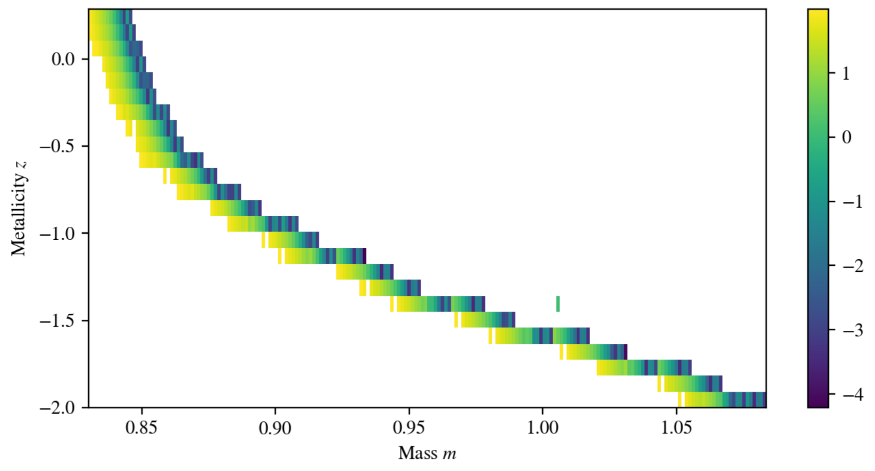

Figure 2.1: A visualisation of the PARSEC+COLIBRI [5] isochrone data.

2.1 Cleaning the Data

It is very important to visualise the isochrone table before implementing any interpolation device as any outliers in the data will have an outsized influence on the final luminosity function. This is because a spline is used to interpolate between the discrete data points and an outlier would extend this spline far beyond its plausible range leading to interpolation where the isochrone relation should not be defined. Therefore, any obvious outliers should be removed to prevent this. The assumption that these outliers can be removed is reasonable because they represent distinct branches of the isochrone data and so there is no valid reason to create a spline that extends to these points.

The presence of outliers in the isochrone data and their effect on the resulting luminosity function can lead to major variation in results using this luminosity function. Although the fix is extremely simple, this highlights the importance of properly visualising any data files because a single point can have a far outsized effect on any analysis undertaken on a set of data.

2.2 Constructing the Interpolation Device

The phrase interpolation device is used to refer to a programmatic function that takes an initial mass and metallicity, and returns an absolute magnitude by interpolating from the isochrone table.

Given a mass and metallicity, the constant metallicity cross-section of the isochrone table most closely corresponding to the given metallicity is selected. Note that a small portion of the drawn metallicities will be outside the metallicity range of the isochrone. These drawn values will therefore correspond to the constant metallicity cross-sections at the extremes of the isochrone. Selecting a cross-section will give a set of data points corresponding to a function M_{K_s} (magnitude) with respect to m (mass). Each value of mass corresponds to exactly one value of magnitude. Therefore, a linear spline can be constructed between these data points using SciPy's `UnivariateSpline` method [6] or an analogue and an interpolated value at the given mass can be returned. If a drawn mass is outside the range of the spline then no value is interpolated. Figure 2.2 presents an example cross-section of the isochrone table.

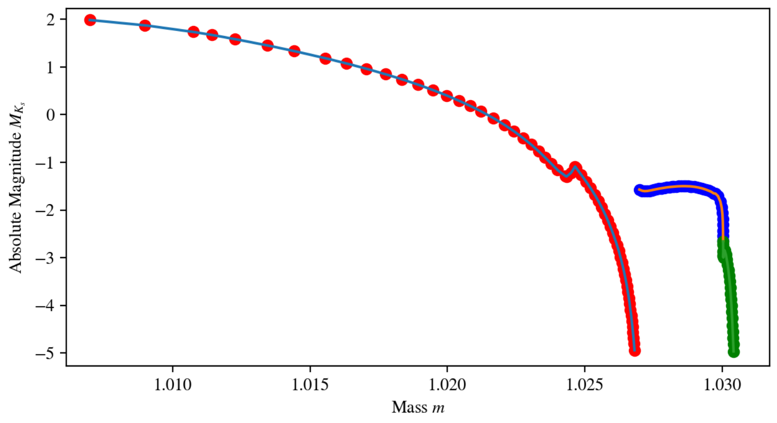

Figure 2.2: An example cross-section of the isochrone table for a constant metallicity of $z = -0.0315$. The corresponding splines have been overlayed. The colours of points correspond to the different stages of stellar evolution. Red corresponds to the Red Giant stage while blue and green correspond to the Red Clump and Asymptotic Giant branches respectively.

Note that it is much more efficient to generate these splines once and store the spline objects than it is to generate the spline on command each time the interpolation device is called.

2.3 Constructing the Inverse Interpolation Device

Rather than take a mass and metallicity and return a magnitude, the inverse interpolation device should take a magnitude and metallicity and return one or more masses by interpolating from the isochrone table.

Constructing the inverse interpolation device follows a similar process but is slightly more involved due to the fact that in the absolute magnitude-mass relations for constant values of metallicity, one magnitude may correspond to multiple values of mass. Given a magnitude and metallicity, the same process of choosing the cross-section of the isochrone table corresponding to the metallicity closest to the given metallicity is used. This gives a set of points now corresponding to an absolute magnitude-mass relation (the inverse of the mass-absolute magnitude relation from before).

To interpolate from these points, multiple splines must be generated corresponding to one-to-one sections of the absolute magnitude-mass relations. This can be done by finding the local maxima and minima with respect to magnitude of these points. One can then take the sets of points bounded by these maxima and minima and separate them into sub-arrays with the each of the local extrema being placed into the both neighbouring sub-arrays to avoid gaps in the splines. These sub-arrays are then ordered with respect to magnitude and splines generated for each sub-array using SciPy's `UnivariateSpline` method, taking note of the ranges on which each of these splines is defined. For each metallicity, this gives a list of splines with their ranges. Therefore, to interpolate for a given magnitude and metallicity, the closest metallicity cross-section is taken and for each spline corresponding to this cross-section that also has the given magnitude within its range, an interpolated value is returned. This returns 0 or more interpolated values corresponding to a given magnitude and metallicity. Figure 2.3 presents an example cross-section of the isochrone data with inverse splines overlayed.

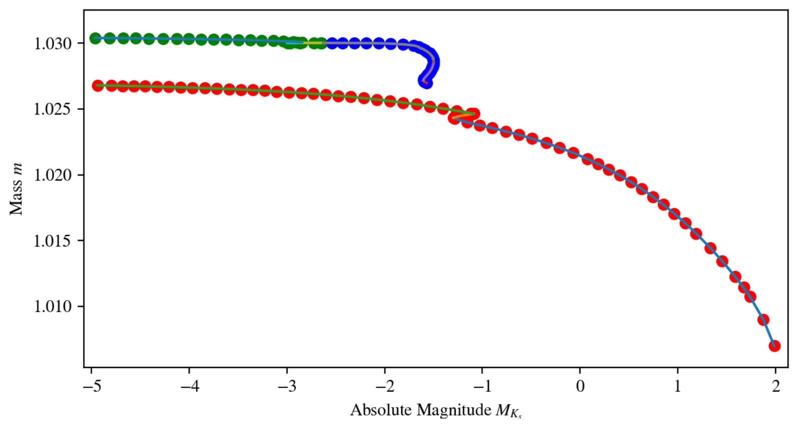

Figure 2.3: An example cross-section of the isochrone table for a constant metallicity of $z = -0.0315$. This time the corresponding inverse splines have been overlayed. The colours of points correspond to the different stages of stellar evolution.

3. Monte-Carlo Method

The method most commonly used for producing a luminosity function is a Monte-Carlo simulation. A given population of stars can be modelled by taking into account only the initial mass and metallicity of each star. Our chosen system to check the methods is the Galactic bulge as modelled by P19. For this population of stars, it can be assumed that metallicities will be distributed normally with $\mu = 0.0$ and $\sigma = 0.4$ as in Zoccali et al. (2017) [7], and initial masses will be distributed according to a log-normal Chabrier (2003) [8] initial mass function (IMF) (See equation 3.1). The absolute magnitude of each star given its initial mass and metallicity is then interpolated from the isochrone table. N samples of the metallicity distribution function and initial mass function can be calculated to gain N mass-metallicity pairs. These pairs can subsequently be substituted into the isochrone interpolation device described in Section 2.2 to generate a number of samples of the final luminosity function. Less than N samples of magnitude will be produced because many mass-metallicity pairs will not correspond to any value of absolute magnitude.

$$\xi(m) = 0.158 \times \exp\left\{ - \frac{(\log(m) - \log(0.079))^2}{(2 \times 0.69^2)} \right\} \tag{3.1}$$

With these samples of the magnitude distribution, a histogram can be constructed which will give a discrete approximation to the magnitude probability distribution function. i.e. the luminosity function. The bin centres of this histogram can subsequently be convolved with a gaussian of standard deviation 0.05 representing a typical uncertainty for absolute magnitude. Convolution with a Gaussian acts to smooth the luminosity function according to the measurement uncertainty in absolute magnitude. This process is described in more detail in Section 3.2. This Monte-Carlo method can be performed separately using branches of the isochrone corresponding to the different stages of stellar evolution. The result of this process can be seen in Figure 3.1.

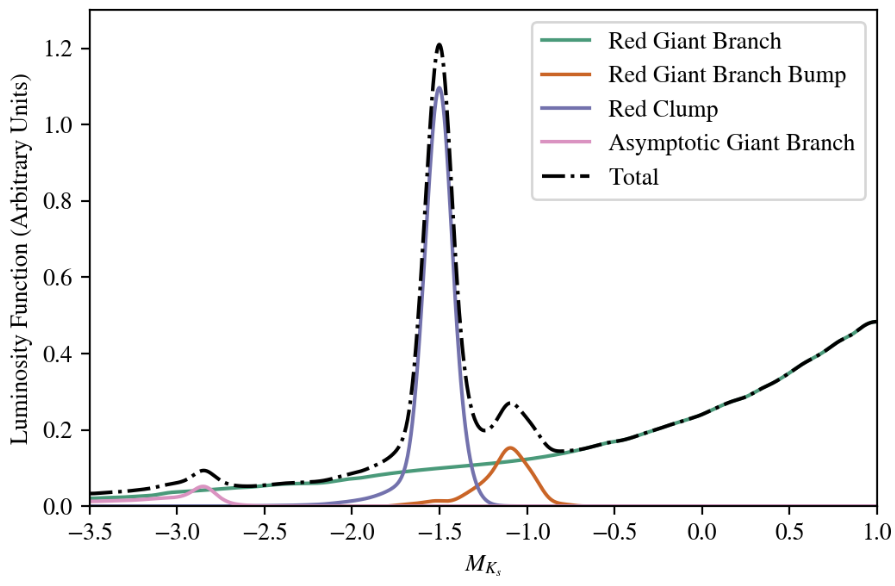

Figure 3.1: Monte-Carlo simulation of the luminosity function of a 10 Gyr population using PARSEC+COLIBRI isochrones, a Chabrier (2003) IMF and a Zoccali (2008) metallicity distribution function. The luminosity function has been convolved with a Gaussian of $\sigma = 0.05$ and normalised on the range of the plot.

3.1 Sampling an Arbitrary Distribution

Any Monte-Carlo simulation relies heavily on obtaining a large number of samples of some arbitrary distribution. For the metallicity distribution function, this is trivial as the normal distribution is extremely well studied and any programming environment will have a heavily optimised function for sampling a normal distribution with a given mean and standard deviation. For the log-normal initial mass function, the process is not so simple. Many programming environments will have an optimised function for sampling a log-normal distribution, however, the Chabrier (2003) IMF, although log-normal, is not of the same specific form as the PDFs used in these functions.

SciPy also provides a method for sampling an arbitrary distribution. However, this method is very inefficient for high sample sizes. The time taken to generate the number of samples needed to construct a reasonable luminosity function is not practical. Therefore, a different method must be used.

3.1.1 Inverse Transform Sampling

Any arbitrary distribution can be approximately sampled by discretising its probability density function. The first step is to use SciPy's `quad` function or an analogue in another programming environment to calculate the discretised cumulative distribution function (CDF) of a log-normal distribution. This is done by creating a finely sampled linear grid over the range of masses to be sampled (between 1000 and $10,000$ samples will usually suffice). Note that this range of masses can be chosen by looking at the isochrone table to see what the possible range of masses is. With a PARSEC+COLIBRI isochrone this range is around $0.8M_\odot < m < 1.1M_\odot$. Only this portion of the distribution must be sampled because all samples outside a certain mass range will be discarded as they do not correspond to any magnitude in the isochrone. Now, for a given mass m, the CDF is given by the integral from $-\infty$ or in this case 0 (as you cannot have a negative mass) to m of the log-normal distribution computed using SciPy's `quad` function. This will give the discretised CDF of the log-normal distribution (see equation 3.2).

$$F_m(m) = \int_0^m f_m(t)\,dt \tag{3.2}$$

where $F_m(m)$ is the CDF of the random variable m and $f_m(m)$ is the PDF of m.

The next step is to create an interpolation object of the inverse CDF (quantile function). A cubic spline is suitable and gives a very good approximation to the analytic CDF. With SciPy, this is as simple as substituting the m and $F_m(m)$ arrays into the `UnivariateSpline` library function. This returns an interpolation object that takes a given value of F_m and returns the corresponding value of m. This process assumes that the CDF of the distribution is injective and so has an inverse. In the case of a log-normal distribution, this property is satisfied. The initial PDF must also be normalised on the range of sampling.

N samples are then taken from a uniform distribution using NumPy's `random.rand` function or an analogue in another programming environment. Each of these samples is then substituted into the inverse CDF to gain a set of N mass samples which will be distributed according to a log-normal distribution.

3.2 Gaussian Smoothing

Also known as a Weierstrass transform, Gaussian smoothing is the process of convolving a signal with a Gaussian distribution of given standard deviation. This has the effect of smoothing the signal with the depth of smoothing corresponding to the standard deviation of the Gaussian. This is the same process used by image editing software to compute a Gaussian blur.

3.2.1 The Discrete Convolution Operation

Working with a discretised luminosity function rather than an analytic expression, one must perform a discrete convolution. The convolution of two functions f and g is denoted $f * g$ and given by Equation 3.3.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)\,d\tau \tag{3.3}$$

From this follows the definition of discrete convolution. Given two arrays a and b, the convolved array $a * b$ can be found. (Note that square brackets indicate array subscript notation.)

$$(a * b)[n] = \sum_{m=-\infty}^{\infty} a[m]b[n - m] \tag{3.4}$$

The NumPy library for Python provides a function `convolve` which performs this operation given two arrays to be convolved. For further convenience, one might use SciPy's function `ndimage.gaussian_filter` which gives the same result without the need to assemble a discretised gaussian distribution.

Using this method to convolve the bin centres from the histogram of magnitudes gives a final luminosity function accounting for measurement uncertainty in absolute magnitude.

4. Semi-Analytic Method

Another method used to construct a luminosity function, and the method presented in P19, is the *semi-analytic* method. Rather than sampling from the metallicity and initial mass distribution functions, the PDFs of the initial distributions can be manipulated to find the resulting distribution of magnitudes. As the isochrone interpolation device is not an analytic expression, an analytic expression for the luminosity function cannot be obtained and so, like the Monte-Carlo method, this method will produce an arbitrarily accurate discrete approximation.

The statistical problem faced can be summarised as follows. There exist two random variables m and z distributed according to log-normal and normal distributions respectively. There also exists some function $M_{K_s} = f(m,z)$ namely the isochrone interpolation device. As the distributions of m and z are known and it is known how they are combined to form a third variable M_{K_s}, one can determine how this variable is distributed. Since the initial variables m and z are independent, this can be done by first finding the distribution of M_{K_s} as a function of both magnitude and metallicity and then integrating over metallicity to find the luminosity function as a function of magnitude only. As in the Monte-Carlo method, this method can be performed separately using branches of the isochrone corresponding to different stages of stellar evolution.

The luminosity function as a function of both magnitude and metallicity can be given by a statistical change of variables.

$$\phi(M_{K_s}, z) = \sum_i \xi(\theta^{-1}(M_{K_s}, z)) \left| \frac{d\theta^{-1}(M_{K_s}, z)}{dM_{K_s}} \right| \tag{4.1}$$

where M_{K_s} is magnitude, z refers to metallicity, ξ is the initial mass function and θ^{-1} is the inverse isochrone interpolation device as defined and implemented in Section 2.3. The summation refers to summing over all the possible solutions to the inverse isochrone relation θ^{-1}.

The luminosity function can then be found as a function of magnitude alone by integrating over all values of metallicity to find the expected value with respect to metallicity.

$$\Phi(M_{K_s}) = \int_{-\infty}^{\infty} \phi(M_{K_s}, z) g(z)\, dz \tag{4.2}$$

where $g(z)$ refers to the metallicity distribution function. Note that for practical purposes it is important to integrate over some large range approximating $(-\infty, \infty)$ such as $[-100, 100]$ rather than integrating just over the range of metallicity in the isochrone table. This is because the metallicity distribution functions extends slightly outside the range of the isochrone table and so more weight will be given to metallicities on the ends of the table.

In a previous iteration of this method, it was assumed that one should only integrate over the range of metallicities defined in the isochrone table. This does not account for the greater influence of metallicities outside this range and led to a final luminosity function that looked very similar to the result of the Monte-Carlo method and could have been overlooked. This highlighted the importance of challenging any assumptions made. It also builds a case for using multiple methods of luminosity function construction to confirm the final distribution.

4.1 Implementation of Semi-Analytic Method

Substituting Equation 4.1 into Equation 4.2 gives an expression for the final luminosity function. This expression can be evaluated on a linearly spaced array of points to find a discrete approximation to the luminosity function. This is done by substituting each point in this array of linearly spaced magnitudes into this expression using the previously implemented functions for interpolating the isochrone and its inverse in Equation 4.1 and using SciPy's `simps` function or similar for numerical integration in Equation 4.2.

Finally, Gaussian smoothing of standard deviation 0.05 representing a typical measurement uncertainty in absolute magnitude can be applied using the technique described in Section 3.2.1 to recover a final luminosity function using this *semi-analytic* method. This result is presented in Figure 4.1.

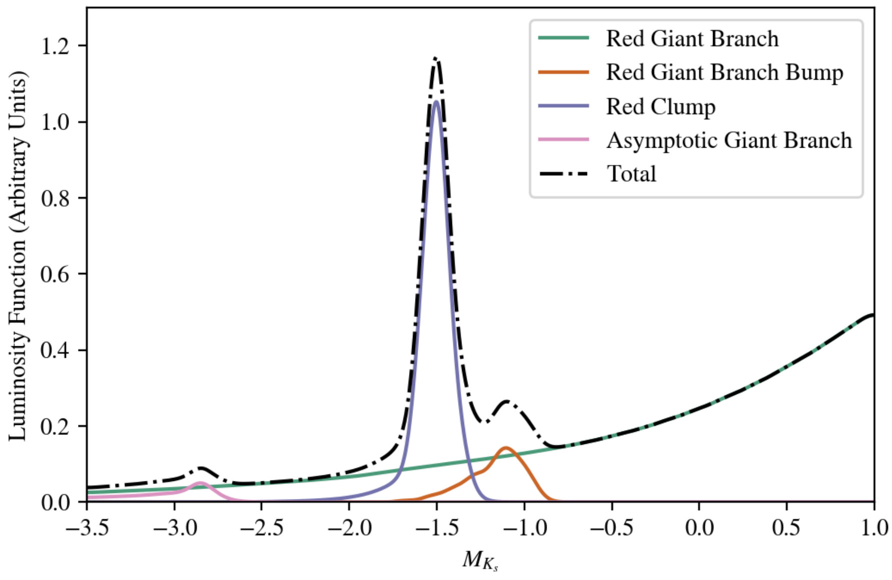

Figure 4.1: Luminosity function of a 10 Gyr population using PARSEC+COLIBRI isochrones, a Chabrier (2003) IMF and a Zoccali (2008) metallicity distribution function. The luminosity function has been convolved with a gaussian of $\sigma = 0.05$ and normalised on the range of the plot.

5. Method Comparison

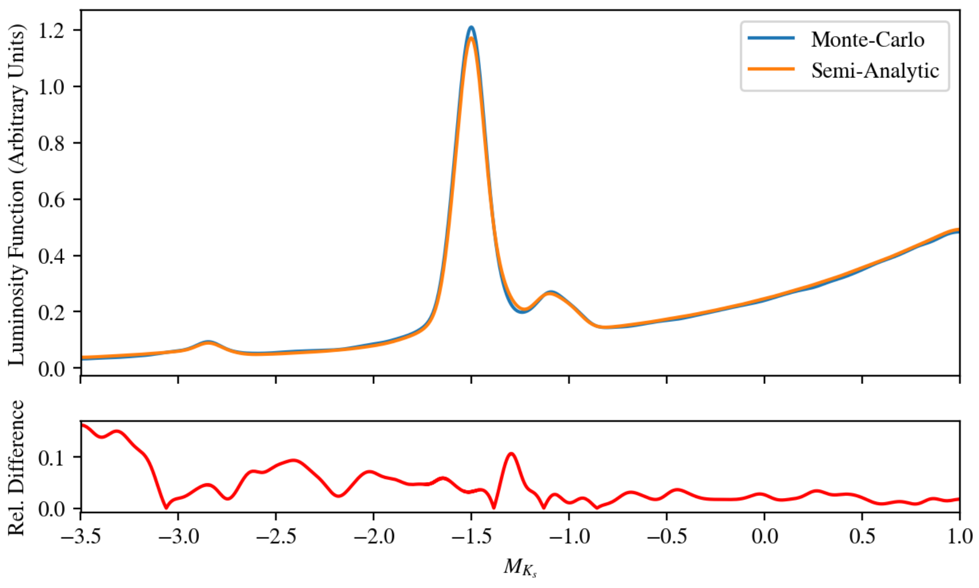

Figure 5.1: Comparison of Monte-Carlo and *semi-analytic* methods of luminosity function construction.

Comparing the output of these two starkly different methods, it is clear that the results are remarkably similar (See Figure 5.1). However, this is to be expected given that these two methods are mathematically identical. The small differences that are present will be due to the sampling error in the Monte-Carlo method and discretisation error in both the Monte-Carlo and *semi-analytic* methods.

Comparing these outputs to the luminosity function constructed in P19 (See Figure 2 of P19), there is almost complete correspondence with a minor shift in the Red Giant Branch Bump. This discrepancy is likely due to a difference in the parameters of the isochrone values used. The parameters used to generate the isochrone values in this report can be found in Appendix A.

Previous investigations into Galactic structure, such as WG13 and S17, have opted for a Monte-Carlo approach to luminosity function construction, presumably due to its simplicity. The fundamental difference between this method and the *semi-analytic* approach is that the Monte-Carlo method works on samples of the initial and final distributions, whereas the *semi-analytic* method works directly on the probability density functions of the initial and final distributions. It is for this reason that it is preferable to opt for a *semi-analytic* method of luminosity construction. Monte-Carlo methods, although easy to implement, can become very computationally expensive due to the need to generate a large number of samples of some distribution and perform calculations on each of these samples. Furthermore, Figure 2.1 shows that the majority of these samples will be thrown away as they do not correspond to any absolute magnitude. The use case of Monte-Carlo methods tends to be problems that do not have another viable solution. In this case, the *semi-analytic* method is a viable alternative that does not involve heavy sampling of a distribution.

This does not mean that a Monte-Carlo method should be completely written off as a luminosity function construction method. Due to its ease of implementation, it lends itself to acting as a confirmation for a *semi-analytic* or alternative method. As an interpolation device will have already been constructed for the main method, implementing a Monte-Carlo check is as simple as sampling the two initial distributions and substituting them into the interpolator. Because the *semi-analytic* method is more involved to implement, having a check with a much simpler implementation can provide a level of confidence in the implementation of the main method.

6. Conclusions

Two viable methods for the construction of a stellar luminosity function were laid out, the standard Monte-Carlo method relying on repeated sampling of initial distributions of mass and metallicity, and a *semi-analytic* method, as presented in P19, that works on the basis of a statistical change of variables. Both methods rely on interpolation of an isochrone table, a procedure discussed in Section 2.

Comparing these two methods, the *semi-analytic* method was most favourable largely because it is much more efficient as it does not require the generation of a large number of samples. However, because of the simplicity in implementation of the Monte-Carlo method, it can easily serve as a confirmation of another method. Especially if an isochrone interpolator has already been constructed.

This report may be useful to anyone constructing stellar luminosity functions from isochrones by acting as a general overview of the options available for luminosity function construction and providing a run-through of how these options may be implemented.

Acknowledgements

I would like to thank Dr. Chris Gordon of the School of Physical and Chemical Sciences, University of Canterbury for supervising this project. I am also thankful to Dylan Paterson for promptly answering any questions I had about his work.

Bibliography

[1] D. Paterson, B. Coleman, and C. Gordon, "*Further evidence that the Milky Way bar has undergone a buckling phase*", arXiv preprint arXiv:1911.04716 (2019).

[2] B. W. Carroll, D. A. Ostlie, and M. Friedlander, "*An Introduction to Modern Astrophysics*", Physics Today **50**, 63–72 (1997).

[3] C. Wegg and O. Gerhard, "*Mapping the three-dimensional density of the Galactic bulge with VVV red clump stars*", Monthly Notices of the Royal Astronomical Society **435**, 1874–1887 (2013).

[4] I. T. Simion, V. Belokurov, M. Irwin, et al., "*A parametric description of the 3D structure of the Galactic bar/bulge using the VVV survey*", Monthly Notices of the Royal Astronomical Society **471**, 4323–4344 (2017).

[5] P. Marigo, L. Girardi, A. Bressan, et al., "*A new generation of PARSEC-COLIBRI stellar isochrones including the TP-AGB phase*", The Astrophysical Journal **835**, 77 (2017).

[6] E. Jones, T. Oliphant, P. Peterson, et al., *SciPy: Open source scientific tools for Python*, 2001–.

[7] M. Zoccali, V. Hill, A. Lecureur, et al., "*The metal content of bulge field stars from FLAMES-GIRAFFE spectra-I. Stellar parameters and iron abundances*", Astronomy & Astrophysics **486**, 177–189 (2008).

[8] G. Chabrier, "*Galactic stellar and substellar initial mass function*", Publications of the Astronomical Society of the Pacific **115**, 763 (2003).

A. Isochrone Parameters

The PARSEC+COLIBRI [5] isochrones used to construct the luminosity function were generated using the web form at `http://stev.oapd.inaf.it/cgi-bin/cmd_3.3` with parameters as described in this header.

```
File generated by CMD 3.3 (http://stev.oapd.inaf.it/cmd)
on Thu Jan  2 01:21:46 CET 2020

Isochrones based on PARSEC release v1.2S +  COLIBRI S_35
Thermal pulse cycles included
On RGB, assumed Reimers mass loss with efficiency eta=0.2
Photometric system: VISTA ZYJHK_s (Vegamag)
Using YBC version of bolometric corrections as in Chen et al. (2019)
O-rich circumstellar dpmod60alox40 dust from Groenewegen (2006)
C-rich circumstellar AMCSIC15 dust from Groenewegen (2006)
IMF: Chabrier (2001) lognormal
```

B. Python Code

The code used to produce these results may be found at `https://github.com/alg107/PHYS391-Project`. Instructions on how to run this code can be found in the file README.md. Only 3 of 7 scripts are included in this appendix for brevity.

```python
#!/usr/bin/env python

"""
MonteCarlo.py: Constructs a luminosity function using
            the Monte-Carlo method
"""

import numpy as np
import matplotlib.pyplot as plt
import scipy.stats as ss
import lib.IMF as IMF
from progressbar import progressbar as pb
import pandas as pd
from lib.Iso import Isochrone


# The number of samples of metallicity and IMF
N = int(1e7)


# Metallicity

metallicity_mean = 0.0
metallicity_std = 0.4
metallicity_bins = 39

# IMF

IMF_bins = 100

# 2: Sampling
# 2.1: Metallicity

# Sampling metallicities from a normal distribution
sampled_metallicities = np.random.normal(metallicity_mean, metallicity_std, N)

```

```python
plt.figure()
plt.title("Metallicity Distribution")
plt.hist(sampled_metallicities, 100, density=True, histtype='step')
X = np.linspace(-2*metallicity_std, 2*metallicity_std, 10000)
plt.plot(X, ss.norm.pdf(X, metallicity_mean, metallicity_std))

print("Metallicity Done")

# 2.2: IMF

# Sampling masses from lognormal IMF
sampled_IMF = IMF.IMF_sample(N)

plt.figure()
plt.title("IMF Distribution")
IMF_seq = np.linspace(0.8, 1.1, IMF_bins)
plt.hist(sampled_IMF, IMF_seq, density=True, histtype='step')
X = np.linspace(0.8, 1.1, 1000)
plt.plot(X, IMF.chabrier2_V(X))

print("Masses Done")


# Plots the result
def plot_samples(sampls, bins, title):
    plt.figure()

    # Plotting the LF histogram
    final_bins = bins
    bins_seq = np.linspace(-3.5, 1.0, final_bins)
    plt.hist(sampls, bins_seq)

    plt.title(title)
    plt.xlabel("Magnitude")
    plt.ylabel("Count")

total_sampled_mags = []

for t in [1,2,3]:
    print("Interpolating Branch", t)
    iso = Isochrone(typs=[t])

    # This is the array in which we will build up our final histogram
    sampled_mags = []
    for i, m in pb(enumerate(sampled_IMF), max_value=N):
        val = iso.interpolate(m, sampled_metallicities[i])
        if not np.isnan(val):
            sampled_mags.append(val)
    sampled_mags = np.array(sampled_mags)

    np.save("Results/MCSamples/type"+str(t)+"N"+str(N), sampled_mags)
    print("Efficiency: ", len(sampled_mags)*100/N, "%")

    plot_samples(sampled_mags, 75, "Type "+str(t)+" LF")
    total_sampled_mags.extend(sampled_mags)

plot_samples(total_sampled_mags, 75, "Total LF")
plt.show()
```

```python
#!/usr/bin/env python

"""
SALF.py: Constructs a luminosity function using
            the semi-analytic method
"""

import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from scipy.interpolate import interp1d, InterpolatedUnivariateSpline, splev
from scipy.integrate import simps, trapz
from scipy.ndimage.filters import gaussian_filter1d
from progressbar import progressbar as pb
from lib.Iso import Isochrone
from lib.IMF import chabrier2 as chabrier

iso = Isochrone()

# Normal distribution
def norm(x, m, s):
    return (1/np.sqrt(2*np.pi*s**2))*np.exp(-((x-m)**2/(2*s**2)))

# Mass distr. fn.
def MDF(z):
    return norm(z, 0.0, 0.4)

def phi(M, z, typs):
    ms, derivs = iso.inverse_interpolate(M, z, typs)
    phi_c = 0
    for i, m in enumerate(ms):
        phi_c += chabrier(m)*np.abs(derivs[i])
    return phi_c

RES = 1000
x = np.linspace(-3.5, 1.0, RES)

def Phi(M, typ):
    #zs = iso.zs
    zs = np.linspace(-100.0, 100.0, 1000)
    phis = np.array([phi(M, z, [typ]) for z in zs])
    MDFs = np.array([MDF(z) for z in zs])
    ys = phis*MDFs
    I = trapz(ys, zs)
    return I


plt.figure()
plt.xlabel("Magnitude")
plt.ylabel("Luminosity Function (Arbitrary Units)")

for i in pb([1,2,3], redirect_stdout=True):
    # Plotting the results
    ys = np.array([Phi(M, i) for M in x])
    np.save("Results/SALF/xs_t"+str(i), x)
    np.save("Results/SALF/ys_t"+str(i), ys)
    plt.plot(x, ys)
    print("Done Branch ", i)


plt.show()
```

```python
#!/usr/bin/env python

"""
Iso.py: An object oriented wrap for the Isochrone table.
        provides methods for interpolation and visualisation
        among other things.
"""

import numpy as np
import matplotlib.pyplot as plt
import matplotlib
import pandas as pd
from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import interp1d, UnivariateSpline, NearestNDInterpolator
from progressbar import ProgressBar as pb
from scipy.stats import binned_statistic_2d

# Classifies stage based on statement in paper 1
def classify_stage(val):
    # 1: Red Giant
    # 2: RC
    # 3: Asymptotic Giant
    if val<=3:
        return 1
    elif val <= 6:
        return 2
    else:
        return 3
classify_stageV = np.vectorize(classify_stage)

# Gets a colour given a number from 1-3
def colour_from_type(typ):
    if typ==1:
        return "red"
    elif typ==2:
        return "blue"
    elif typ==3:
        return "green"
    else:
        return "yellow"

def find_nearest(array, value):
    array = np.asarray(array)
    idx = (np.abs(array - value)).argmin()
    return array[idx]

def find_neighbour(array, value):
    # Array must be sorted and unique
    # Returns index
    array = np.asarray(array)
    dists = (np.abs(array-value))
    dist = dists.min()
    idx = dists.argmin()
    first = array[idx]
    if idx+1 == len(array):
        return -1
    elif idx == 0:
        return 0
    elif dists[idx+1] < dists[idx-1]:
        return idx
    else:
        return idx-1


# Chabrier IMF PDF
def chabrier(m):
    if m <= 0:
        return 0
    else:
        return (0.158/(np.log(10)*m))*np.exp(-((np.log10(m)-np.log10(0.079))**2/(2*(

# Normal distribution
def norm(x, m, s):
    return (1/np.sqrt(2*np.pi*s**2))*np.exp(-((x-m)**2/(2*s**2)))

# Mass distr. fn.
def MDF(z):
    return norm(z, 0.0, 0.4)

# jiggles points very very slightly just to get around
# the spline restriction of not allowing points with equal
# x-values
def jiggle_pnts(pnts):
    return np.array([np.random.random()*0.00000001+i for i in pnts])

# The same but for one point
def jiggle_pnt(pnt):
    return np.random.random()*0.00000001+pnt
jiggle_pntV = np.vectorize(jiggle_pnt)


# Object oriented wrap for the isochrone table
class Isochrone():
    #def __init__(self, binx=750, biny=25, fname="iso.db"):
    def __init__(self, binx=200, biny=25, fname="data/iso_big.db", typs=[1,2,3]):
        # Taking the useful stuff from the isochrone table
        iso_table = np.loadtxt(fname)
        MH = iso_table[:,1]
        masses = iso_table[:,3]
        Kmag = iso_table[:,32]
        types = iso_table[:,9]
        df_arr = np.column_stack((MH, masses, Kmag, classify_stageV(types)))
        df = pd.DataFrame(df_arr, columns=["MH", "masses", "Kmag", "types"])

        df = df[df['Kmag'].between(-5.0, 2.0)]
        df['Kmag'] = jiggle_pntV(df['Kmag'])
        df['masses'] = jiggle_pntV(df['masses'])

        self.typs = typs
        self.df = df

        self.df_ret = binned_statistic_2d(df['masses'], df['MH'], df['Kmag'], bins=[binx,
        self.gen_splines()
        self.gen_inverse_splines()

    def gen_splines(self):
        zs = np.sort(np.unique(self.df.MH))
        self.zs = zs
        spls = {}
        for z in zs:
            spls[z] = []
            df_local = self.df[self.df['MH']==z]
            df_local = df_local.drop_duplicates(subset=['masses'])
            df_local = df_local.sort_values(by="masses")
            for typ in self.typs:
                df2 = df_local[df_local.types==typ]

                mmin = df2.masses.min()
                mmax = df2.masses.max()
                spl = UnivariateSpline(df2.masses, df2.Kmag, k=1, s=0)

                spls[z].append((spl, mmin, mmax))
        self.spl_dict = spls
        return spls

    def gen_inverse_splines(self):
        zs = np.sort(np.unique(self.df.MH))
        self.zs = zs
        spls = {}
        for z in zs:
            spls[z] = {}
```

```python
            for t in self.typs:
                spls[z][t] = []
                df_local = self.df[(self.df['MH']==z)&(self.df['types']==t)]

                pnts = np.column_stack((df_local.Kmag, df_local.masses))
                pnts = pnts[pnts[:,1].argsort()]
                split_idx = []
                for i, pnt in enumerate(pnts[1:-1]):
                    if pnts[i][0] < pnts[i-1][0] and pnts[i][0] < pnts[i+1][0]:
                        # print("Relative Minima")
                        split_idx.append(i)
                    elif pnts[i][0] > pnts[i-1][0] and pnts[i][0] > pnts[i+1][0]:
                        # print("Relative Maxima")
                        split_idx.append(i)
                split_pnts = np.split(pnts, split_idx)

                # This loop just puts extrema in both sides' splines
                for i, j in enumerate(split_pnts[:-1]):
                    split_pnts[i] = np.vstack((split_pnts[i], split_pnts[i+1][0]))

                # Delete sections with lengths less than two
                # so spline doesn't fail
                del_idxs = []
                for i, pnt in enumerate(split_pnts):
                    if len(pnt)<=1:
                        del_idxs.append(i)
                split_pnts = np.delete(split_pnts, del_idxs, axis=0)

                split_pnts = np.array(split_pnts)


                for i, sec in enumerate(split_pnts):
                    sec = sec[sec[:,0].argsort()]
                    mmin = sec[:,0].min()
                    mmax = sec[:,0].max()
                    spl = UnivariateSpline(sec[:,0], sec[:,1], k=1, s=0)
                    spls[z][t].append((spl, mmin, mmax))

        self.inv_spl_dict = spls
        return spls



    def plot(self, df=None):

        if df is None:
            df = self.df
        # Plotting these 3 vars in a box just
        # to get a feel for the data
        fig = plt.figure()
        ax = Axes3D(fig)
        plt.title("Isochrone mass-magnitude")

        for typ in self.typs:
           filt = df[df["types"]==typ]
           ax.scatter(filt["masses"], filt["MH"], filt["Kmag"], marker=".",
                   color=colour_from_type(typ))
        x = 0.9
        y = -0.7
        ax.set_xlabel("Mass ($m$)")
        ax.set_ylabel("Metallicity ($z$)")
        ax.set_zlabel("Magnitude ($M_{K_s}$)")
        return plt

    def colour_plot(self):
        plt.figure()
        df = self.df
        plot_arr = self.df_ret
        extent = [plot_arr.x_edge[0],
                  plot_arr.x_edge[-1],
                  plot_arr.y_edge[0], plot_arr.y_edge[-1]]
        plt.imshow(plot_arr.statistic.T, aspect='auto',  extent=extent)
        cbar = plt.colorbar()
        cbar.ax.set_ylabel("Absolute Magnitude $M_{K_s}$")
        plt.xlabel("Mass $m$")
        plt.ylabel("Metallicity $z$")

    def interpolate(self, m, z):
        closest_z = find_nearest(self.zs, z)
        for i, typ in enumerate(self.typs):
            spl, mmin, mmax = self.spl_dict[closest_z][i]
            if m < mmin or m > mmax:
                continue
            else:
                return spl(m)
        return np.nan
    interpolateV = np.vectorize(interpolate)

```

```python
    def inverse_interpolate(self, Kmag, z, typs=[1,2,3]):
        results = []
        dresults = []
        closest_z = find_nearest(self.zs, z)
        for typ in typs:
            spls = self.inv_spl_dict[closest_z][typ]
            for spl, mmin, mmax in spls:
                if Kmag >= mmin and Kmag <= mmax:
                    val = float(spl(Kmag))
                    # Not really needed but this just
                    # stops extrema from being counted twice
                    if not val in results:
                        results.append(val)
                        dresults.append(float(spl.derivative()(Kmag)))
        return results, dresults

    def plot_slice(self, z, w_spl=True):
        closest_z = find_nearest(self.zs, z)
        local_df = self.df[self.df["MH"]==closest_z]
        pl = plt.figure()
        for i, typ in enumerate(self.typs):
            df2 = local_df[local_df["types"]==typ]
            plt.scatter(df2["masses"], df2["Kmag"], color=colour_from_type(typ))
            spl, mmin, mmax = self.spl_dict[closest_z][i]
            x = np.linspace(mmin, mmax, 100000)
            y = spl(x)
            plt.plot(x,y)
        return pl

    def plot_inverse_slice(self, z, w_spl=True):
        closest_z = find_nearest(self.zs, z)
        local_df = self.df[self.df["MH"]==closest_z]
        pl = plt.figure()
        for typ in self.typs:
            df2 = local_df[local_df["types"]==typ]
            plt.scatter(df2["Kmag"], df2["masses"], color=colour_from_type(typ))
            for spl, mmin, mmax in self.inv_spl_dict[closest_z][typ]:
                x = np.linspace(mmin, mmax, 100000)
                y = spl(x)
                plt.plot(x,y)

        return pl


```

```python
if __name__=="__main__":

    matplotlib.rcParams['mathtext.fontset'] = 'stix'
    matplotlib.rcParams['font.family'] = 'STIXGeneral'

    iso = Isochrone()
    iso.plot()

    iso.colour_plot()
    plt.tight_layout()
    print("Colour plot done")

    plt.show()
```