

STELLAR ATMOSPHERES AND SPECTROSCOPY: INTERPRETING THREE DIMENSIONAL HYDRODYNAMIC SIMULATIONS WITH PYTHON

Guy Leckenby, Martin Asplund, Remo Collet

Research School of Astronomy and Astrophysics,
The Australian National University, Canberra, ACT 2611, Australia

ABSTRACT

The spectra produced by astronomical objects provides us with a vast amount of information on the chemical composition and structure of the object. For stars, the most luminous astronomical objects, these spectra are produced in the upper atmosphere and hence accurate modelling of the photosphere and chromosphere are required to draw correct conclusions from the spectra observed. Recent advances in the understanding of spectral lines in stars and an increase in computational capacity have lead to much more complex three dimensional hydrodynamic time-dependent models which have demonstrated excellent agreement with observation. This paper demonstrates the elegance of using Python to provide an interpretive interface for these models as well as providing reasons for why astrophysical computing as a field is moving towards Python as the language of choice. This is demonstrated by the introduction of a rudimentary Python module for interfacing with the 3D models accompanied by comparative plotting examples of a variety of profiles. Suggestions are also provided for further improvements and expansions for the module for wider astronomical use in stellar modelling.

I. INTRODUCTION

As light is the main source of information we receive from outside the solar system, it is essential to understand the spectra of astronomical objects to understand the universe. In particular the spectra of stars are crucial as they are the most common luminous object in the universe and the light we receive from them informs our understanding of their own internal structures but also the other visible objects in the universe through photon interactions. The spectra studied from stars are absorption spectra produced in the stars atmosphere. The absorption is with respect to the continuum spectrum of the star where the continuum is generated by the super heated gas in the photosphere which produces a smooth intensity distribution [1]. Gas that exists in the chromosphere then absorbs from the continuum to produce the observed absorption spectra. Hence a full understanding of the processes occurring inside stellar atmospheres is crucial for understanding the structures of the universe.

A good example of the importance of understanding stellar atmospheres is the importance of hydrogen. Hydrogen is the most abundant element in stellar atmospheres and hence it is a good characterising element for determining the properties of the star via spectroscopy. Due to its simple atomic structure, hydrogen is much more sensitive to the atmospheric properties than heavier elements [2] and hence the wings and core structure of the absorption lines relate directly to atmospheric conditions [3]. The Balmer series in particular is commonly used to study stellar atmospheres as it occupies the visible region taking advantage of the opacity of the atmosphere for ground observations. As the lower levels of the hydrogen atom are much more frequently populated, significant opacity is produced at the core of the absorption lines whilst interactions with charged species and other hydrogen atoms result in extended wings. Furthermore, as hydrogen forms the main continuum opacity source, changes in hydrogen abundance barely affect the line's strengths and as gravity and heavy element abundance produce weak effects on the absorption line,

surface temperature perturbation has the most dramatic effect in determining the shape of the Balmer lines [2]. Hence, the Balmer lines are a very powerful surface temperature indicators and as they are reddening independent they may, in principle, provide an accuracy on the order of 50 K [4].

Understanding of the stellar atmosphere does not stop with hydrogen however. The entire chemical composition of stars, in particular our sun, is derived from absorption spectra. Chemical abundance data also informs our understanding of the internal workings of stars and thus comparison with observed spectra provides a fundamental comparison with helioseismologic models [5]. Understanding stellar compositions is also essential for determining the spread of elements in other astronomical objects as each stellar system is believed to have been generated from the same nebula.

For stellar spectra to be used for these purposes however, powerfully predictive models are needed. Due to the complexity of a stellar atmosphere, a vast collection of factors affect the observed wing shape. In particular, the absorption coefficient is affected by normal absorption (natural broadening), the velocity of the absorbing particle (thermal Doppler and microturbulent broadening), interactions with charged particles (linear Stark broadening), and interactions with neutral particles (van der Waals broadening) [6]. Early models such as those by Vidal et. al. [7] have now been superseded by more powerful and comprehensive calculations.

With recent advances in computational capacity, the classical approximations used in previous stellar atmospheric models have been challenged. In particular, the assumptions of a static 1D atmosphere with a mixing length treatment of convection and the assumption of local thermal equilibrium (LTE) throughout the atmosphere have attracted significant scrutiny [9]. Whilst no model is yet able to fully remove both of these assumptions simultaneously, attempts have been made to remove each individually. For the geometrical assumption, there has been significant development in full three dimensional hydrodynamical time-dependent simulations of convection

(referred to hereafter as 3D models) for application to the solar photosphere [4][5]. These predictions from these 3D models are in excellent agreement with the observed spectra for the $H\alpha$ and $H\beta$ Balmer lines which Fuhrmann et. al. [8] claim are the most powerful benchmarks for atmospheric models. Furthermore, the 3D models correctly predict the lifetimes and sizes of solar granulation observed at high spatial resolution[9]. However the radical prediction of significantly lowered oxygen, carbon and nitrogen abundances by the 3D models have led to considerable disagreement with helioseismological models of the stellar interior. Whilst this presents concerning implications, the 3D models have significantly more predictive power over the stellar spectrum than the previous 1D LTE models [10].

II. INTERACTING WITH THE 3D MODELS

As with all complex computational models, an accessible interface is needed for the results to be of scientific value. The remainder of this paper will examine an interface for the hydrodynamic 3D models, in particular using the Python programming language. The 3D modelling considered here is the 3D, radiative, hydrodynamical, and conservative STAGGER-CODE[11]. This model employs a rectangular section of the solar atmosphere with, for example, dimensions of $6 \times 6 \times 3.8 Mm^3$ which has a Cartesian resolution of 240^3 data points. The horizontal grid is equally placed whilst the vertical grid is non-linear to accommodate for higher resolution over high temperature gradients. The top and bottom edges of the simulation are open, transmitting boundaries to allow for appropriate handling of free convection flows to ensure realistic treatment[9].

As temperature, abundance, etc. are not directly observable but rather inferred from spectral absorbance lines, the 3D models are used to predictively match observed spectra to determine the composition and temperature flows. In particular, the 3D models produce a variety of spectra that can be directly compared to observation. The models employed in this paper produce spectra for the total flux profile, the perpendicular spatially averaged intensity profile, spatially resolved intensity profiles, angularly resolved intensity profiles and a continuum intensity (or flux) associated with each profile. This data is stored in NetCDF file storage (a universal data storage format run by Unidata) and hence program routines need to be developed to extract the data and present it in a useful format.

Currently, the Research School of Astronomy and Astrophysics (referred hereafter as RSAA) uses the IDL language, a programming language popular in the field of astronomy for its vectorised and numerical data analysis. IDL is very good at astronomical data processing due to it being an array focused language as well as also having mature and extensive astronomical libraries such as `idlutils`. However student access to IDL is at best difficult and otherwise the subscription fee is quite substantial which can present problems during collaborations. Furthermore, as the language was first constructed in 1977 and is based primarily on FORTRAN and C, it has some frustrating syntax requirements that can be difficult to learn and requires extensive knowledge to produce elegant and efficient code. It is also a rather narrow in its

applications and not suited to generalised programming due to its focus on numerical analysis only.

There is a movement building throughout the astronomical community to move to Python as it deals with many of these problems. Firstly it is open source and hence accessible to the entire global community which in turns generates an extensive user base. Python was designed recently with its inception during 1989 but having major re-releases in 2000 and 2008 and it was designed specifically for scientific use. It is also designed to take full advantage of its interpretive nature and is a highly abstracted language which is more intuitive for the programmer and allows for easier debugging. Hence basic operations can often be done more easily, more elegantly and cheaper in Python than in IDL. It also has much more flexibility in non-numerical aspects of programming like databases, web page generation, web services, text processing, process control, etc.[12] The disadvantages of Python mainly stem from the youth of the language and as a result many processes well established in IDL are in current construction or not existent at all in Python. However these problems are being rectified as the Python community matures.

As a result of the problems with IDL, there is clearly a need for routines in Python to interpret, compare and display the results produced by the 3D models. Presented below is some preliminary Python code which achieves this function.

III. DESIGNING THE INTERFACE

The data in the 3D models is stored in NetCDF format which is a data storage format that is independent of any software, it just requires an interface module in the language you are working with. The Python NetCDF module uses an object oriented interface as Python is primarily an object oriented language. To achieve the desired interface for interacting with the 3D models, the data needed to be extracted and then manipulated with a set of functions. As the interface requires multiple actions on different profiles, an object oriented approach centred around each profile was ideal as it not only runs cleaner and more simply but also reduces redundancy. The code that was produced with this in mind the generates a basic interface module for the 3D models is displayed in Appendix A. In particular, this module is name `profPlot` (hereafter referred to as the module).

The module was constructed with a set of desired outcomes in mind. In particular, it had to,

- calculate the temporal average of any profile;
- continuum normalise any profile;
- calculate the equivalent width for any profile; and
- plot any combination of these profiles for comparison.

The module was required to do this for every variable in the data which vary in dimensions and shape. This is ideal for an object oriented approach.

The module was constructed primarily around the idea of taking the NetCDF variables and converting them to workable objects to perform functions on. There are three object classes defined in the code, a normal variable, a spatial variable and an angular variable. Normal variables encompass simple multi dimensional arrays with the dimensions of time steps, g

factor values, and finally the intensity profile¹. The variable object include the perpendicular intensity profile, `prof_int`, and the flux profile, `prof_flux`. The angular and spatial variables, `prof_angle` and `prof_xy` respectively, have all the dimensions of the parent class, ‘variable’, but include angular and spatial resolution respectively and were hence implemented as child classes.

Using the functions attributed to each class, options can be executed such as temporal averaging, continuum normalisations and equivalent width calculations. Furthermore, the primary function of the module is to produce comparison plots to examine the effects of convection (and other elements of the 3D modelling process) on the various profiles. Hence the plot function is of crucial construction. The plot function takes in a list of profiles to be plotted and returns a graph of the comparison. As each variable is an object, the construction of the plotting list can often be completed by a simple for loop which provides extensive flexibility. Part of this flexibility is that other functions outside the module can still be plotted with the desired profiles² by simply adding a `pylab.plot(xVals, yVals)` call before the `profPlot.plot` call. The module also supports alternating the x value that the profiles are plotted against. For example, another common scale is to use the Doppler velocity units with respect to the base wavelength. Any wavelength can be expressed in terms of a reference wavelength, λ_0 and the appropriate Doppler shift by

$$\lambda - \lambda_0 = \frac{v}{c} \lambda_0.$$

Hence, a velocity scale can be produced that varies linearly with respect to wavelength. This use of Doppler notation is particularly useful when considering line profiles in a moving medium such as a stellar atmosphere because it allows the direct translation of physical velocities into the Doppler broadening presented in the absorption spectra.

The data is presented in large multi dimensional arrays that include up to five different variables. The intensity profiles can be selected by inputting the appropriate indexing during initialisation of the variable. Otherwise, if the indexing is not included, the module resorts to pre-set defaults. For the time step, the module automatically take the temporal average, and for the g factor value, the angles and xy coordinates, the middle index is selected by default.

The module evidently has limitations to its capabilities and for a variety of reasons, users may want to access the core data. To this purpose, each variable object opens the NetCDF file in read only format. Hence the raw NetCDF dataset can be accessed through any `self.rootgrp` call using the NetCDF module syntax. This allows for descriptions of the shapes of variables, access to the raw data and attributes of the NetCDF file. Further information on the Python’s NetCDF module can be found through the netCDF4 Module manual by Whitaker[13].

IV. CAPABILITIES OF THE INTERFACE

As the module was designed with flexibility in mind, not all the capabilities of the module will be demonstrated here but rather the most useful selection. The file used is a solar spectrum centred on the Fe(I) emission line at 5041.7559 Å (hereafter referred to as FeI-504). This allows for full demonstrations of not only convective influences but abundance changes.

One of the requirements was to produce temporal averaging and continuum normalisation for each profile provided as these are how spectral lines are conventionally presented in astrophysics. Both calculations involve basic multi dimensional array mechanics which is managed in python through numpy. To demonstrate these capabilities, the flux profile will be used. The flux of a star is of special distinction in astronomy as it differs from the intensity which is a common source of confusion. The intensity represents the physical number of photons being received by the observer per unit of time. The flux however is the intensity integrated over any closed surface enclosing the star (the surface can be any shape, the same number of photons will pass through). Hence it is the total amount of light being emitted from the star over its entire sphere at any one.

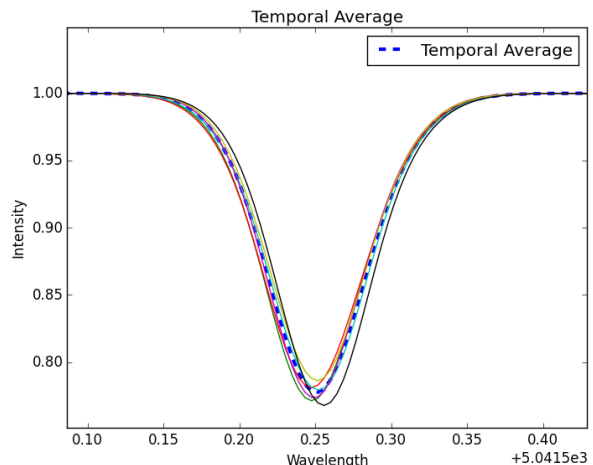


Fig. 1: The temporal average and the individual time steps for the flux profile.

As is evident from Figure 1, the temporal changes are minimal but still important. The thick dotted line represents the temporal average. Note that for each profile produced, if a time step is not specified, the temporal average is taken by default. Note also that the continuum norm is also calculated practically by default (there is no function to produce a list of the actual intensity values, this can only be accessed directly through NetCDF commands).

The 3D models also provide variation in abundance in terms of the gf values. Varying the gf values allows a simulation of variation in the abundance as the equivalent width is proportional to the abundance by gf . However during the calculations, the gf term is varied instead of abundance as it is more convenient (less computationally heavy) to adjust the gf value then to recompute the quantum effects of abundance and then combine it with the changes in abundance. The 3D

¹These concepts will be more thoroughly explored in Section IV.

²An example is seen in Section IV.

models often produce an output of $\pm n0.2$ of the standard $\log g$ value to provide a variety of abundances to consider (for our data, $n = 1$ producing 3 gf values as in Figure 2). Changing the abundance of an element, especially one with a relatively low abundance in the stellar composition like Iron (0.3% of total atoms), drastically changes the strength of the spectral absorbance line.

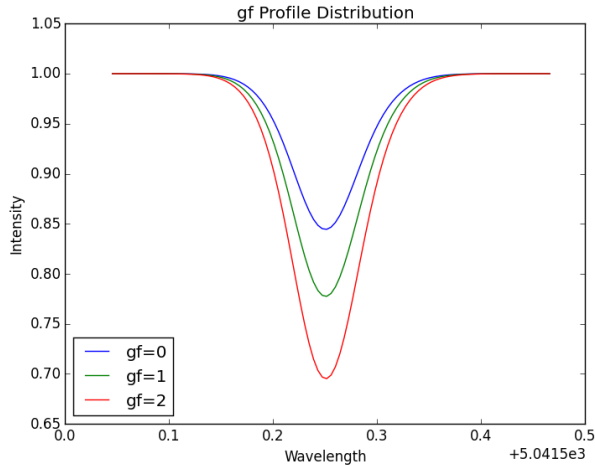


Fig. 2: The 3 different gf values for the flux profile.

This is evident in Figure 2 as those with higher gf values, corresponding to higher abundance, produce stronger absorption lines. Note that by default the middle gf value is chosen.

The equivalent width is a quantitative measure of the strength of spectral lines and is a commonly used piece of data to initially assess the spectra. The equivalent width itself corresponds to the normalised area of the absorption line which is equivalent to a rectangle (of normalised height 1) with width equal to the area of the absorption line. The equivalent width for any given profile can be calculated using the `self.equivWidth()` function included in the module. Computing the equivalent width for the time averaged flux profile of the solar FeI-504 emission line gives $W_\alpha = 1.8812 \times 10^{-2} \text{ \AA}$, for example.

The 3D models maintain angular resolution of the spectrum through the variable `prof_angle`. This takes the intensity profile at a variety polar (μ) and azimuthal (ϕ) angles (in spherical polar coordinates). The profile does not change drastically throughout the azimuthal distribution. However the polar angles produce considerable variation. Note that each angle has its own observed continuum level. To provide the comparison in Figure 3, the intensity profiles were normalised against the perpendicular continuum.

As is evident from Figure 3, increasing μ values (corresponding to greater angles away from the perpendicular) leads to a decrease in the intensity. This is intuitive as for any given section of the solar atmosphere, it is expected that the intensity will obey a cosine relation over the polar angles which is demonstrated in Figure 3. That is lots of light will be emitted perpendicularly and zero will be emitted at perfect right angles the modelled patch. Note that each profile plotted in Figure 3 is an average over its respective ϕ values using the `self.phiAvg()` function.

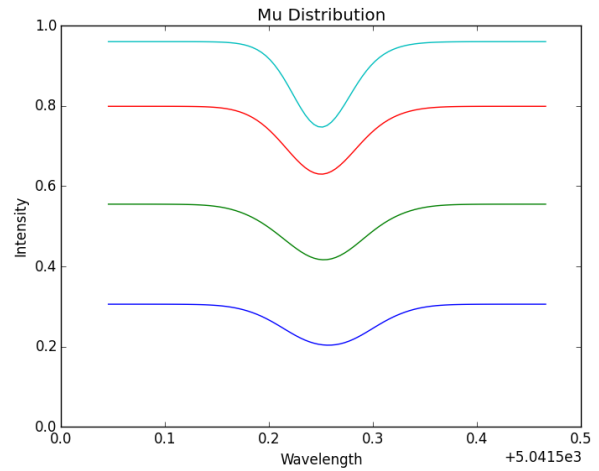


Fig. 3: The distribution of μ intensity profiles.

Furthermore, the 3D models maintain spatial resolution of the spectrum through the variable `prof_xy`. This takes the perpendicular intensity over the modelling surface grid. The solar data set used for the examples provides a Cartesian grid of 120×120 data points. This spatial resolution shows how convective flows and atmospheric granulation affect the produced absorbance lines through thermal Doppler and microturbulent broadening.

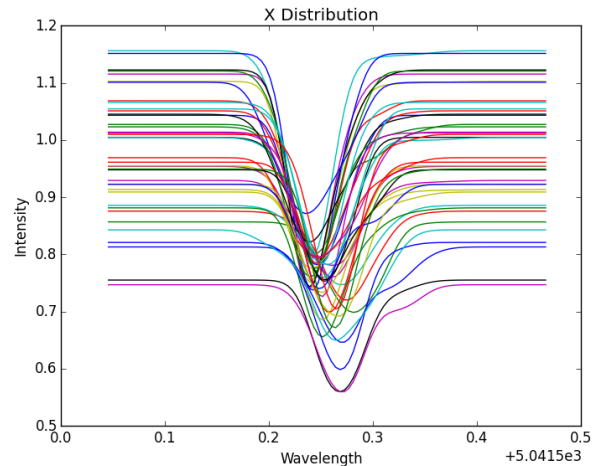


Fig. 4: The distribution of x intensity profiles with $y = 60$.

Figure 4 demonstrates a `xpos` cross section of the modelled atmospheric rectangle with `ypos = 60`. Note only one in every three data points was plotted to reduce the cluttering generated by over plotting 120 profiles. Further note that again, the values are normalised against the average continuum level and hence the spread above and below a continuum of 1 is expected. It is clear from the figure that those profiles with lower continuum values (lower intensity profiles generated by cooler gas) are shifted right of the average to longer wavelengths whilst profiles with higher continuum values (high intensity produced by hot gas) are shifted left to shorter wavelengths. This is the result of Doppler shifting by convective flows with large hot flows of upward moving gas producing a redshift whilst with smaller downflows of cooler gas produce a blueshift. This

clearly demonstrates that convective flows have large effects on the thermal Doppler broadening of the observed wings of spectral lines demonstrating the need for 3D models over 1D approximations.

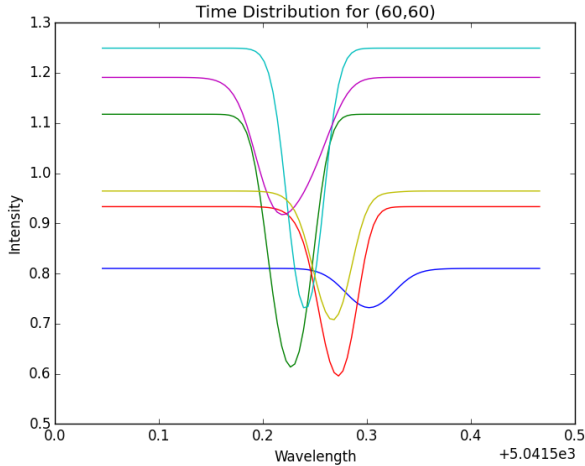


Fig. 5: The time step distribution at a specific point, (60, 60).

Figure 5 plots all six time steps in the data file for the centre point of the grid, (60, 60), displaying the shift of the profile over time. It is clear that the individual profile changes dramatically in response to convective flows. Profiles are visible over all ranges of absorbance and intensity ranging from low-low to high-high and everything in between. This demonstrates how the 3D model is dynamic in its use of convective flows affect the intensity and Doppler shift enormously over the time steps. Furthermore, it demonstrates the dynamic nature of the abundance which also varies with time producing very deep and very shallow absorbance lines. Hence, examining the spatial resolution of the model demonstrates how complex and realistic it is.

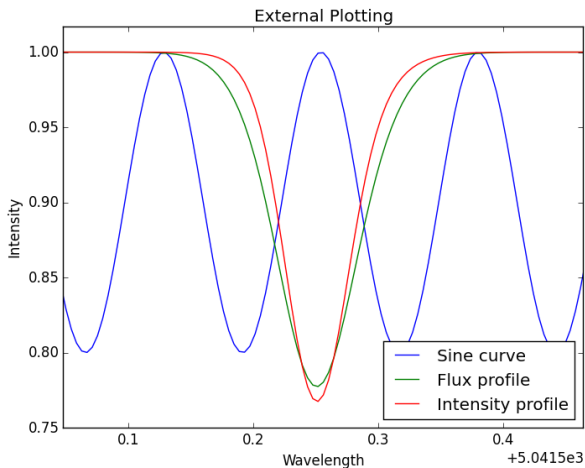


Fig. 6: A sine curve plotted over the intensity and flux profiles.

This covers the majority of the capabilities of the module. Obviously it is significantly flexible but as it only plots the data given, there are some inherent limits to its applicability. One thing to note is that other functions, if specified using numpy before the `profPlot.plot` is called, can also be plotted

over the graph. For instance, consider Figure 6 where a simple sine curve has been plotted against the perpendicular intensity and the total flux profiles. Note also the subtle differences between the intensity profile and the flux profile. Both have very similar equivalent widths however the flux profile is slightly broader as the global integration takes into account the rotational broadening (of the whole star). Hence the distinction between flux and intensity is an important one.

V. REFLECTIONS AND IMPROVEMENTS

With each project completed in Python, further experience in object oriented programming is gained. In particular, whilst working on the construction of the module, numerous problems were encountered that often required complete restructuring of the module. Some of the issues that still plague the module are presented below.

Of primary concern is that the module is not fully optimised. Currently, to calculate a time averaged profile the full multi dimensional array is averaged. Whilst this is appropriate for comparing flux and intensity profiles (which correspond to different variable arrays in the data), when comparing the full cross section of the spatial profile (as in Figure 4), the profile is averaged 120 times. As all 120 profiles come from the same array, this array needs only be averaged once. This produces considerable computational delay which only affects a select range of profiles but results in the computation not being optimised for full generality. This is a result of less than ideal class structure however to fix the issue would require completely re-evaluating the class structure and restarting the debugging process.

Also of minor concern is that currently, the module relies heavily on the assumption that the data structures of the NetCDF files will be identical to the provided example solar data. In particular, NetCDF stores its variables in multi dimensional arrays. If the dimensions of these arrays change, then the module becomes useless because all of the in built slicing of arrays no longer function.

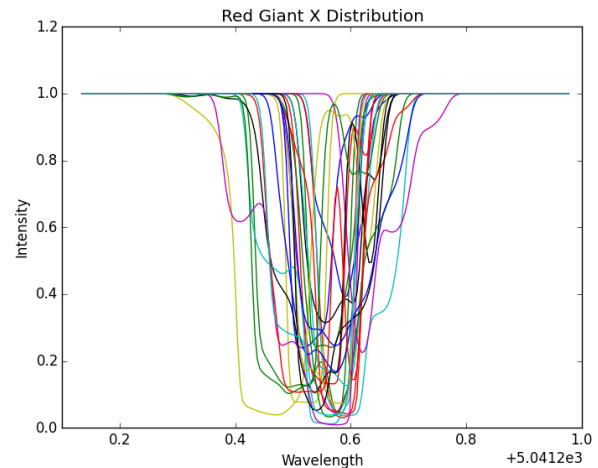


Fig. 7: The distribution of x profiles for a Red Giant.

For example, if a simplified profile was included that no longer had the `nt` or `ngf` dimensions, the module would crash with

an index error. Fortunately, the 3D models used at RSAA always produce data of the same format. Consider Figure 7 which came from a profile file of FeI-504 for a Red Giant atmosphere with only 1 time step and g_f value provided. The profiles plotted are an `xpos` cross section (as in Figure 4) with every fifth element considered. As is evident, the profiles are considerably stronger ($W = 0.1028$) than the sun due to the lower atmospheric temperature in red giants such that the line opacity increases. However the profiles are also much more broadened and distorted due to the more complex convection processes occurring. This example demonstrates the flexibility of the module given a specific data format.

Due to Python's extensive user modules, it is not particularly hard to open and read the data files as efficiently as possible. In fact most actions required by the module can be coded from scratch in less than 50 lines. Hence the wider applications of the module are limited because unlike IDL, not a lot of preparatory work is needed before the data is accessible in the desired format. Furthermore the module does not include anything further than comparative plotting which limits its uses in quantitative astronomy without further development. However if regular and repetitive production of the demonstrated plots is required, the module will significantly increase productivity.

VI. CONCLUSIONS

The primary aim of the presented investigation was to enable the author to become proficient in the use of Python for scientific programming purposes. As with any skill, there are a variety of levels of 'proficiencies' however given the work produced in Appendix A and the courses successfully completed in Appendix B, the author has moved beyond basic proficiency and now requires projects or advanced computing courses to expand his skill set. In this sense, the investigation was a success.

Secondarily, it has been demonstrated that an advanced interface for the analysis of the results of full three dimensional hydrodynamical time-dependent simulations of convection can be implemented successfully using the Python programming language. Included (Appendix A) is a rudimentary implementation of such an interface which allows for a variety of comparative plotting techniques. This includes demonstrations of all the key features of the 3D models as well as qualitative analysis of a variety of profiles. Furthermore, external functions including experimental observations can be over plotted to provide comparisons and assess the success of the 3D models.

There are extensive improvements to be made to make the module presented in Appendix A useful in astrophysical terms. Quantitative analysis methods could be included and the integration with external plotting of observations could be improved upon. Furthermore, several optimisations in the class structure could be implemented to produce a faster and more efficient module. However basic functionality is implemented smoothly and demonstrates the ease and elegance of Python in astronomical applications.

Understanding the processes in the solar atmosphere informs our understanding of other stars and the formation of the spectra that reach us. In particular, spectra from observations provide a

huge amount of information of the stellar system and are used to determine temperatures and other features used to classify stars. Hence good modelling is required to draw correct conclusions about the nature of the star. The mixing length and local thermal equilibrium assumptions provided for the traditional one dimensional models of stellar atmospheres are currently being replaced by three dimensional hydrodynamical time-dependent simulations for vastly improved predictive power. As with any modelling system, interpretive routines are required to turn numerical predictions into useful results. This paper has demonstrated that Python can be used just as effectively as IDL to produce the required interface with the 3D models, often more elegantly and efficiently.

REFERENCES

- [1] L. Schanne, 2014. *Astronomical Optical Spectroscopy; spectra of stars and other celestial objects*. Viewed, 26 May 2015. http://www.astrospectroscopy.eu/index_e.html
- [2] P.S. Barklem, H.C. Stempels, C. Allende Prieto, O.P. Kochukhov, N. Piskunov, and B.J. O'Mara. *Detailed analysis of Balmer lines in cool dwarf stars*. *Astronomy and Astrophysics*, 385:951-967, April 2002.
- [3] O. Kochukhov, S. Bagnulo, and P.S. Barklem. *Interpretation of the core-wing anomaly of Balmer line profiles of cool A_p stars*. *The Astrophysical Journal*, 578:L75-L78, September 2002.
- [4] H.G. Ludwig, N.T. Behara, M. Steffan, and P. Bonifacio. *Impact of granulation effects on the use of Balmer lines as temperature indicators*. *Astronomy and Astrophysics*, 502:L1-L4, June 2009.
- [5] M. Asplund, N. Grevesse, A. J. Sauval, and P. Scott. *The chemical composition of the Sun*. *Annual Review of Astronomy and Astrophysics*, 47:481-522, September 2009.
- [6] C.R. Cowley and F. Castelli. *Some aspects of the calculation of Balmer lines in the sun and stars*. *Astronomy and Astrophysics*, 387:595-604, May 2002.
- [7] C.R. Vidal, J. Cooper, and E.W. Smith. *Hydrogen Stark broadening calculations with the unified classical path theory*. *Journal of Quantitative Spectroscopy and Radiative Transfer*, Vol. 10, pp. 1011-1063, January 1970.
- [8] Fuhrmann, K., Axer, M., and Gehren, T. *Balmer lines in cool dwarf stars*. *Astronomy and Astrophysics*, 271:451-462, March 1993.
- [9] T.M.D. Pereira, M. Asplund, R. Collet, I. Thaler, R. Trampedach, and J. Leenaarts. *How realistic are solar model atmospheres?* *Astronomy and Astrophysics*, 554:A118, June 2013.
- [10] M. Asplund, and K. Lind. *The light elements in the light of 3D and non-LTE effects*. *IAU Symposium No. 268*, February 2010.
- [11] Å. Nordlund, and K. Galsgaard. 1995. *A 3D MHD Code for Parallel Computers*. Tech. rep., Astronomical Observatory, Copenhagen University.
- [12] K. Cruz, 2009. *IDL vs. Python*. AstroBetter. Viewed 28 May 2015. <http://www.astrobetter.com/blog/2009/05/04/idl-vs-python/>
- [13] J. Whitaker. 2014. *Module netCDF4*. Viewed 30 May 2015. <http://netcdf4-python.googlecode.com/svn/trunk/docs/netCDF4-module.html>

APPENDIX A

Attached below are the Python routines of the interface module. Tabbing is preserved as per Python syntax. The module explanations and function descriptions are included after the definition of each class and function. However if a full module description is required, the author can be contacted at guy.leckenby@gmail.com. Note that to use the module below, you will need the netCDF4 and pylab modules installed.

```

import netCDF4
import pylab

# The following is the Python Routine for reading and constructing spectroscopic
# graphs using NetCDF files.

class Variable(object):
    """
    A variable that contains all the information you need to perform the required functions on
    it.
    """
    def __init__(self, fileName, name, ts = None, gf = None):
        """
        Initiates variable inheriting from ProfileFile, ie automatically opens file. Sets
        instance
        variables for use in later calculations.

        Inputs: fileName: - string representing the name of the file to be opened.
               name - string informing us of the variable of interest.
               ts - int representing the time step to be calculated for. Default is None which
                   assumes the variable is time independent. If this is not the case the initial
                   time step is chosen as default.
               gf - int representing the gf value to be calculated for. Default is the middle
                   value.
        """
        self.fileName = fileName
        try:
            self.rootgrp = netCDF4.Dataset(fileName, 'r')
        except RuntimeError:
            print 'No such file or directory. Please ensure you include the full path name.'

        self.name = name
        self.ts = ts
        self.gf = gf

        assert ts < self.rootgrp.variables[name].shape[0], ts + " is not a valid index."

        if gf is None:
            self.gf = self.rootgrp.variables[name].shape[1]/2

    def timeAvg(self, var=None):
        """
        Calculates the temporal average for the provided variable and returns an ndarray.

        Inputs: var - a string representing the variable you want to time average for.
                Default is to use the instance variable.
        Outputs: An ndarray which is simply the average of the old array.
        """
        if var is None:
            return pylab.average(self.rootgrp.variables[self.name], 0)
        else:
            return pylab.average(self.rootgrp.variables[var], 0)

    def contNorm(self, contVar=None):
        """
        Normalises the variable against a chosen continuum.

        Inputs: contVar - a string that is the name of the variable whose continuum you
                wish to normalise against. Default is variable's own continuum.
        Outputs: a list containing the normalised values.
        """
        if contVar is None:

```

```

        contVar = self.name+'_cont'
    if self.ts is None:
        cont = self.timeAvg(contVar)
        norm = self.timeAvg(self.name)[self.gf]/cont
    else:
        cont = self.rootgrp.variables[contVar][self.ts]
        norm = self.rootgrp.variables[self.name][self.ts,self.gf]/cont
    return norm

def equivWidth(self, xVals=None):
    """
    Uses contNorm to calculate the equivalent wavelength for the variable.

    Inputs: attr - the name of the attribute to be used as x values.
    Outputs: a float that is the equivalent width.

    """
    if xVals is None:
        xVals = getattr(self.rootgrp, 'wavelength')
    norm = self.contNorm()
    intNorm = pylab.trapz(norm, x=xVals)
    intCont = xVals[-1]-xVals[0]
    return intCont - intNorm

def __str__(self):
    """
    Directs name of variable to be a string specifying name of variable and the selected
    indexing values.
    """
    return self.name+' (ts='+str(self.ts)+', gf='+str(self.gf)+')'

class SpatialVariable(Variable):
    def __init__(self, fileName, name, ts = None, gf = None, xpos = None, ypos = None):
        """
        Initialises the variable this time keeping spatial information.
        """
        super(SpatialVariable, self).__init__(fileName, name, ts, gf)
        self.xpos = xpos
        self.ypos = ypos

        if xpos is None:
            self.xpos = self.rootgrp.variables[self.name].shape[4]/2
        if ypos is None:
            self.ypos = self.rootgrp.variables[self.name].shape[3]/2
        else:
            assert xpos < self.rootgrp.variables[name].shape[4], xpos + " is not a valid x index."
            assert ypos < self.rootgrp.variables[name].shape[3], ypos + " is not a valid y index."

    def contNorm(self, contVar=None):
        """
        Effectively the same as contNorm for a Variable but calculates for the supplied
        positions.

        Inputs: contVar - a string that is the name of the variable whose continuum you
            wish to normalise against. Default is variable's own continuum.
        Outputs: a list containing the normalised values.
        """
        if contVar is None:
            contVar = self.name+'_cont'
        if self.ts is None and 'xy' in contVar:
            cont = self.timeAvg(var=contVar)[self.ypos,self.xpos]
            norm = self.timeAvg()[self.gf,:,self.ypos,self.xpos]/cont
        elif self.ts is None and 'xy' not in contVar:
            cont = self.timeAvg(var=contVar)
            norm = self.timeAvg()[self.gf,:,self.ypos,self.xpos]/cont
        elif 'xy' in contVar:

```



```

        cont = self.rootgrp.variables[contVar][self.ts, self.ypos, self.xpos]
        norm = self.rootgrp.variables[self.name][self.ts, self.gf, :, self.ypos, self.xpos]/cont
    else:
        cont = self.rootgrp.variables[contVar][self.ts]
        norm = self.rootgrp.variables[self.name][self.ts, self.gf, :, self.ypos, self.xpos]/cont
    return norm

def __str__(self):
    """
    Directs name of variable to be a string specifying name of variable and the selected
    indexing values.
    """
    return self.name+' (ts='+str(self.ts)+' , gf='+str(self.gf)+' , x =' +str(self.xpos)+' ,
        y='+str(self.ypos)+' )'

class AngularVariable(Variable):
    def __init__(self, fileName, name, ts = None, gf = None, mu = None, phi = None):
        """
        Initialises the variable this time keeping angular information.
        """
        super(AngularVariable, self).__init__(fileName, name, ts, gf)

        if gf is None:
            self.gf = self.rootgrp.variables[name].shape[3]/2

        self.mu = mu
        self.phi = phi

        if mu is None:
            self.mu = self.rootgrp.variables[self.name].shape[1]/2
        if phi is None:
            self.phi = self.rootgrp.variables[self.name].shape[2]/2
        else:
            assert mu < self.rootgrp.variables[name].shape[1], mu + " is not a valid mu index."
            assert phi < self.rootgrp.variables[name].shape[2], phi + " is not a valid phi index."

    def contNorm(self, contVar=None, phiAvg=False):
        """
        Effectively the same as contNorm for a Variable but calculates for the supplied angles.

        Inputs: contVar - a string that is the name of the variable whose continuum you
            wish to normalise against. Default is variable's own continuum.
        Outputs: a list containing the normalised values.
        """
        if contVar is None:
            contVar = self.name+'_cont'
        if self.ts is None and 'angle' in contVar:
            cont = self.timeAvg(var=contVar)[self.mu, self.phi]
            norm = self.timeAvg()[self.mu, self.phi, self.gf]/cont
            phi = self.timeAvg()[self.mu, :, self.gf]/cont
        elif self.ts is None and 'angle' not in contVar:
            cont = self.timeAvg(var=contVar)
            norm = self.timeAvg()[self.mu, self.phi, self.gf]/cont
            phi = self.timeAvg()[self.mu, :, self.gf]/cont
        elif 'angle' in contVar:
            cont = self.rootgrp.variables[contVar][self.ts, self.mu, self.phi]
            norm = self.rootgrp.variables[self.name][self.ts, self.mu, self.phi, self.gf]/cont
            phi = self.rootgrp.variables[self.name][self.ts, self.mu, :, self.gf]/cont
        else:
            cont = self.rootgrp.variables[contVar][self.ts]
            norm = self.rootgrp.variables[self.name][self.ts, self.mu, self.phi, self.gf]/cont
            phi = self.rootgrp.variables[self.name][self.ts, self.mu, :, self.gf]/cont
        if phiAvg is True:
            return phi
        else:
            return norm

```

```

def phiAvg(self, cont=True, contVar=None):
    """
    This function averages over all the phi values for the given mu value.

    Output: a list containing the phi averaged values.
    """
    if cont is True:
        contNorm = self.contNorm(contVar=contVar, phiAvg=True)
        avg = pylab.average(contNorm, 0)
        return avg
    else:
        index = self.rootgrp.variables[self.name].dimensions.index(u'nphi')
        return pylab.average(self.rootgrp.variables[self.name], index)

def __str__(self):
    """
    Directs name of variable to be a string specifying name of variable and the selected
    indexing values.
    """
    return self.name+' (ts='+str(self.ts)+' , gf='+str(self.gf)+' , mu =' +str(self.mu)+' ,
        phi='+str(self.phi)+' )'

def plot(variable, xVals, lgnd = None, xlabel = None, ylabel = None, title = None):
    """
    Plots a given variable against a given attribute without normalising the variable.


    Inputs: variable - list of Variables or lists of values to be plotted.
            xVals - a list containing the x values to be plotted against.
            lgnd - list of strings to be used as labels. Default is no labels.
            title, xlabel, ylabel - string representing aforementioned. Default is no labels.
    Outputs: A graph plotting the variable against the attribute.
    """
    if lgnd is None:
        for a in variable:
            pylab.plot(xVals, a)
    else:
        for a,b in enumerate(variable):
            pylab.plot(xVals, b, label = lgnd[a])
    if xlabel != None:
        pylab.xlabel(xlabel)
    if ylabel != None:
        pylab.ylabel(ylabel)
    if title != None:
        pylab.title(title)
    pylab.legend(loc = 'best')
    pylab.show()

```

APPENDIX B

The author at the beginning of this investigation had no knowledge of any programming language. Since then, he has become proficient in the use of Python and the understanding of fundamental programming concepts if not the higher algorithm structuring. As proof of this, included are the certificates gained from completing two online courses through the platform edX. The respective grades in each course were 96% and 91%.

HONOR CODE
CERTIFICATE



Guy Leckenby

successfully completed and received a passing grade in

6.00.1x: Introduction to Computer Science and Programming Using Python

a course of study offered by MITx, an online learning initiative of The Massachusetts Institute of Technology through edX.

W. Eric L. Grimson
Bernard Gordon Professor of Medical Engineering
Chancellor for Academic Advancement
Massachusetts Institute of Technology


John Guttag
Dugald C. Jackson Professor of
Computer Science and Electrical Engineering
Massachusetts Institute of Technology

Sanjay Sarma
Director of Digital Learning
Massachusetts Institute of Technology

HONOR CODE CERTIFICATE
Issued March 13th, 2015

Verify the authenticity of this certificate at
<https://verify.edx.org/cert/30f749f981244f42abf9af26e844ba24>

HONOR CODE
CERTIFICATE



Guy Leckenby

successfully completed and received a passing grade in

6.00.2x: Introduction to Computational Thinking and Data Science

a course of study offered by MITx, an online learning initiative of The Massachusetts Institute of Technology through edX.

W. Eric L. Grimson
Bernard Gordon Professor of Medical Engineering
Chancellor for Academic Advancement
Massachusetts Institute of Technology

John Guttag
Dugald C. Jackson Professor of
Computer Science and Electrical Engineering
Massachusetts Institute of Technology

Sanjay Sarma
Dean of Digital Learning
Massachusetts Institute of Technology

HONOR CODE CERTIFICATE
Issued May 20, 2015

Verify the authenticity of this certificate at
<https://verify.edx.org/cert/54956e32b73d4d1e8457366a707ecba6>