COMPUTATIONALLY INTENSIVE PROBLEMS OF PHYSICS AND ASTRONOMY: OSCILLATOR STRENGTHS AND DEPARTURE COEFFICIENTS OF THE HYDROGEN ATOM IN THE INTERSTELLAR MEDIUM

A thesis submitted to Auckland University of Technology
in partial fulfilment of the requirements for the degree of Master of Science

Supervisors
Sergei Gulyaev
Andrew Ensor

May 2013

By
Boris Feron
School of Computing and Mathematical Sciences

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my primary supervisor, Professor Sergei Gulyaev, for his invaluable help and guidance. Without his support, knowledge and keen insight into the area of radio astronomy this thesis could not have been written. Furthermore, I highly value his patience in going through my thesis countless times to correct for any inaccuracies, no matter how small.

I would also like to thank my secondary supervisor, Dr Andrew Ensor. With his expertise in computer science, and especially high performance computing, I am confident that this thesis will help bring to light some of the new and very interesting ways that old problems in radio astronomy can be revitalised.

Thank you to Jordan Alexander for his countless efforts in aiding my understanding of numerous topics in radio astronomy. I also greatly appreciate the discussions we have had on unrelated topics to help me get my mind off things.

A big thanks to Dr Guy Kloss for assisting me in running experiments on the test machine and helping with odd Linux questions.

Thank you very much to Eleanor Da Fonseca for having patience and giving me encouragement when things were getting off track.

I would like to thank Kordia, the Bank of Nordea, the IRASR at AUT as well as the Danish Government's recent study abroad scholarship. Without the financial help of any of these, none of this would have been possible.

Lastly I would like to thank everyone at the Institute for Radio Astronomy and Space Research at AUT for their company and encouragement.

Declaration

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person (except where explicitly defined in the acknowledgements), nor material which to a substantial extent has been submitted for the award of any other degree or diploma of a university or other institution of higher learning.

Copyright

Abstract

Calculating *departure coefficients*, b_n, as well as $b_{n,l}$, for non-LTE gases/plasmas, is a fundamental computational problem in radio astronomy, physics of the interstellar medium, and for diagnostics of plasmas of nuclear reactors. Most work in this area was done in the 1960s and 1970s. Recent advances in computing technology have rendered the technology used in these two decades obsolete. Hence we ask if the approximate techniques developed to compensate for the technological limitations of the 1960s and 1970s are still needed.

In this thesis we introduce modern computational techniques to solve, exactly, the computational problems relating to departure coefficients. Specifically, we have made use of *arbitrary precision arithmetic* as well as introducing *GPU & parallelization* techniques to already established solutions.

We investigated the problem of the hypergeometric function which arises as the solution of the wave equation for hydrogen, which is the key component in Einstein coefficients, radiative recombination rates and the Stark broadening theory. Furthermore, we implemented, optimized and compared two different techniques for calculating b_n coefficients and developed a matrix approach for dealing with the $b_{n,l}$ problem.

We hope that the solutions resulting from this thesis will pave the way for further development in the outlined area, allowing for exact solutions up to $n = 1000$ and greater.

Chapter 1

Introduction

Radio astronomy is a rather young area of science and its birth is generally credited to Karl Jansky who first discovered radio emission from the Milky Way in 1932 ("Encyclopaedia Britannica", 2013). However, not until the firm identification of "radio stars" was conducted in New Zealand in the end of the 1940's (Bolton & Stanley, 1948) and the discovery of the 21cm hydrogen line in the 1950's (van de Hulst, 1951) did the importance of Jansky's findings become apparent to the general scientific community (Dopita & Sutherland, 2003, p. 1).

In this thesis we will deal with problems related to the area known as Departure Coefficients which become important when dealing with the Radiative Transfer Model as used in the field of Radio Recombination Lines. A brief introduction on the relation between Radio Astronomy, Radio Recombination Lines, the Radiative Transfer Model and finally the Departure Coefficients will therefore follow.

1.1 Radio Astronomy

In ground based radio astronomy we deal with electromagnetic waves of frequencies approximately between 10 MHz and 1 THz, or, correspondingly, wavelengths between 30 m and 0.3 mm. This part of the electromagnetic spectrum corresponds to the "transparency window" of the Earth's atmosphere at these wavelengths. A telescope on Earth that is observing a radio source with a peak in this spectrum will receive a substantially large amount of the source's emitted radio waves towards the Earth as the atmosphere will not absorb these wavelengths (Dopita & Sutherland, 2003).

1.2 Radio Recombination Lines

Radio recombination lines are a special case of spectral lines formed when a positively charged ion recombines with a free electron in plasmas or partially ionized gases. This electron, which is normally captured in a high energy state, will then cascade down through allowed energy states. The probability of these transitions are governed by the laws of quantum mechanics. Each spontaneous transition releases energy in the form of radiation.

Observing a radio source that produces radio recombination lines, and using methods of spectroscopy, we are capable of determining the source's temperature and density, as well as the composition of the chemical elements that make up these sources (Gordon & Sorochenko, 2002).

1.2.1 Radiative Transfer Model

After obtaining a spectrum for a given source we can use the radiative transfer model to predict its density and temperature. The radiative transfer equation takes on the form:

$$dI = -I\kappa dx + jdx \tag{1.1}$$

where dI is the net change in the radiation intensity at a given frequency, ν; x is the distance the radiation travels in the source towards the observer, κ is the linear absorption coefficient for all depletions from the radiation in the direction of the observer, and j is the emission coefficient for all gains in intensity in the direction of the observer (Chandrasekhar, 1960, p.9).

It should be noted that although the convention is to use I, a more appropriate notation would be I_ν, as we are measuring the intensity at a specific frequency. Fig. 1.1 depicts the radiation travelling in the source.

Radio recombination lines are not the only contributing factor to the intensity received by the telescope. We must also account for continuum radiation from heated gas particles. This radiation is also known as the thermal radiation due to "free-free" transitions (Gordon & Sorochenko, 2002, p.58). "Free-free" transitions occur when an electron close to an ion decelerates due to electric interaction i.e. loses kinetic energy which is converted into radiation. This radiation, unlike that of radio recombination lines, is continuous. However, it

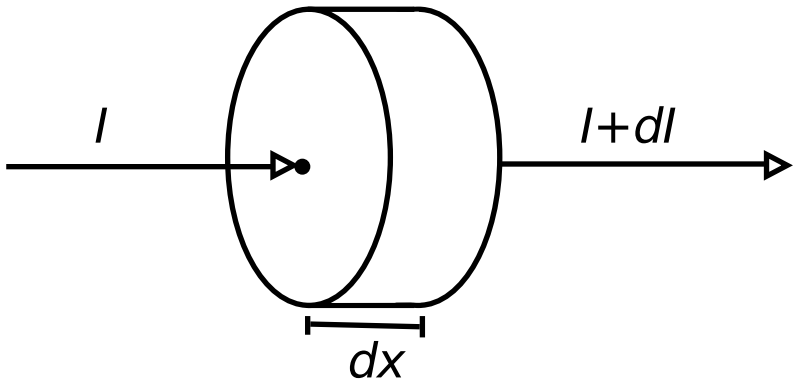

Figure 1.1: Illustration of the Radiative Transfer Model

still contributes to the intensity of a given frequency. Rewriting the intensity, I, into its components we obtain:

$$I = I_C + I_L \tag{1.2}$$

where I_C is the intensity of continuous radiation and I_L is the intensity of line radiation. This is possible due to the linearity of κ and j. We shall henceforth deal exclusively with the discrete subcomponents, κ_L and j_L. We note that under the assumption of thermodynamic equilibrium:

$$j_L = \kappa_L B_\nu(T) \tag{1.3}$$

as stated by Chandrasekhar (1960, p.8) where $B_\nu(T)$ is the Planck function for radiation of a black body with temperature T. Using this equation we need only focus on arriving on an expression for κ_L after which we can use (1.3) to find j_L.

Following Gordon and Sorochenko (2002, p.64) we have:

$$\kappa_L = \frac{h\nu}{4\pi}\phi_\nu(N_{n_1}B_{n_1,n_2} - N_{n_2}B_{n_2,n_1}) \tag{1.4}$$

In this equation, n_1 denotes the lower principal quantum number and n_2 the upper principal quantum number; N, along with its subscript, denotes the population density at that principal quantum number; $B_{n,m}$ denotes the Einstein coefficient for stimulated absorption and emission in the direction left to right. The units of the $B_{n,m}$ Einstein coefficient are inverse specific intensity per unit

time; ϕ_ν is the line profile with units Hz^{-1} and h is the Planck constant.

Assuming thermodynamic equilibrium we can write the equation for the relative populations between two principal quantum states, n_1 and n_2:

$$\frac{N_{n_2}}{N_{n_1}} = \frac{\omega_{n_2}}{\omega_{n_1}} e^{-h\nu/(kT)} \tag{1.5}$$

where ω_n is the statistical weight at the level n. Secondly, according to Lang (1975, p.91):

$$\omega_m B_{m,n} = \omega_n B_{n,m} \tag{1.6}$$

Substituting (1.5) and (1.6) into (1.4) we get:

$$\kappa_L = \frac{h\nu}{4\pi} \phi_\nu N_{n_1} B_{n_1,n_2} \left[1 - e^{-h\nu/(kT)}\right] \tag{1.7}$$

Using the notion of an oscillator strength, given by:

$$f_{n_1,n_2} = -\frac{\omega_{n_2}}{\omega_{n_1}} f_{n_2,n_1} \tag{1.8}$$

$$= \frac{m_e c h\nu}{4\pi^2 q_e^2} B_{n_1,n_2} \tag{1.9}$$

where q_e is the charge of an electron, we can rewrite our equation for κ_L as:

$$\kappa_L = \frac{\pi q_e^2}{m_e c} \phi_\nu N_{n_1} f_{n_1,n_2} \left[1 - e^{-h\nu/(kT)}\right] \tag{1.10}$$

This may not appear to be of much use. However, much research has gone into oscillator strengths and it can therefore be convenient, as we will see later, to use this instead of the form involving B_{n_1,n_2}.

By once again assuming thermodynamic equilibrium we can make use of the Saha-Boltzmann equation to obtain an expression for N_{n_1} relating it to temperature and the density of electrons and ions, respectively N_e and N_i:

$$N_{n_1} = \frac{N_e N_i}{T^{3/2}} \frac{n_1^2 h^3}{(2\pi m_e k)^{3/2}} \exp\left(\frac{Z^2 E_{n_1}}{kT}\right) \tag{1.11}$$

where Z is the atomic number ($Z = 1$ for the case of hydrogen which we are dealing with) and E_{n_1} is the ionization energy at level n_1. We are now able to

write an explicit equation for κ_L using (1.10) and (1.11):

$$\kappa_L = \frac{\pi h^3 q_e^2}{(2\pi m_e k)^{3/2} m_e c} n_1^2 f_{n_1,n_2} \phi_\nu \times \frac{N_e N_i}{T^{3/2}} \exp\left(\frac{Z^2 E_{n_1}}{kT}\right)\left(1 - e^{-h\nu/(kT)}\right) \tag{1.12}$$

Using (1.12) we are able to obtain an expression for j_L assuming thermodynamic equilibrium, as given by (1.3).

So far we have assumed thermodynamic equilibrium in order to derive our expression for κ_L. Although thermodynamic equilibrium does not occur in the astronomical systems that we are dealing with, there are situations where locally one may assume thermodynamic equilibrium. We refer to these as being in local thermodynamic equilibrium (abbr. LTE). When assuming LTE we can accordingly use (1.12) (Gordon & Sorochenko, 2002, pp.70-71).

1.3 Departure Coefficients

So far we have assumed LTE conditions when calculating line strengths using κ_L. This allows us to assign a single temperature, T, to the entire system. However, comparing observed temperatures of the M17, Orion and W51 nebulae (Gordon & Sorochenko, 2002, p.69) (using different, more well-established techniques for calculations of T) gives a difference of a factor of 2 as compared to using the temperature calculated from the radiative transfer model, using the derived expression for κ_L given in (1.12).

This disagreement in temperature stems from the incorrect assumption of LTE when dealing with populations of atomic levels. When non-LTE conditions are present we have not one, but two temperatures present: The excitation temperature, T_{ex}, which describes the relative population of bound quantum levels and T_e, which is the electron temperature of the ionized gas in the nebula (Gordon & Sorochenko, 2002, p.71). Using the result of Goldberg (1966, p.1225) we have:

$$e^{h\nu/(kT_{ex})} = \frac{b_n}{b_{n-1}} e^{-h\nu/(kT_e)} \tag{1.13}$$

In (1.13) we introduce the *departure coefficient*, b_n. It is a correction factor that gives the ratio between the actual (non-LTE) number of atoms in a level n,

compared to the number of atoms in level n, when assuming LTE. We reproduce the final equation for the line absorption coefficient including correction factors, as given by Gordon and Sorochenko (2002, p.73):

$$\kappa_L = \kappa_L^* b_{n_1} \left[\frac{1 - (b_{n_2}/b_{n_1}) e^{-h\nu/(kT_e)}}{1 - e^{-h\nu/(kT_e)}} \right] \tag{1.14}$$

$$= \kappa_L^* b_{n_1} \beta \tag{1.15}$$

where κ_L^* is the LTE version of κ_L.

In order to find κ_L and solve the radiative transfer equation (1.1) we must find the unknown b_n coefficients for each atomic level n. As each b_n is simply a correction-factor that gives us the actual population density at level n i.e. N_n compared to the density assuming LTE i.e. N_n^*, we have:

$$b_n \equiv N_n / N_n^* \tag{1.16}$$

We now solve for each N_n through the following system of equations, which states that the number of all possible transitions out of a given quantum level n is equal to the number of all the transitions into the level n. This is known as the statistical equilibrium equation:

$$N_n \sum_{m=n_0, n \neq m}^{\infty} P_{nm} = \sum_{m=n_0, n \neq m}^{\infty} N_m P_{mn} \tag{1.17}$$

where the lower limit is the lowest quantum state considered. Commonly we set $n_0 = 1$ or $n_0 = 2$. We refer to these as Case A and Case B respectively. Coefficients P_{nm} and P_{mn} in equation (1.17) are the probabilities of the corresponding transitions between quantum states (see below). Infinity represents the theoretical upper quantum state. Given that the system of equations must be solved numerically, infinity must be replaced with a finite number. However, according to Dupree (1969, p.493), $b_n = 1$ above some level n_{max} due to collisional coupling with the continuum. As such we can let $b_n = 1$ for $n > n_{max}$ and thus establish a finite system of equations.

The processes that contribute to level depopulation, and their probabilities, are:

$$A_{nm} = \text{Spontaneous radiation from level } n \text{ down to level } m \quad (1.18)$$
$$C_{nm} = \text{Collisional transitions out of level } n \text{ to level } m \text{ (up \& down)} \quad (1.19)$$
$$B_{nm} = \text{Stimulated radiation out of level } n \text{ to level } m \text{ (up \& down)} \quad (1.20)$$
$$C_{ni} = \text{Collisional ionization out of level } n \text{ to } \textit{continuum} \quad (1.21)$$
$$B_{ni} = \text{Stimulated radiative ionization out of level } n \text{ to } \textit{continuum} \quad (1.22)$$

The processes that contribute to level population are:

The same processes as in equation (1.18)-(1.20) , but in reverse along with three more components:

$$\alpha_n = \text{Radiative recombination from the } \textit{continuum} \text{ to level } n \quad (1.23)$$
$$C_{in} = \text{Collisional three-body recombination from } \textit{continuum} \text{ to level } n \quad (1.24)$$
$$B_{in} = \text{Stimulated radiative recombination from } \textit{continuum} \text{ to level } n \quad (1.25)$$

Using these contributions to level population/depopulation we can expand on (1.17) and obtain:

$$N_n \left(\sum_{m=n_0, m\neq n}^{\infty} (C_{nm} + B_{nm}\rho_\nu) + \sum_{m=n_0, m\neq n}^{n-1} A_{nm} + C_{ni} + B_{ni}\rho_\nu \right) = \quad (1.26)$$
$$\sum_{m=n_0, m\neq n}^{\infty} N_m(C_{mn} + B_{mn}\rho_\nu) + \sum_{m=n+1}^{\infty} N_m A_{mn} + N_e N_i(\alpha_n + C_{in}) + B_{in}\rho_\nu$$

where ρ_ν is the radiation density at frequency ν.

1.4 Introduction to Computations on Graphical Processing Units (GPUs)

General Purpose GPU programming did not become popular until Nvidia released their Compute Unified Device Architecture (CUDA) in 2006/2007 and the Khronos group standardised the Open Computing Language (OpenCL) (Murthy, Ravishankar, Baskaran, & Sadayappan, 2010). However, when presently discussing high-performance scientific computations it is impossible to avoid

the topic of GPUs. Before that, any code written for execution on the GPU had to be done using native graphics APIs, such as OpenGL and Microsoft DirectX, and graphics programming languages, such as GLSL and HLSL (for writing what is known as the kernel) (Strzodka, Doggett, & Kolb, 2005, p.670). These languages are not well suited for general purpose computation as they are aimed at facilitating programming graphics problems that are ultimately meant to be displayed on the computer screen.

To understand the need for the development of CUDA and OpenCL, we must first clarify the difference between the Central Processing Unit (CPU) architecture and the GPU architecture.

The CPU architecture performs what is known as *instruction stream processing*. In this architecture, which is based on the von Neumann architecture, we store data and instructions together in the same memory space (Strzodka et al., 2005, p.668). In sequence, each instruction will load in data from memory. No use is made of large data blocks that all use the same instruction, as each instruction will load in the data needed only when it is executed. This model is called the SISD model (Single Instruction Single Data) and is the most commonly implemented instruction architecture on CPUs. Fig. 1.2 shows an overview of instruction stream processing.

GPUs rely on what is called the SIMD model (Single Instruction Multiple Data). Although modern CPUs also make use of the SIMD model, GPUs are highly optimized for this model and have many more of its benefits, as well as its drawbacks. This model makes use of the fact that the same instructions will be run on a large block of data. A very effective example for use of this is a loop where the body is working on a large set of data which at each point is independent of its neighbour points. A commonly used example is the loop given in Fig. 1.3. This loop has properties that make it ideal for the SIMD model:

1. It is a *for*-loop resulting in a fixed number of iterations
2. The computation of element $[i, j]$ is independent of all other computations

Property 1 means that before executing the loop, we know how many iterations there will be, as opposed to a while loop where the number of iterations will be undetermined. A while-loop that will in fact have a fixed amount of iterations can be turned into a for-loop.

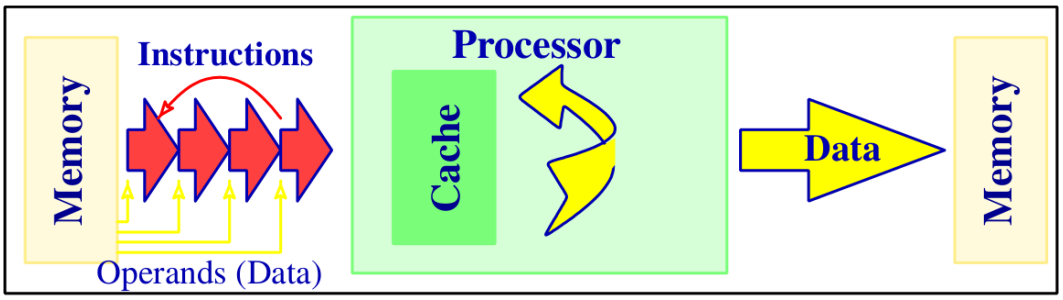

Figure 1.2: Instruction Stream Processing (Strzodka et al., 2005).

```
1 for(i=0;i<row;i++)
2     for(j=0;j<col;j++)
3         Output[i][j] = Input1[i][j]+Input2[i][j];
```

Figure 1.3: `GPU_loop.c`

Property 2 means that for any i, j we can compute the result without regard to other computations. This very important property means that all calculations of the loop can be done in parallel without synchronization. As such we can let the GPU decide on its own which calculation needs to be done when. This demonstrates how the function of the loop is not to determine in what order elements are calculated, but rather to iterate through the loop variables i and j. Hence, instead of a loop we could imagine a 2D array with dimensions $i_{max} \times j_{max}$, where the loop iterates through, possibly at random, all possible combinations of i and j. This is precisely how the GPU architecture is constructed. CUDA and OpenCL have slightly different ways of maintaining the indexing as just described (1D, 2D and 3D indexing). However, both rely on defining a *kernel*, as mentioned earlier, where the body of the loop is performed. The GPU is then free to run the kernel in parallel for any given index values.

This shows the nature of SIMD where we have a large set of data on which we perform the same instruction independently of each other. The reason the GPU is so fast at performing tasks like this is because it uses a *Data stream processing* architecture as opposed to the CPU's instruction stream processing. In this architecture we configure the GPU *pipeline*, as illustrated in Fig. 1.4. This includes what the input and output array should be along with loading the kernel function. Once this is configured we need only load in the entire data once and the GPU can in parallel compute the kernel value for each index

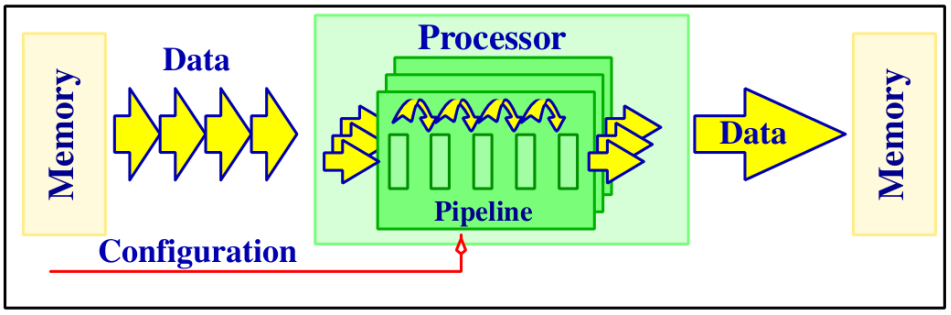

Figure 1.4: Data Stream Processing (Strzodka et al., 2005).

pair i, j. GPUs have many more computing cores than CPUs and as such can perform many more computations in parallel.

In this thesis we will be dealing with certain computationally intensive tasks. Each of these tasks will be subcomponents of the overlying departure coefficient problem. The goal of the departure coefficient problem is to correctly describe the temperature and density for a given spectrum. As such, these parameters can be considered variables for the departure coefficient problem. We shall see examples further on where this assumption allows us to favour simplicity in calculations over speed. However, in general, where GPU optimization has not been possible or feasible due to time constraints, measures have been taken to optimize run time.

1.5 Research Problem

The majority of papers relating to the b_n problem and its subcomponents (oscillator strengths, Gaunt factors, recombination coefficients etc.) have been written in the 1960s and 1970s (see for example a review of the b_n problem in Gordon and Sorochenko (2002, pp.77-78)). Even though, by that time, computer techniques had advanced far beyond the early ABC and ENIAC computers of the 1940s and 1950s, the largest mainframes of the 1970s would still be no match for modern day computers. As an example we can mention the University College London IBM 360/65 mainframe, which was used to solve one of the problems we will cover in this thesis, that ran at approximately 0.1 MFLOPS ("System/360 Model 50", 2013). With this kind of computing performance, approximate techniques were necessarily developed to make the

calculations manageable. This was especially needed when including angular momentum quantum number l in b_n (now $b_{n,l}$) calculations, as the complexity greatly increased.

We can compare the performance of an IBM 360/65 mainframe with that of a current Nvidia Tesla C20xx series GPU, which is nothing more than a component in a desktop computer. At 515 GFLOPS, the Nvidia Tesla GPU outperforms the IBM 360/65 mainframe by a factor of more than 5 million. With this kind of difference in performance, one might wonder if we still need the approximate solutions developed in the 1960s and 1970s.

In this thesis we introduce modern computational techniques to solve computational problems, relating to departure coefficients, exactly. This will include:

- *Arbitrary precision arithmetic*: Introducing the technique of arbitrary precision arithmetic to solve exactly the hypergeometric function (see eq. (2.17)) without the use of recursive relations and analyse its behaviour as it relates to departure coefficients.

- *GPU & parallelization*: Optimizing existing solutions and discussing possible optimizations to departure coefficient problems by re-implementing solutions to make use of modern-day techniques such as GPUs and parallelization.

- $b_{n,l}$ *problem*: Do the groundwork needed to allow for further development in the area of departure coefficients that focuses on creating an exact solution to the $b_{n,l}$ problem for n up to 1000 and beyond.

We hope that these developments will allow us to advance from current simplistic to more complex, and hence more realistic, models of the interstellar medium (ISM) objects, thereby progressing to a better understanding of the physics and evolution of the ISM and star-formation in our galaxy.

In Chapter 2 we introduce the topic of Einstein coefficients and oscillator strengths. We give a thorough presentation of the key component of both terms i.e. the hypergeometric function and some of the problems that occur when trying to compute it.

In Chapter 3 we introduce radiative recombination. As for Chapter 2 the key component is the hypergeometric function. However, this time we use a recursive relation and give a thorough computer science based break down of how to solve it most efficiently and optimize it to make use of GPUs.

In Chapter 4 we solve the entire b_n problem based on an iterative scheme by Sejnowski and Hjellming (1969). We then discuss the results and how GPU optimizations would be possible.

In Chapter 5 we solve the entire b_n problem by solving the system of linear equations given in equation (1.26). We then optimize certain key components to make use of GPUs and compare the results to that of the single threaded solution. Finally, we compare this approach to solving the b_n problem to that of the iterative approach in Chapter 4.

Lastly, the conclusion is given in Chapter 6.

Chapter 2

Einstein Coefficients and Oscillator Strengths

2.1 Introduction to Einstein Coefficients and Oscillator Strengths

In this chapter we will deal with the notions of Einstein coefficients and oscillator strengths. As we only touched very briefly on this in Chapter 1 we will now elaborate further.

Let us consider a neutral hydrogen atom in an excited state, in a low density gas, in the absence of any magnetic field, far from any source of radiation. On a short time scale, as it is far away from other particles, we might assume it to be "isolated". The state of this atom is then fully described by the laws of spontaneous radiation. These laws determine the probability of spontaneous transition from a given upper state n, l downwards to a state n', l'. The upper number n is not limited whereas the lowest value of n' is 1. Furthermore, as by the definition of l, we have $l \in \{0, 1, ..., n-1\}$. However, quantum selection rules dictate that $\Delta l = \pm 1$ (Bethe & Salpeter, 1957). From this we have that each level n has $(n-1)^2$ possible downward transitions (see section 2.5). We use the Einstein coefficient for spontaneous radiation (also known as spontaneous emission) to describe this downward transition as a probability giving the number of transitions per second per unit volume (Pradhan & Nahar, 2011, p.73). It should be noted that this transition is completely random and that the Einstein coefficient is only an overall probability and cannot be used to

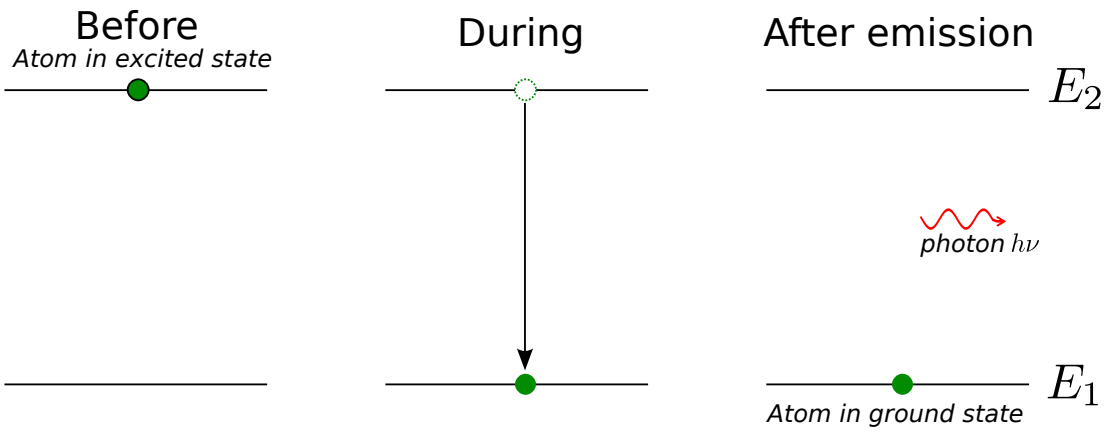

Figure 2.1: Spontaneous radiation from energy state E_2 down to energy state E_1. Adopted from http://commons.wikimedia.org/wiki/File:Spon
taneousemission.svg. Licensed under the "GNU Free Documentation License".

tell when a specific electron will make a transition downwards. When this transition occurs, it releases a photon of energy $E_2 - E_1 = h\nu$ as depicted schematically in Fig. 2.1. This photon is, unlike for stimulated radiation, released in no particular direction and with no particular phase.

Let us now assume that a source of external radiation is present. The atom is now no longer isolated as photons from the external radiation will interfere with its state. Following Dopita and Sutherland (2003, p.13) we define the energy per unit volume received from this electromagnetic field as the *energy density* and denote it by $U(\nu_{12})$, measured in erg cm^{-3}, where ν_{12} denotes the frequency at which the radiation occurs. We recognise this as the radiation density, ρ_ν, from equation (1.26). This interference occurs in the form of stimulated emission and stimulated absorption.

We have already introduced the Einstein coefficients for these in equation (1.4), where equation (1.6) describes the relation between the two Einstein B coefficients. Furthermore, the relation between the Einstein coefficients A and B is given by:

$$\left(\frac{8\pi h}{c^3}\right)\nu_{12}^3 B_{12} = A_{21}\frac{\omega_2}{\omega_1} \tag{2.1}$$

$$\Updownarrow$$

$$B_{12} = A_{21}\frac{\omega_2}{\omega_1}\left(\frac{c^3}{8\pi h}\right)\frac{1}{\nu_{12}^3} \tag{2.2}$$

where B_{12} is the Einstein coefficient for absorption and ω_1 and ω_2 are the statistical weights of states 1 and 2. Note that the units of the B coefficients are not s^{-1} as we need to multiply them by ρ_ν to account for the amount of radiation received. Hence the rate of excitation per second of an atom from a lower state 1 to an upper state 2, is:

$$B_{12}\rho_\nu \tag{2.3}$$

For stimulated radiation we simply reverse the subscripts and use the appropriate detailed balance to describe it in terms of the Einstein A coefficient:

$$B_{21} = \left[\frac{\omega_1}{\omega_2}\right] B_{12} = \left[\frac{\omega_1}{\omega_2}\right] A_{12} \left[\frac{\omega_2}{\omega_1}\right] \left(\frac{c^3}{8\pi h}\right) \frac{1}{\nu_{12}^3} \tag{2.4}$$

$$= A_{12} \left(\frac{c^3}{8\pi h}\right) \frac{1}{\nu_{12}^3} \tag{2.5}$$

We thus have that the rate of stimulated radiation from an upper level 2 to a lower level 1 is:

$$B_{21}\rho_\nu \tag{2.6}$$

We note that the subscript of ν is identical for absorption and radiation at given upper level 2 and lower level 1, as the frequency of the incident photon is the same. Fig. 2.2 shows the process of stimulated radiation and Fig. 2.3 shows the process of stimulated absorption. Note that in stimulated radiation, the phase, frequency and direction of the emitted photon is identical to that of the incident photon. Hence when stimulated radiation occurs, an incident photon is effectively "transformed" into two identical photons traveling in the same direction, having identical frequency and phase. This principle, along with that of population inversion, are the main principles behind masers, which are important, naturally occurring objects in radio astronomy (Singer, 1959).

So far we have only introduced the two quantum numbers n and l. However, there are two remaining quantum numbers which we have not accounted for, namely the magnetic quantum number m and the intrinsic orbital momentum, or spin, number s. When an external magnetic field is present, the m levels are no longer degenerate i.e. they become distinct energy levels. This is known as the *Zeeman* effect. However, when there is no magnetic field, which is the case we are dealing with in this thesis, all the orbital momentum states m become

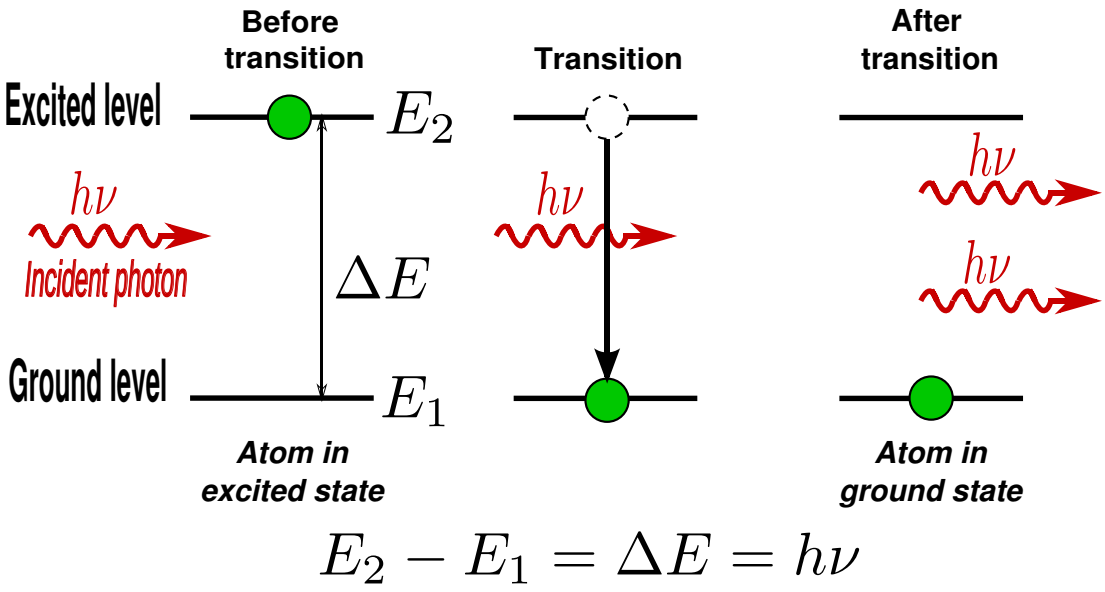

Figure 2.2: Stimulated radiation from energy state E_2 down to energy state E_1. Adopted from http://upload.wikimedia.org/wikipedia/comm
ons/0/09/Stimulated_ Emission.svg. Licensed under the "GNU Free Documentation License".

degenerate and no distinction between each value of m, in terms of energy, can be made. Hence we do not consider m. We include s as the factor 2 in the calculation of statistical weight. Although it is also present, theoretically, in the transition between two identical states that only differ in spin, the probability of this is so low that we do not consider it.

2.2 Calculation of Einstein Coefficients and Oscillator Strengths

In order to calculate the contribution of spontaneous emission (Einstein coefficient A) along with stimulated emission and absorption (Einstein coefficient B) for the population density at a quantum level n, we use the notion of an *oscillator strength*, f, introduced in equations (1.8)-(1.9).
Common for these three terms is the use of the Wave function, Ψ, in order to compute their values. As the concept (not complexity) of the general and "honest" solution (i.e. no approximate solution) for these terms does not differ when considering the orbital momentum quantum number l, we will include l in our calculations of the above-mentioned terms.

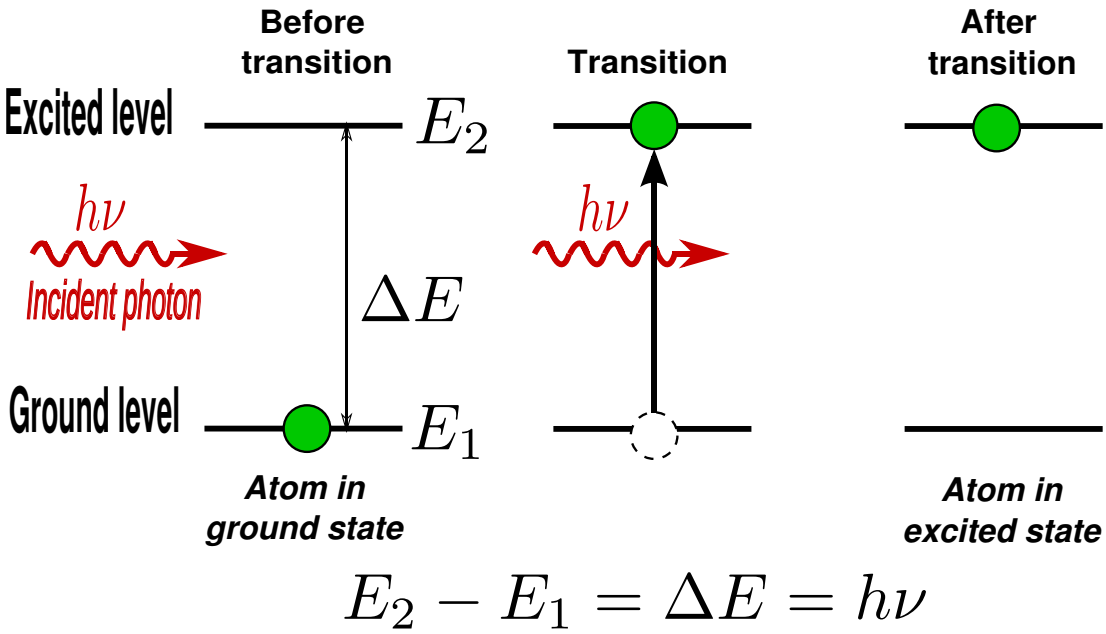

Figure 2.3: Stimulated absorption from energy state E_1 up to energy state E_2. Adopted from http://upload.wikimedia.org/wikipedia/commons /0/09/Stimulated_ Emission.svg. Licensed under the "GNU Free Documentation License".

According to Dopita and Sutherland (2003, pp.17-18), the transition probability A between two levels n, l and n', l' is described by the overlap between each level's wave function. We thus have:

$$A \propto \int \Psi_{nl} \mathbf{r} \Psi'_{n'l'} d\mathbf{r} \tag{2.7}$$

where $\mathbf{r}$ is the position vector.

As we will be dealing purely with hydrogen atoms containing only one electron, we can describe the atom completely through the radial wave function, $R(nl)$, as shown in Brocklehurst (1971, p.474):

$$A_{nl,n'l'} = 2.6674 \cdot 10^9 Z^4 a_{nl,n'l'} \tag{2.8}$$

$$a_{nl,n'l'} = \left(\frac{1}{n'^2} - \frac{1}{n^2}\right)^3 \frac{\max(l, l')}{2l+1} |\rho(n'l', nl)|^2 \tag{2.9}$$

$$\rho(n'l', nl) = \int_0^\infty R(n'l') r R(nl) dr \tag{2.10}$$

where $R(nl) = \Psi_{nl}$.

Furthermore, we are able to solve the integral (2.10) through the use of hypergeometric functions, following Gordon (1929). We use the expression

given in Dopita and Sutherland (2003, p.18) as this is the easiest to follow:

$$|\rho(n',l',n,l)|^2 = [c(n,n',l)H(n,n',l)]^2 \qquad \text{if } l' = l-1 \tag{2.11}$$

$$= [c(n',n,l')H(n',n,l')]^2 \qquad \text{if } l' = l+1 \tag{2.12}$$

where $\rho(n'l',nl) = \rho(n',l',n,l)$. Note that Brocklehurst (1971) uses different notation from Dopita and Sutherland (2003).
We then have:

$$c(n,n',l) = \frac{(-1)^{n'-l}}{4(2l-1)!} \times \sqrt{\frac{(l+n'-1)!(l+n)!}{(n'-l)!(n-l-1)!}} \tag{2.13}$$

$$\times \frac{(4nn')^{l+1}}{(n'+n)^{n'+n}} \times (n-n')^{n+n'-2l-2} \tag{2.14}$$

and:

$$H(n,n',l) = {}_2F_1\left(-n+l+1,-n'+l,2l,\frac{-4nn'}{(n-n')^2}\right) \tag{2.15}$$

$$-\frac{(n-n')^2}{(n+n')^2} \times {}_2F_1\left(-n+l-1,-n'+l,2l,\frac{-4nn'}{(n-n')^2}\right)$$

The function ${}_2F_1(\alpha,\beta,\gamma,\chi)$ is the hypergeometric function which is used to solve many linear second-order ordinary differential equations. In the case of (2.15), we note that for both expressions involving α and β, we have:

$$\alpha \leq 0 \text{ and } \beta \leq 0 \Rightarrow \alpha \text{ and } \beta \text{ are } \textit{non-positive integers} \tag{2.16}$$

and γ and χ are real numbers. Hence we define the hypergeometric function in the regular fashion:

$${}_2F_1(\alpha,\beta,\gamma,\chi) = \sum_{n=0}^{\infty} \frac{(\alpha)_n(\beta)_n}{(\gamma)_n}\frac{\chi^n}{n!} = \sum_{n=0}^{\min(|\alpha|,|\beta|)} \frac{(\alpha)_n(\beta)_n}{(\gamma)_n}\frac{\chi^n}{n!} \tag{2.17}$$

where $(q)_n$ is the Pochhammer symbol

$$(q)_n = \begin{cases} 1 & \text{if } n = 0 \\ q(q+1)\cdots(q+n-1) & \text{if } n > 0 \end{cases} \tag{2.18}$$

(Knuth, 1992).

As both α and β are non-positive integers, it follows that (2.17) will converge in a finite number of steps as indicated by substituting the upper limit of ∞ with $\min(|\alpha|, |\beta|)$.

Note that the hypergeometric function in equation (2.15) is for the general case where both n and l are considered. When l is not considered, the two first arguments to both hypergeometric functions have $l = 0$. The third argument has $l = \frac{1}{2}$ (Menzel & Pekeris, 1935).

As we are dealing with large quantum numbers in this thesis (1000+) it becomes evident that the numbers present in the expression for ρ (equations (2.11)-(2.12)) will become very large, due to the many exponents and factorials involving n. As such it is a fair assumption that double precision arithmetic will not suffice when solving ρ for large values of n, and even for moderate n. E.g. if calculating for $n = 165, n' = 155, l = 1$, we have one of the exponents equal to $n + n' - 2l - 2 = 165 + 155 - 2 - 2 = 316$. This leads to the last term in $c(n, n', l)$ being equal to $(165 - 155)^{311} = 10^{311}$ - three orders of magnitude larger than the highest representable number of an IEEE 754 64-bit binary double-precision floating-point number. Clearly there is a need for a standard that is able to represent numbers of a sufficient degree. IEEE 754 128-bit binary quadruple-precision allows for exponents up to order 10^{6111}. However, the third term in $c(n, n', l)$ has the denominator: $(n' + n)^{n'+n}$ which can be of order $\sim 10^{6600}$ for n and $n' \simeq 1000$. Finally, as can be seen in Fig. 2.4, when $n = 1000, n' = 514$ and we don't consider l, the final sum for the hypergeometric function is off by a factor of 10^{26} for quadruple-precision and 10^{43} for double-precision.

2.3 Insufficient Precision Problem

As was shown in Fig. 2.4 there was a clear discrepency in the final sum of the hypergeometric function for different precisions when considering $n = 1000, n' = 514$. This was established by use of the arbitrary precision package *MPFR* (Fousse, Hanrot, Lefèvre, Pélissier, & Zimmermann, 2007) which allows for theoretically infinite precision - well above the maximum 128-bit precision as set by the IEEE 754 standard. In Fig. 2.4 we calculated the hypergeometric function using multiples of 53-bits precision up to 6 times (except quadruple which is 113 bits). The 53-bits refer to the precision in the mantissa of the

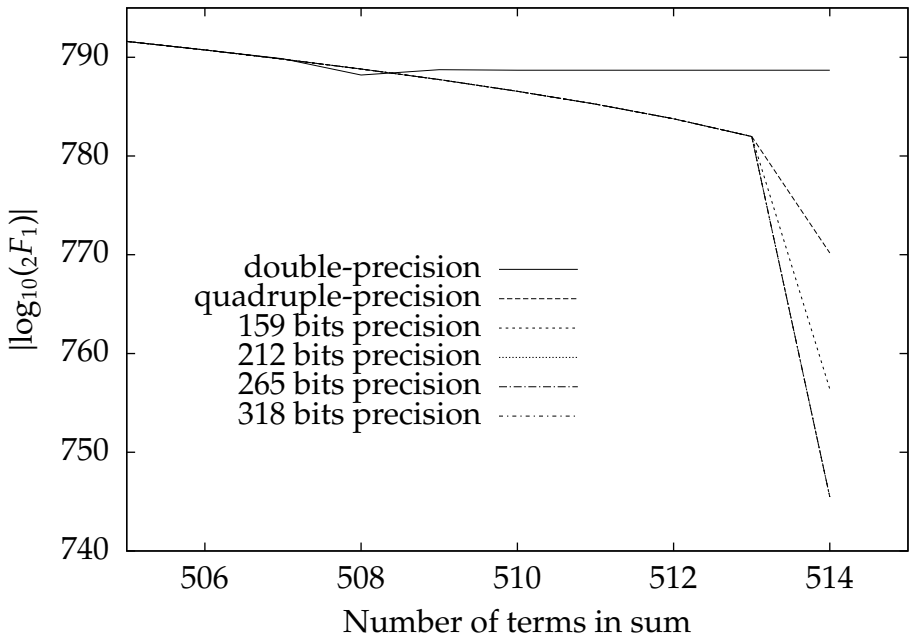

Figure 2.4: Plot of absolute value of the sum of the hypergeometric function for $n = 1000, n' = 514$ and l not considered, as a function of bits of precision. Note that the y-axis is in base 10 logarithmic scale.

number and not to the overall bits used in representing the number. In fact, as mentioned earlier, it would not be possible to represent the number $\sim 10^{788}$ using standard IEEE 754 double-precision as the exponent is only 11 bits which, when accounting for the offset-binary, gives a maximum exponent of $2^{1023} \simeq 10^{308}$ ("IEEE Standard for Floating-Point Arithmetic", 2008, p.8). As such, when using the *MPFR* package we are able to establish the correlation between number of bits in the significand vs. difference in value of the hypergeometric function. It is clear from Fig. 2.4 that in this particular example convergence happens at approximately 212 bits of precision - almost twice the precision as given by IEEE 754 quadruple precision ("IEEE Standard for Floating-Point Arithmetic", 2008, p.8).

To see the effect of limited precision in calculating the hypergeometric function, the oscillator strength transitions from $n = 1000$ to n', where $n' \in \{1, 2, \cdots, 999\}$ have been plotted in Fig. 2.5 and l is not considered. It is evident that there is a discrepancy between 185-bit precision and 250-bit precision. The plot for the 250-bit precision strength is what we would expect. Physically, the reason for this can be understood through the overlap of the wave functions

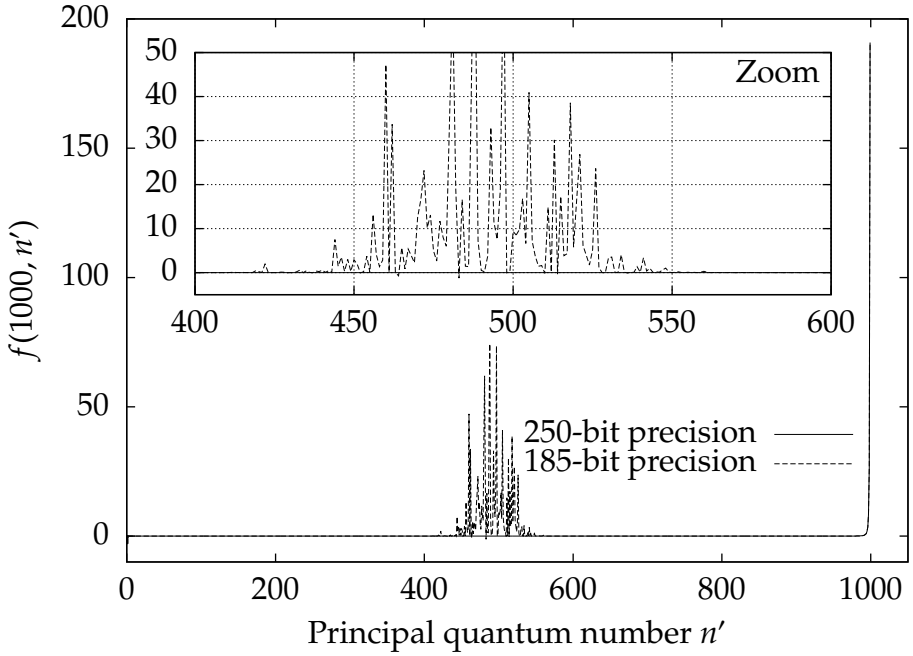

Figure 2.5: Plot of oscillator strength transitions, f, from $n = 1000$ to n', where $n' \in \{1, 2, \cdots, 999\}$

between two different states of hydrogen. The overlap from a state n to a state $n-1$ is larger than that from the state n to the state $n-2$. We know this because, when only considering n, the radius, r, of hydrogen is given as a function of the Bohr radius:

$$r = n^2 a_0 \tag{2.19}$$

This radius signifies the distance from the nucleus at which the electron is most likely to be. The radial probability function for hydrogen is peaked at this radius (Serway & Jewett, 2008, p.1230). Hence the overlap of two probability density functions at different n is greater when $\Delta n = n - n'$ is smaller. We therefore expect a strictly increasing function as that shown for 250-bits precision, whereas that shown for 185-bits precision is un-physical.

2.4 Solution to Insufficient Precision Problem for n using Gaunt Factors

Computing the hypergeometric function for each pair of n, n' when calculating oscillator strengths is time consuming. As such, substituting the calculation

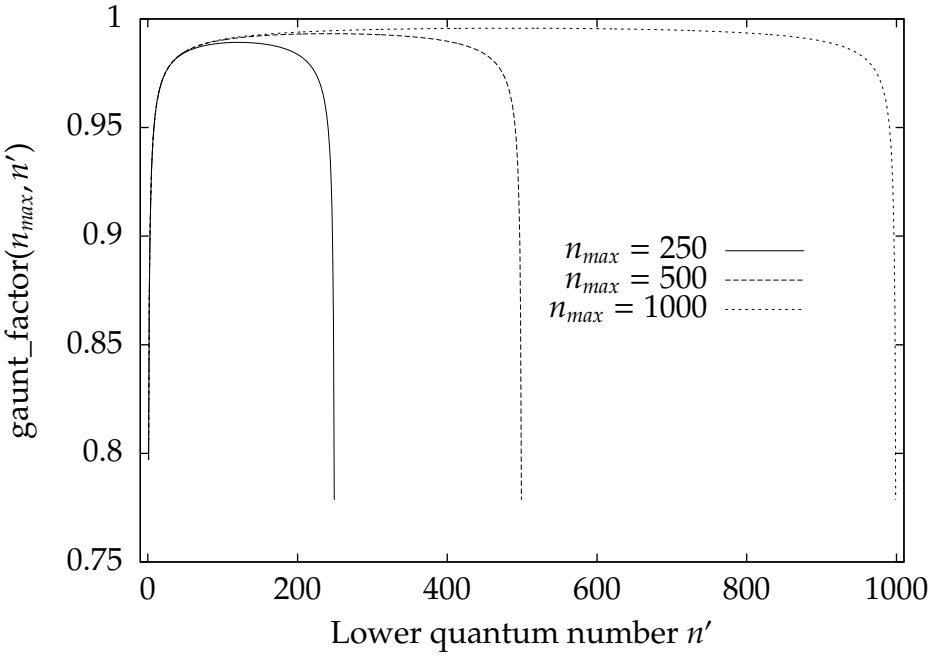

Figure 2.6: Plot of the gaunt factor as a function of maximum principal quantum number n_{max} and lower principal quantum number n'

with a simple algebraic expression would be highly desirable. For that reason Burgess and Summers (1976, p.384) expanded upon the **Gaunt factor**, ${g_{nn'}}^{\mathrm{I}}$, for transitions between bound levels, as introduced by Menzel and Pekeris (1935). They claim that this factor can correct the very simple formula for oscillator strength as given by Kramer (Menzel & Pekeris, 1935, p.84) to within 0.5%:

$$f'_{nn'} = \frac{2^6}{3\sqrt{3}\pi}\frac{1}{\omega'_n}\frac{1}{\left(\frac{1}{n'^2}-\frac{1}{n^2}\right)}\left|\frac{1}{n^3}\frac{1}{n'^3}\right| \tag{2.20}$$

where ω'_n is the statistical weight as introduced in (1.5).

In other words, the oscillator strength $f_{nn'}$ can be presented as the product of Kramer's $f'_{nn'}$ and the Gaunt factor, $g^I_{nn'}$:

$$f_{nn'} = g^I_{nn'} f'_{nn'} \tag{2.21}$$

The analytic presentation of the Gaunt factor, for any n and n', is given in Burgess and Summers (1976, p.384) as follows:

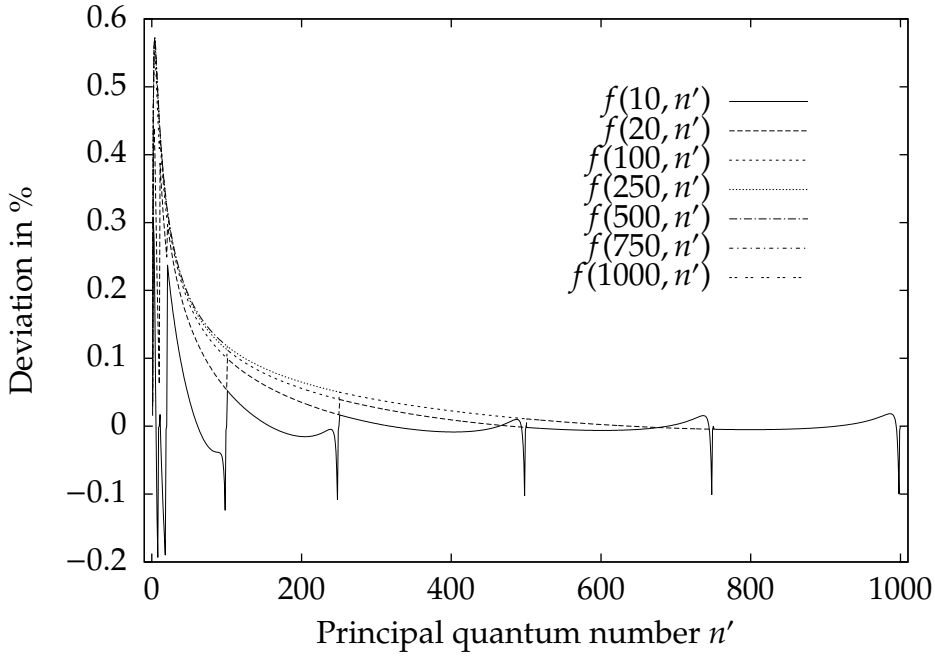

Figure 2.7: Plot of deviation in percent between the real oscillator strength, f, using hypergeometric functions with 300 bits precision, and the oscillator strength given by equation (2.21).

$$g^I_{nn'} \simeq 1.0 - T_4(T_1G_1 + T_2G_2 + T_3G_3) \tag{2.22}$$

where, for $n' < n$,

$$G_1 = \left(0.203 + \frac{0.256}{n^2} + \frac{0.257}{n^4}\right)n \tag{2.23}$$

$$G_2 = 0.170n + 0.18 \tag{2.24}$$

$$G_3 = \left(0.2214 + \frac{0.1554}{n^2} + \frac{0.370}{n^4}\right)n \tag{2.25}$$

$$T_1 = (2n' - n)(n' - n + 1) \tag{2.26}$$

$$T_2 = 4.0(n' - 1)(n - n' - 1) \tag{2.27}$$

$$T_3 = (2n' - n - 0.001)(n' - 0.999) \tag{2.28}$$

$$T_4 = \frac{1}{(n - 1.999)^2}\frac{1}{nn'^{2/3}}\left(\frac{n-1}{n-n'}\right)^{2/3} \tag{2.29}$$

The Gaunt factor, computed according to the approximation (2.22)-(2.29), changes between 0 and 1 as shown in Fig. 2.6.

In Fig. 2.7 we have plotted the deviation in percent between the real oscillator strength calculated using the hypergeometric function with 300 bits precision compared to that of using equation (2.20) multiplied by the gaunt factor. It is clear that the claimed deviation of 0.5% is a realistic estimate. An interesting side-note is that for the transition $f(2,1)$ the deviation is -44.2%. However, this is a Case A example as mentioned earlier which we will not deal with in this thesis and the transition $f(3,1)$ does not exhibit this behaviour.

2.5 Solution to the Insufficient Precision Problem for n, l

When considering quantum orbital momentum l, there does not exist an approximation formula for the Gaunt factor to correct for approximate estimates of the oscillator strength. This becomes a serious computation issue. We will now compare the run time of pure n calculations with that of n, l calculations by showing the complexity for n and n, l respectively.

In equation (1.26), instead of having infinity as an upper limit we choose a finite number, n_{max}. It is clear that each level n must then have n_{max} number of A terms in equation (1.26). This equation also shows the symmetry between the A_{mn} and A_{nm} terms. Furthermore, it very quickly becomes evident that these terms are not unique. Representing each A_{nm} term in a matrix demonstrates symmetry about the diagonal, with the diagonal itself being equal to zero. The matrix below shows this for $n_{max} = 4$:

n	1	2	3	4
1	0	A_{21}	A_{31}	A_{41}
2	A_{21}	0	A_{32}	A_{42}
3	A_{31}	A_{32}	0	A_{43}
4	A_{41}	A_{42}	A_{43}	0

Table 2.1: Matrix demonstrating the symmetry of spontaneous radiation, for n only, up to $n = 4$.

In Table 2.1 we have an upper/lower diagonal matrix, with diagonal zero. We note that each level has $n - 1$ downward transitions. Hence we have, for

the number of unique spontaneous transitions, $S_{n_{max}}$:

$$S_{n_{max}} = \sum_{n=1}^{n_{max}} n - 1 = \frac{n_{max}(n_{max} - 1)}{2} = \frac{n_{max}^2 - n_{max}}{2} \tag{2.30}$$

When computing the number of unique terms including l, we must obey the rules of quantum mechanics known as selection rules. These state that a transition from a higher state n, l to a lower state n', l', must obey the rule:

$$\Delta l = l - l' = \pm 1 \tag{2.31}$$

where the usual definition of l states that for a given n, we have $l \in \{0, 1, 2, ..., n-1\}$

n,l	$1,0$	$2,0$	$2,1$	$3,0$	$3,1$	$3,2$	$4,0$	$4,1$	$4,2$	$4,3$
$1,0$	0	0	$A_{2,1;1,0}$	0	$A_{3,1;1,0}$	0	0	$A_{4,1;1,0}$	0	0
$2,0$	0	0	0	0	$A_{3,1;2,0}$	0	0	$A_{4,1;2,0}$	0	0
$2,1$	$A_{2,1;1,0}$	0	0	$A_{3,0;2,1}$	0	$A_{3,2;2,1}$	$A_{4,0;2,1}$	0	$A_{4,2;2,1}$	0
$3,0$	0	0	$A_{3,0;2,1}$	0	0	0	0	$A_{4,1;3,0}$	0	0
$3,1$	$A_{3,1;1,0}$	$A_{3,1;2,0}$	0	0	0	0	$A_{4,0;3,1}$	0	$A_{4,2;3,1}$	0
$3,2$	0	0	$A_{3,2;2,1}$	0	0	0	0	$A_{4,1;3,2}$	0	$A_{4,3;3,2}$
$4,0$	0	0	$A_{4,0;2,1}$	0	$A_{4,0;3,1}$	0	0	0	0	0
$4,1$	$A_{4,1;1,0}$	$A_{4,1;2,0}$	0	$A_{4,1;3,0}$	0	$A_{4,1;3,2}$	0	0	0	0
$4,2$	0	0	$A_{4,2;2,1}$	0	$A_{4,2;3,1}$	0	0	0	0	0
$4,3$	0	0	0	0	0	$A_{4,3;3,2}$	0	0	0	0

Table 2.2: Matrix demonstrating the symmetry of spontaneous radiation, for n and l, up to $n = 4$.

Table 2.2 presents the matrix that arises when following these selection rules for $n_{max} = 4$. Taking each pair of quadrants reflected in the diagonal, it is easy to see that these are symmetric. We note that for each level n, there are $(n-1)^2$ downward transitions which can be seen from counting the amount of non-zero terms appearing in either the upper or lower diagonal, for each level. As

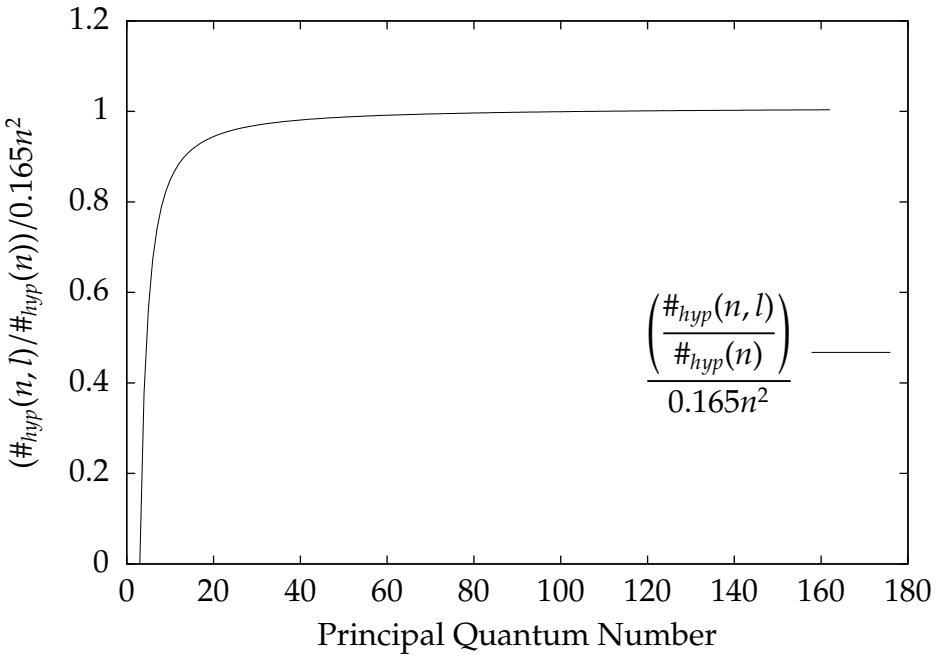

Figure 2.8: This plot shows the magnitudes of difference between $x = n$ or $x = n, l$ for the function #hyp(x).

such we have term $S_{n_{max},l}$ equal to:

$$S_{n_{max},l} = \sum_{n=1}^{n_{max}} (n-1)^2 = \sum_{n=0}^{n_{max}-1} n^2 \overset{P=n_{max}-1}{=\!=\!=} \frac{P^3}{3} + \frac{P^2}{2} + \frac{P}{6} \tag{2.32}$$

From equation (2.30) and (2.32) we have that $S_{n_{max}} \in O(n^2)$ and $S_{n_{max},l} \in O(n^3)$, where O is the "Big O" notation.

Although we have established the number of terms in $S_{n_{max}}$ and $S_{n_{max},l}$ we must also analyse them with respect to the amount of terms in the hypergeometric function as this will have a very large impact on the overall runtime. For this purpose s_{d}we define the function:

$$\#_{\mathrm{hyp}}(x) \tag{2.33}$$

as the total number of unique terms in the hypergeometric function for a given x, where $x = n$ or $x = n, l$.

In Fig. 2.8 we have plotted the function:

$$\frac{\left(\frac{\#_{\text{hyp}}(n,l)}{\#_{\text{hyp}}(n)}\right)}{0.165n^2} \tag{2.34}$$

where the quadratic, $0.165n^2$, was established through manual curve fitting. As can be seen, the plot quickly becomes constant. Hence the difference in runtime for n vs. n, l is a whole magnitude larger than what we predicted when using equation (2.30) and equation (2.32).

We determined $\#_{\text{hyp}}(n,l)$ and $\#_{\text{hyp}}(n)$ by activating a counter in the `einstein_coefficient_calc_mpfr.c` program.

2.6 Implementation

For computation of the Einstein coefficients, including orbital quantum number l, we rely here on the very extensive paper by Brocklehurst (1971). However, as noted earlier, we have used Dopita and Sutherland (2003) for the description of the radial function, as this definition is more clear. Furthermore, they observe that this integral can in fact always be expressed as a rational number since the square root in equation (2.13) and (2.14) will eventually be squared. As such, one could technically avoid performing this square root. However, for simplicity, we perform the square root. Secondly, in a case where $n = 1000, n' = 999, l = 998$, the fraction of the square root term in (2.13) would look like this:

$$\frac{(l+n'-1)!(l+n)!}{(n'-l)!(n-l-1)!} = \frac{(998+999-1)!(998+1000)!}{(999-998)!(1000-998-1)!} \tag{2.35}$$

$$= \frac{1996!1998!}{1!1!} \tag{2.36}$$

$$= 1996!1998! \tag{2.37}$$

$$= 1996!1996! \cdot 1998 \cdot 1997 \tag{2.38}$$

$$= (1996!)^2 \cdot 1998 \cdot 1997 \tag{2.39}$$

$$\simeq (2.07 \cdot 10^{5722})^2 \cdot 4 \cdot 10^6 \tag{2.40}$$

$$\simeq 4 \cdot 10^{11444} \cdot 4 \cdot 10^6 \tag{2.41}$$

$$\simeq 1.6 \cdot 10^{11451} \tag{2.42}$$

This is a very large number but by taking the square root we reduce the size of it enough to allow us to perform calculations on it. The term will then subsequently be cancelled out by other very small terms (which arise from the sizes of n, n' and l').

When n, n' are large and l is small, the fraction of the square root becomes close to unity. Similarly, the remaining terms also cancel. For small n, all fractions become close to unity as n is the largest of all the quantum numbers.

2.6.1 The *MPFR* Library

The program for computing the Einstein Coefficients was implemented in C with the additional *MPFR* library to support arbitrary precision. This library does not extend previously defined C functions to allow for arbitrary arithmetic but rather defines its own arithmetic functions. Fig. 2.9 explains the general structure of reading code containing *MPFR* functions.

The *MPFR* library is built on top of the *GMP* library which is widely used when dealing with arbitrary precision. In fact, the GCC compiler uses the *MPFR* library, which uses the *GMP* library, to evaluate built-in maths functions at compile time ("GCC 4.3 Release Series Changes, New Features, and Fixes", 2013). Fousse et al. (2007, p.9) have compared the run time of the *MPFR* library to that of three other widely used arbitrary precision packages, also based on *GMP*, with the results being widely in favour of *MPFR*. As such, we are confident that the *MPFR* package is a good choice. However, there are a few concerns of how some of the functions are computed when using the *MPFR* package. As the analysis of the computation pointed out, the hypergeometric function can have numerous terms. Hence it is important that the calculations are performed optimally.

Fig. 2.10 shows that the calculation of $n!$ in the *MPFR* library is performed naively as $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$. Based on the suggestion of "FastFactorialFunctions" (2013) (referred to by Black (2013)) we suggest implementing the factorial function known as *PrimeSwing*. This is in fact the algorithm implemented by the *GMP* library (Granlund & the GMP development team, 2013, p.107). Surprisingly, this implementation is not offered by the *MPFR* library.

When taking the square root the *MPFR* library is implemented identically

```
// Set default MPFR precision to 300
mpfr_set_default_prec(300);

// Declares variable
mpfr_t temp1, temp2, temp3;

// Initialise variable
mpfr_init(temp1);
mpfr_init(temp2);

// Initialise and set precision of variable
mpfr_init2(temp3,350);

// Assign values to variables
// temp1 = 10.0 with 'MPFR_RNDN' rounding
// temp2 = 20 with 'MPFR_RNDN' rounding
// temp3 = temp2 with 'MPFR_RNDN' rounding
mpfr_set_d(temp1,10.0,MPFR_RNDN);
mpfr_set_ui(temp2,20,MPFR_RNDN);
mpfr_set(temp3,temp2,MPFR_RNDN);

// Do arithmetic on variables:
// temp3 = temp2*temp1 with 'MPFR_RNDN' rounding
mpfr_mul(temp3,temp2,temp1,MPFR_RNDN);

// temp1 = temp1-temp2 with 'MPFR_RNDN' rounding
mpfr_sub(temp1,temp1,temp2,MPFR_RNDN);

// temp3 = 5/temp2 with 'MPFR_RNDN' rounding.
// 5 is an 'unsigned long int'
mpfr_ui_div(temp3,5,temp2,MPFR_RNDN);

// temp3 = temp2/5 with 'MPFR_RNDN' rounding.
// 5 is an 'unsigned long int'
mpfr_ui_div(temp3,temp2,5,MPFR_RNDN);

// Get double value of 'temp3' with MPFR_RNDN rounding.
double answer = mpfr_get_d(temp3,MPFR_RNDN);

mpfr_clear(temp1);
mpfr_clear(temp2);
mpfr_clear(temp3);
```

Figure 2.9: General usage of the *MPFR* library.

```
mpfr_t t;     /* Variable of Intermediary Calculation*/
unsigned long i;
int round;
...
for (i = 2 ; i <= x ; i++)
{
  round = mpfr_mul_ui (t, t, i, rnd);
  ...
}
```

Figure 2.10: Naive implementation of $n!$ in `factorial.c` from the *MPFR* library.

to the *GMP* library (The MPFR Team, 2013, p.13), using the "Karatsuba Square Root" (Granlund & the GMP development team, 2013).

For exponentiation (`pow`) the *MPFR* library makes use of the identity $\log(x^y) = y\log(x)$ by defining the function in terms of `exp` as such:

$$\text{pow}(x, y) = x^y = e^{y\log(x)} = \exp(x, y) \tag{2.43}$$

This requires the calculation of a `mul` function along with an `exp` and `log` function. However, all three of these functions are implemented with focus on optimization (unlike the factorial function).

2.7 GPU Optimization

Based on the analysis of the computation of the radial integral given in equation (2.10), optimization will be an important factor. For this, use of parallel computation techniques such as GPUs will be highly desirable. At the time of implementation, the author was not familiar with any arbitrary precision packages available for GPUs and as such no attempt has been made to implement arbitrary precision on the GPU. However, both Nakayama and Takahashi (2011) and Lu, He, and Luo (2010) have demonstrated significant performance gains by implementing arbitrary arithmetic, also based on *GMP*, on the GPU as opposed to running those same calculations on the CPU using a *GMP* based arbitrary arithmetic package. It should be noted though that Fig. 2.4 indicates that arbitrary arithmetic may not be needed for a large number of terms

in the hypergeometric series, if we exclude the magnitude of the exponent, as precision only decreases in the last 8 terms. Hence there may be a way of calculating using double-precision on the GPU, and then returning to the CPU for the remaining 8 calculations. However, the relatively small size of the double-precision mantissa must be dealt with if this approach is taken, such as using logarithms.

2.8 Conclusion

Lastly, it should be noted that the nature of the departure coefficient problem is dependent on temperature and density, as mentioned earlier. As the Einstein coefficients are purely dependent on the atomic states of the atom, and independent of temperature and density, one can pre-calculate all Einstein coefficients and simply reload the values when solving the $b_{n,l}$ problem. Furthermore, there is no dependence on previous calculations of n, l, i.e. if calculating all the Einstein coefficients for $n_{max} = 1000$, there is no need to recompute these values when continuing calculation for $n_{max} > 1000$. The program included with this thesis does exactly this and can in a matter of seconds reload from a (large) file all the previously calculated Einstein coefficients. Even when changing the precision one can still use pre-calculated values and as such the reduced performance of calculating the Einstein coefficients on the CPU rather than on the GPU will not be an obstacle for the calculations of $b_{n,l}$ for any temperature. In Fig. 2.11 we show the calculation of all Einstein coefficients up to $n_{max} = 1000$ for different bits of precision.

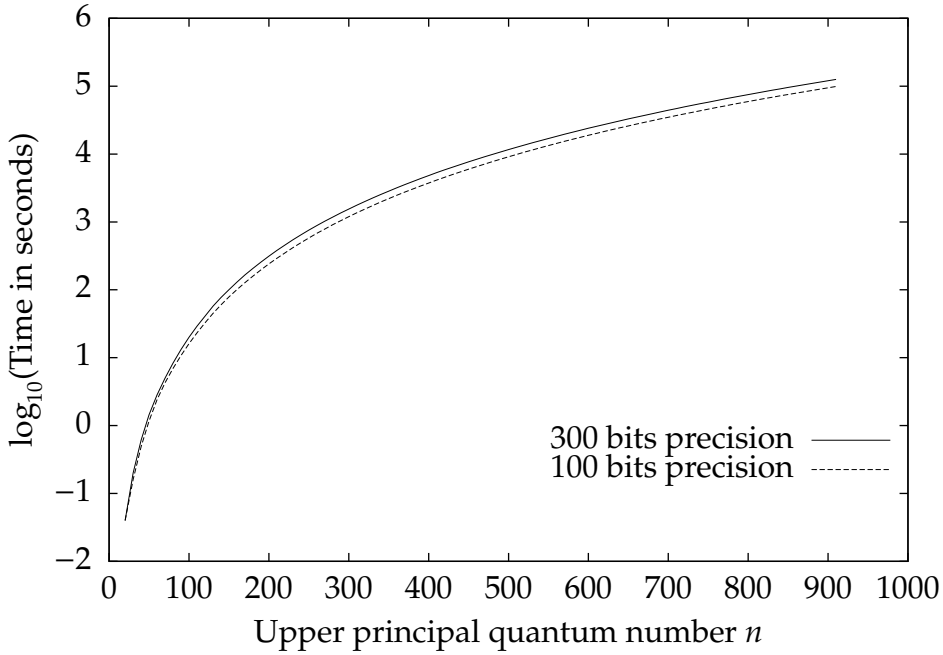

Figure 2.11: Plot of time to calculate Einstein coefficients as a function of upper principal quantum number n in $\log_{10}$ scale.

Chapter 3

Radiative Recombination

3.1 Introduction to Radiative Recombination

In this chapter we introduce the concept of radiative recombination. This process takes place in plasmas where the electrons are separated from atoms (who have thus become ions) and are said to be in a "free" state. As ions can accept electrons and the electrons are now free, an ion can capture an electron. When this occurs, the electron is said to *recombine* with the ion. After recombining, the electron can transition up, down or become ionized according to the processes given in equations (1.18)-(1.22).

In Chapter 4, equation (4.36), we derive the expression for n of the radiative recombination coefficient α_n. We will not do that here and simply state the result:

$$\alpha_n = 5.197 \times 10^{-14} x_n^{3/2} S_0(x_n) \tag{3.1}$$

where

$$x_n = \frac{15.789 \times 10^4}{T_e n^2} \tag{3.2}$$

and

$$S_0(x) = e^x \int_x^\infty \frac{e^{-v}}{v} dv \tag{3.3}$$

and T_e is the electron temperature.

Although this involves an exponential integral, we only have to do n numerical integrations and so α_n does not become an intensive computational task and this calculation is well understood. Hence we progress to the calculation of α_{nl} in the following section.

3.2 Calculation of Radiative Recombination Coefficients

When including orbital momentum l, the equation for α_n (now α_{nl}) becomes increasingly more complex as, which was the case for Einstein coefficients, we must calculate the hypergeometric function. However, we will this time make use of a second approach, relying on a recursive scheme developed by Burgess (1965).

In order to derive an expression for α_{nl} we use the fact that radiative recombination and photoionization can be related by the Milne relation and are in fact inverse processes (see Chapter 4). When photoionization occurs, the electron is ejected with a dimensionless energy of k^2 which obeys the energy conservation condition:

$$h\nu = \left(\frac{1}{n^2} + k^2\right) I_H \tag{3.4}$$

where I_H is energy needed to ionize an electron in its ground state. The cross section for photoionization is then given by:

$$a_{nl}(k^2) = \left(\frac{4\pi\alpha a_0^2}{3}\right) n^2 \sum_{l'=l\pm1} \frac{\max(l,l')}{2l+1} \Theta(n,l,\kappa,l') \tag{3.5}$$

where

$$\Theta(n,l,\kappa,l') = (1+n^2\kappa^2)^2 |g(n,l,\kappa,l')|^2 \tag{3.6}$$

and

$$g(n,l,\kappa,l') = \frac{1}{n^2} \int_0^\infty \Psi_{nl} \mathbf{r} \Psi'_{\kappa l} d\mathbf{r} \tag{3.7}$$

As $\kappa \equiv k/Z$, where $Z = 1$ for hydrogen, we have made a direct substitution for k with κ. As in Chapter 2 the overlap of the wave functions in (3.7) represents the transition probability from the state n,l to the state κ,l, where κ,l is the ionization state with energy that obeys equation (3.4). Following Burgess (1965) we now define:

$$\alpha_{nl} = \frac{2\pi^{1/2}\alpha^4 a_0^2 c}{3} \frac{2y^{1/2}}{n^2} \sum_{l'=l\pm1} I(n,l,l',t) \tag{3.8}$$

where

$$I(n,l,l',t) = \max(l,l') y \int_0^\infty (1+n^2\kappa^2)^2 \Theta(n,l,\kappa,l') e^{-\kappa^2 y} d(\kappa^2) \tag{3.9}$$

and

$$t = \frac{T_e}{10^4} \tag{3.10}$$

and

$$y = \frac{Rhc}{kT_e} \simeq \frac{15.789}{t} \tag{3.11}$$

Similar to the Einstein coefficients we are able to get an exact expression for $g(n, l, \kappa, l \pm 1)$, stated by Burgess (1965) as:

$$g(n,l,\kappa,l') = \sqrt{\frac{\pi}{2}\frac{(n+1)!}{(n-l-1)!(1-e^{-2\pi/\kappa})}\prod_{s=0}^{l'}(1+s^2\kappa^2)} \times \left(\frac{4n}{1+n^2\kappa^2}\right)^{\min(l,l')} \times \frac{\exp\left[-\frac{2}{\kappa}\tan^{-1}(n\kappa)\right]}{4n^2(2l\pm1)!}\Upsilon_{\pm} \tag{3.12}$$

where

$$\Upsilon_{+} = i\eta\left(\frac{n-i\eta}{n+i\eta}\right)^{n-l}\left[{}_2F_1\left(l+1-n, l-i\eta, 2l+2, \frac{-4ni\eta}{(n-i\eta)^2}\right) - \left(\frac{n+i\eta}{n-i\eta}\right)^2 \times {}_2F_1\left(l+1-n, l+1-i\eta, 2l+2, \frac{-4ni\eta}{(n-i\eta)^2}\right)\right] \tag{3.13}$$

and

$$\Upsilon_{-} = i\eta\left(\frac{n-i\eta}{n+i\eta}\right)^{n-l-1}\left[{}_2F_1\left(l-1-n, l-i\eta, 2l, \frac{-4ni\eta}{(n-i\eta)^2}\right) - \left(\frac{n+i\eta}{n-i\eta}\right)^2 \times {}_2F_1\left(l+1-n, l-i\eta, 2l, \frac{-4ni\eta}{(n-i\eta)^2}\right)\right] \tag{3.14}$$

and $\eta = 1/\kappa$ and $i^2 = -1$. There are a few major differences between the expression for $g(n, l, \kappa, l')$, given by equation (3.12) and the expression for $\rho(n'l', nl)$, given by equation (2.11). These originate in the radial wave equation for $\Psi_{\kappa l}$ because the atom is ionized rather than transitioning to another bound state. The equation for $\Psi_{\kappa l}$ includes the term $\pi/2$ and $\prod_{s=0}^{l}(1+s^2\kappa^2)$ which is why we see variants of these terms appearing in equation (3.12). It is for this same reason that some of the parameters to the hypergeometric function are now

complex.

As mentioned at the start of this section, we will be implementing a second technique that uses a recursive scheme by Burgess (1965) to solve $g(n, l, \kappa, l')$ rather than using hypergeometric functions. As the reason behind this recursive scheme is rather involved we simply restate the equations and refer the reader to Burgess (1965):

$$g(n, l, \kappa, l') = \sqrt{\frac{(n+l)!}{(n-l-1)!} \prod_{s=0}^{l'} (1 + s^2\kappa^2)} (2n)^{l-n} G(n, l, \kappa, l') \tag{3.15}$$

where

$$G(n, n-1, 0, n) = \sqrt{\frac{\pi}{2}} \frac{8n}{(2n-1)!} (4n)^n e^{-2n} \tag{3.16}$$

$$G(n, n-1, \kappa, n) = \frac{1}{\sqrt{1 - e^{-2\pi/\kappa}}} \frac{\exp\left[2n - 2/\kappa \tan^{-1}(n\kappa)\right]}{(1 + n^2\kappa^2)^{n+2}} \tag{3.17}$$

$$\times G(n, n-1, 0, n) \tag{3.18}$$

$$G(n, n-2, \kappa, n-1) = (2n-1)(1 + n^2\kappa^2) n G(n, n-1, \kappa, n) \tag{3.19}$$

$$G(n, n-1, \kappa, n-2) = \left(\frac{1 + n^2\kappa^2}{2n}\right) G(n, n-1, \kappa, n) \tag{3.20}$$

$$G(n, n-2, \kappa, n-3) = (2n-1)\left[4 + (n-1)(1 + n^2\kappa^2)\right] G(n, n-1, \kappa, n-2) \tag{3.21}$$

Furthermore, we have the following recurrence relations:

$$\begin{aligned} G(n, l-2, \kappa, l-1) = {} & \left[4n^2 - 4l^2 + l(2l-1)(1 + n^2\kappa^2)\right] G(n, l-1, \kappa, l) \\ & - 4n^2(n^2 - l^2)\left[1 + (l+1)^2\kappa^2\right] G(n, l, \kappa, l+1) \end{aligned} \tag{3.22}$$

$$\begin{aligned} G(n, l-1, \kappa, l-2) = {} & \left[4n^2 - 4l^2 + l(2l+1)(1 + n^2\kappa^2)\right] G(n, l, \kappa, l-1) \\ & - 4n^2\left[n^2 - (l+1)^2\right](1 + l^2\kappa^2) G(n, l+1, \kappa, l) \end{aligned} \tag{3.23}$$

3.3 Implementation

It is clear from equation (3.8) that the most intensive part of the calculation will be performed calculating the value $I(n, l, l', t)$, defined in equation (3.9). Although we are not using arbitrary arithmetics for the calculation of $g(n, l, \kappa, l')$ (which was neccessary for the method used for the Einstein coefficients) we do

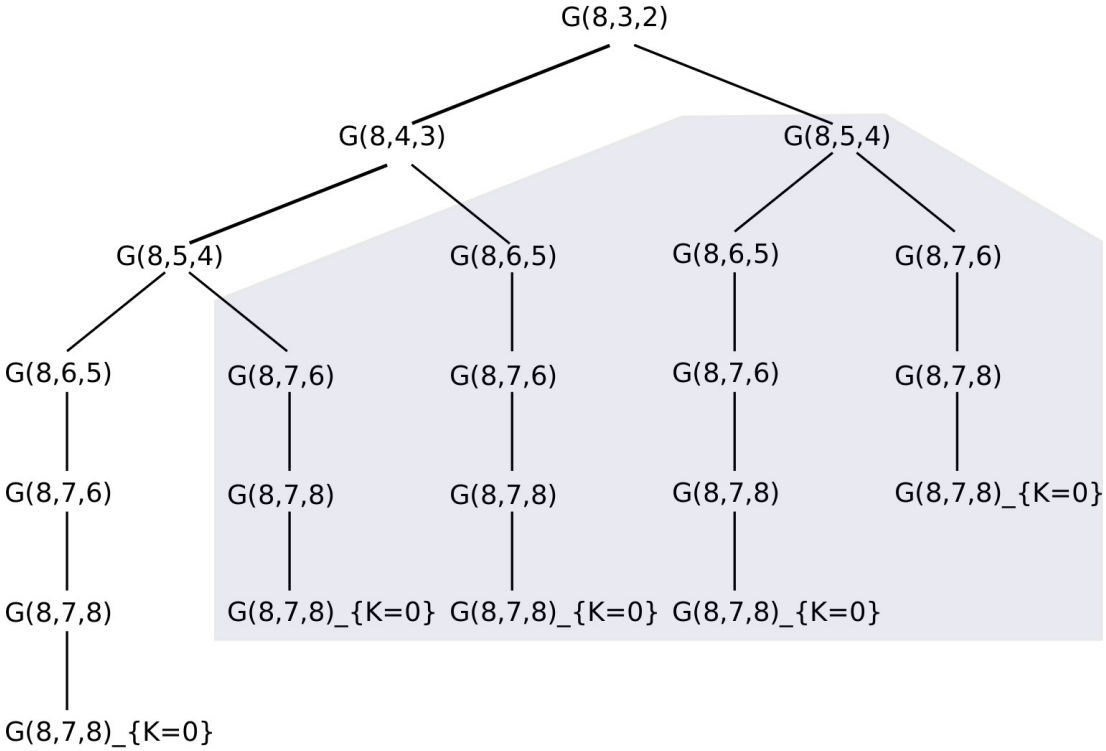

Figure 3.1: The picture shows the naive approach to solving the recurrence relation when $G(n, l, \kappa, l') = G(8, 3, \kappa, 2)$. The shaded rectangles show where duplicate calculations are being performed. κ is not shown for brevity.

use IEEE 754 128-bit binary quadruple-precision to avoid the possible errors that can build up when multiplying the highly fluctuating G terms in equations (3.16)-(3.21) (Burgess, 1965), as we calculate for very high n.

We can solve the problem naively via a top-down approach by using the recurrence relations (3.22) and (3.23). In Fig. 3.1 we show this approach for $G(8, 3, 2)$. As can be seen from the shaded rectangles, we are performing many calculations more than once. In fact, out of the 22 calculations made in all, only 7 are unique.

Instead of the naive approach, a memoization algorithm could be used (Cormen, 2009). This will still solve the algorithm in a top-down approach but will store the results obtained so that when the same result is needed, it can be returned in constant time rather than performing the computation once more. This has the advantage of completely avoiding duplicate computation but uses more memory. To avoid this, we use a bottom-up dynamic programming approach (Cormen, 2009). We can do this because the problem exhibits the two desired

criteria for a dynamic programming approach:

- Optimal substructure
- Overlapping subproblems

Optimal substructure is fulfilled because each solution to a problem contains the solution to subproblems.

Overlapping subproblems requires the size of the independent subproblem space to be significantly smaller than that of the entire problem space i.e. solving all of the independent subproblems is much less time-consuming than solving the entire problem naively. As was demonstrated in Fig. 3.1 this is clearly the case for our recurrence relation.

As both criteria are fulfilled the problem is well-suited for a bottom-up dynamic programming approach.

We now make the following observation:

For any n and l, we have $n \geq l + 1$. We assume $l' = l - 1$ and call this *Case 1*. This does not change the following argument as the concept will be the same for $l' = l + 1$. Hence we can write $l = n - 2 - s$ where $-1 \leq s \leq n - 2 - l$. If $s \leq 0$, the solution is trivial as we can match it to one of our base cases in equations (3.16)-(3.21). For $s = 0$, this requires the calculation of four terms. When $s > 0$, we define, by rewriting equation (3.23) to better suit our problem:

$$\begin{aligned} G\left[n, (n-2)-s, (n-3)-s\right] = {} & A\left[n, (n-2)-s\right] \\ & \times G\left[n, (n-2)-s+1, (n-3)-s+1\right] \\ & + B\left[n, (n-2)-s\right] \\ & \times G\left[n, (n-2)-s+2, (n-3)-s+2\right] \end{aligned} \tag{3.24}$$

From this it follows that:

$$\begin{aligned} G\,[n,(n-2)-s-1,(n-3)-s-1] = A\,[n,(n-2)-s-1] \\ \times G\,[n,(n-2)-s,(n-3)-s] \\ + B\,[n,(n-2)-s-1] \\ \times G\,[n,(n-2)-s+1,(n-3)-s+1] \end{aligned} \tag{3.25}$$

Equations (3.24) and (3.25) show that each calculation of $G(n,l,\kappa,l')$ is defined entirely by $G(n,l+1,\kappa,l'+1)$ and $G(n,l+2,\kappa,l'+2)$. An analogous result can be established for $l' = l+1$ (*Case 2*).

We can now create a program that first checks whether $G(n,l,\kappa,l')$ is in the form of any of the base case equations given in (3.16) - (3.21). If so, simply return the answer directly. If not, one of the two following cases are true:

1. $G(n,l,\kappa,l') = G(n,l,\kappa,l-1)$
2. $G(n,l,\kappa,l') = G(n,l,\kappa,l+1)$

In *Case 1*, our base cases are $G(n,n-1,\kappa,n-2)$ and $G(n,n-2,\kappa,n-3)$, both defined through relation (3.24) and (3.25), where $s=-1$ and $s=0$ respectively. Furthermore, we are able to reach any $G(n,n-2-s,\kappa,n-3-s)$ through these two base cases, where s is finite and $s>0$. In our program we therefore define h_1 and h_2 as:

$$h_1 = G(n,n-1,\kappa,n-2) \tag{3.26}$$

$$h_2 = G(n,n-2,\kappa,n-3) \tag{3.27}$$

We now define h_i according to equation (3.24):

$$h_i = A_i h_{i-1} + B_i h_{i-2} \tag{3.28}$$

As each h_i is defined only in terms of h_{i-1} and h_{i-2} we can store h_i in the place of h_{i-2}. In Fig. 3.4 we show the code for the function `G_n_l_K_lg` where `lg` means l' is less than l i.e. $l' = l-1$. Furthermore, Fig. 3.2 shows an example of the recursion tree for this new approach when $n=8, l=3$ and $l>l'$.

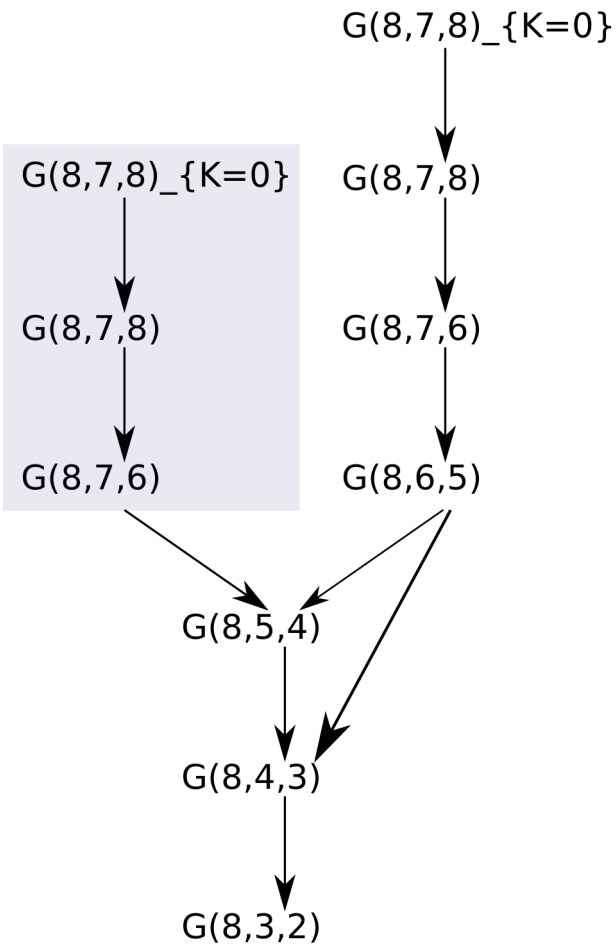

Figure 3.2: The picture shows the dynamic programming approach to solving the recurrence relation when $G(n, l, \kappa, l') = G(8, 3, \kappa, 2)$. The shaded rectangle shows where duplicate calculations are being performed. κ is not shown for brevity.

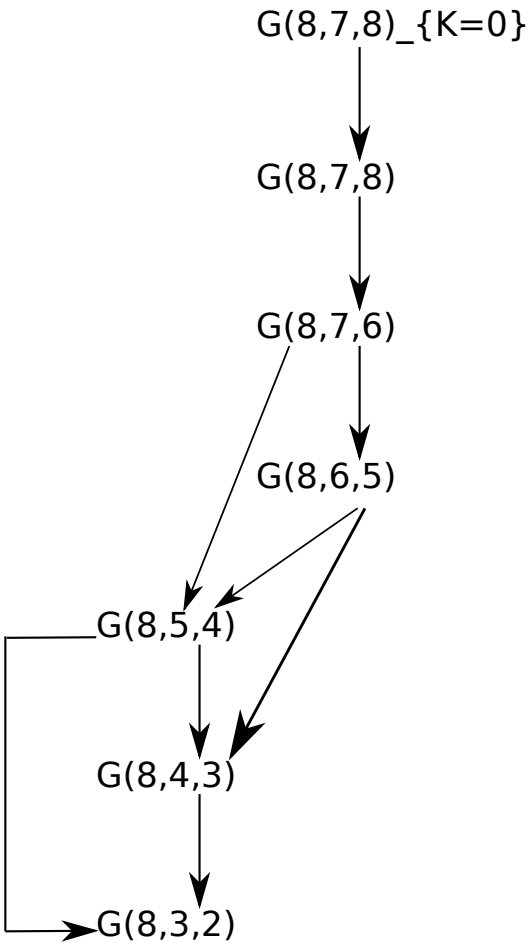

Figure 3.3: The picture shows the optimized dynamic programming approach to solving the recurrence relation when $G(n, l, \kappa, l') = G(8, 3, \kappa, 2)$. κ is not shown for brevity.

```
G_l_K_lg(n,l,K)
{
    h1 = G_n_n_1_K_n(n,K);
    h2 = G_n_n_2_K_n_1(n,K);

    for(i=3;i<=n-l;i++)
    {
        if(i%2==0)
            h1 = A*h2+B*h1;
        else
            h2 = A*h1+B*h2;
    }
    // Check whether or not last
    // returned value was h2 or h1
    if(i%2==0)
        return h2;
    else
        return h1;
}
```

Figure 3.4: `G_l_K_lg` algorithm

We note here that, initially, $i = 3$ which we will discuss shortly.

Using this scheme, we have reduced the memory needed for the algorithm compared to the previously suggested memoization algorithm, as we overwrite h_{i-2} in each iteration with the value of h_i. Furthermore, we have also eliminated the checks needed to verify whether or not a result has already been computed.

In Fig. 3.2, three of the relations are solved twice, namely $G(n, n-1, \kappa, n-2), G(n, n-1, \kappa, n)$ and $G(n, n-1, 0, n)$. However, we can easily avoid the duplicate calculations by first computing the value of $F = G(n, n-1, \kappa, n-2)$ and then setting $G(n, n-2, \kappa, n-3) = (2n-1)\left[4 + (n-1)(1+n^2\kappa^2)\right] \times F$, thus avoiding the additional three calculations. This is depicted in Fig. 3.3 where the two branches from Fig. 3.2 have now become one.

The height of the tree in Fig. 3.3 is seven. In order to get a general expression for the height of the tree, we must define it in terms of the two parameters n and l. Calculating $G\left[n, (n-2)-s-1, (n-3)-s-1\right]$ after $G\left[n, (n-2)-s, (n-3)-s\right]$ has been calculated requires one calculation following relations (3.24) and (3.25). Hence the number of terms needed to calculate $G\left[n, (n-2)-s, (n-3)-s\right]$, when $G\left[n, (n-2), (n-3)\right]$ has been calculated, is s. We mentioned earlier that calculating $G\left[n, (n-2)-s, (n-3)-s\right]$, where $s = 0$, requires four calculations. Hence to calculate $G\left[n, (n-2)-s, (n-3)-s\right]$ for any s requires $s+4$ calculations. As we defined $s = n-2-l$ we can write total number of calculations as:

$$H_{dyn_>} = s + 4 = (n-2-l) + 4 = n - l + 2 \tag{3.29}$$

which is therefore the height of the tree. In our example we have $n = 8, l = 3$ and hence the height can be written as $8 - 3 + 2 = 7$ which is the height we stated earlier.

For the naive approach, we remind the reader that we are using a top-down approach.

The height of the leftmost leaf is the same height as the tree for the dynamic programming approach. This is true because all the nodes on the (only) path to the leftmost leaf node will decrease s by one, until $s = 0$, when traversing from the node at level i to level $i+1$, where s is defined as for the dynamic programming approach and when using relations (3.24) and (3.25).

We will now show the height of the rightmost node. Following relations (3.24) and (3.25), the path to the rightmost leaf will decrease s by two when traversing from the node at level i to level $i+1$. Hence, when $n-l$ is even, s will have decreased by two exactly $\frac{n-l}{2}$ times when s becomes zero. Thus, the length of the path to the rightmost leaf node is:

$$L_{even} = \frac{n-l}{2} + 3 \tag{3.30}$$

Note that, unlike for the dynamic programming approach, we add three, not four, in our equation (3.30). This is because the naive method is a top-down approach and hence when $s = 0$ the calculation has been counted in the term for n, l. Thus we only need to compute $G(n, n-1, 0, n), G(n, n-1, \kappa, n)$ and $G(n, n-1, \kappa, n-2)$, whereas for the dynamic approach, we start at $s = 0$ and hence $G(n, n-2, \kappa, n-3)$ should be added to the the constant for the expression of n, l.

However, when $n-l$ is odd, at some point $s = 1$ and when decreasing s by two, we obtain $s = -1$. After calculating this term, we need only calculate $G(n, n-1, 0, n), G(n, n-1, \kappa, n)$. Hence the length of the path to the rightmost node will be:

$$L_{odd} = \frac{n-l-1}{2} + 3 \tag{3.31}$$

Combining equation (3.30) and (3.31) we obtain the equation:

$$L_g = \left\lfloor \frac{n-l}{2} \right\rfloor + 3 \tag{3.32}$$

where L_g is the length of the path to the rightmost leaf. L_g is also the shortest path to any leaf. Hence when subtracting three from L_g, we get the length at which the tree for the naive approach is a *perfect binary tree*. The number of nodes in a perfect binary tree of height $\left\lfloor \frac{n-l}{2} \right\rfloor$ is:

$$\left(\left\lfloor \frac{n-l}{2} \right\rfloor \right)^2 - 1 \tag{3.33}$$

In *Case 2*, where $l' = l+1$ we use the same reasoning as for $l' = l-1$ to obtain:

$$H_{dyn_s} = n - l + 1 \tag{3.34}$$

For the naive approach, when $l' = l + 1$, we get:

$$L_s = \left\lfloor \frac{n-l}{2} \right\rfloor + 2 \tag{3.35}$$

Using equation (3.29) and (3.34), we obtain a complexity of $O(n-l)$ for the dynamic approach. Using equation (3.33) and (3.35) we obtain a complexity of $O\left((n-l)^2\right)$ for the naive approach. Due to this difference in run-time we have naturally chosen to implement the dynamic programming approach.

3.3.1 Calculating $I(n, l, l', t)$

Unlike for Einstein coefficients, when calculating radiative recombination we are eventually faced with solving an integral as seen in equation (3.9). Burgess (1965) mentions how the integrand is always monotonically decreasing approximately exponentially. As such, we can calculate the integral numerically and increase the step size used as the integration is done on the interval from $0 \rightarrow \infty$, where ∞ is replaced with a sufficiently large upper value of κ^2 (the integration variable). As for Einstein coefficients, it is possible to pre-calculate all the values of $\Theta(n, l, \kappa, l')$. Thus when integrating, we need only read the values of the already calculated $\Theta(n, l, \kappa, l')$ values. Due to the nature of the integral for $I(n, l, l', t)$, it would seem favourable to calculate it using a Gauss-Laguerre method. However, this method uses a weighted step size as a function of its variables. As t is a variable in the integral, we would need evaluation of different intervals for $\Theta(n, l, \kappa, l')$ at varying values of t. Thus whenever t is changed, all values of $\Theta(n, l, \kappa, l')$ would need to be recalculated. As $\Theta(n, l, \kappa, l')$ is excessively more computationally intensive than the calculation of the integral, we need a method that does not make use of weighted step sizes as a function of its variables. Hence we use the fixed point integration method of Boole as suggested by Burgess (1965). However, we do use a significantly smaller initial step size, with more iterations, as we are dealing with significantly higher values of n and l than that of Burgess (1965). He suggests an initial step size of $h = 0.00025/n$ and 26 iterations. We found convergence for all values of n and l, when $n_{max} = 1000$, to require $h = 0.000000025$ and number of iterations 100. We are using a five point integration formula, where step size, h, is doubled after each iteration. In each iteration, κ^2 takes on the values

$\kappa^2, \kappa^2 + h, \kappa^2 + 2h, \kappa^2 + 3h, \kappa^2 + 4h$, where initially $\kappa^2 = 0$ for the first iteration. Hence all the values of κ^2 will be:

1^{st} iteration:

$$\kappa^2 = 0, h, 2h, 3h, 4h$$

2^{nd} iteration:

$$\kappa^2 = 4h, 6h, 8h, 10h, 12h$$

3^{rd} iteration:

$$\kappa^2 = 12h, 16h, 20h, 24h, 28h$$

$$\vdots \tag{3.36}$$

m^{th} iteration: (3.37)

$$\kappa^2 = 4h(2^{m-1} - 1), \ldots, 4h(2^m - 1)$$

As we have chosen the number of iterations, m, to be 100, and we are using a five point integration method, this leaves us with a total of $5 \cdot 100 = 500$ values needed of $\Theta(n, l, \kappa, l')$ for each triplet of n, l, l'. For each triplet we need two values of $\Theta(n, l, \kappa, l')$. In section 3.3 we found that for any n, l we need $n - l + 2$ or $n - l + 3$ calculations for each $\Theta(n, l, \kappa, l')$. We arbitrarily choose $n - l + 3$ as the calculations required for each $\Theta(n, l, \kappa, l')$ due to simplicity. As there are n values of l for each n, the total number of $\Theta(n, l, \kappa, l')$ values needed is:

$$500 \cdot 2 \sum_{n=1}^{n_{max}} n = 1000 \cdot \frac{n_{max}}{2}(n_{max} + 1) \tag{3.38}$$

To find the total number of evaluations of $G(n, l, \kappa, l')$ we must remember that there are $n - l + 3$ calculations for each pair n, l. Hence when calculating the number of calculations of $G(n, l, \kappa, l')$ for each level n, we must calculate the sum:

$$(n - 0 + 2) + (n - 1 + 2) + \ldots + [n - (n - 1) + 2] = (n + 2) + (n + 1) + \ldots + (3) \tag{3.39}$$

As such we can express the sum as:

$$\frac{n}{2}[(n + 2) + 3] = \frac{n}{2}(n + 5) \tag{3.40}$$

Hence the total number of calculations of $G(n, l, \kappa, l')$ for a given n_{max} is:

$$= 500 \cdot 2 \cdot \sum_{n=1}^{n_{max}} \frac{n}{2}(n+5) \tag{3.41}$$

$$= 500 \sum_{n=1}^{n_{max}} (n^2 + 5n) \tag{3.42}$$

$$= 500 \left[\sum_{n=1}^{n_{max}} n^2 + 5 \sum_{n=1}^{n_{max}} n \right] \tag{3.43}$$

$$= 500 \left[\frac{n_{max}^3}{3} + \frac{n_{max}^2}{2} + \frac{n_{max}}{6} + 5\frac{n_{max}(n_{max}+1)}{2} \right] \tag{3.44}$$

$$= 500 \left[\frac{n_{max}^3}{3} + \frac{n_{max}^2}{2} + \frac{n_{max}}{6} + 5\frac{n_{max}^2}{2} + 5\frac{n_{max}}{2} \right] \tag{3.45}$$

$$= 500 \left[\frac{n_{max}^3}{3} + 3n_{max}^2 + \frac{8n_{max}}{3} \right] \tag{3.46}$$

3.4 GPU Optimization

Similarly to the Einstein coefficients, many computations that are needed for the radiative recombination coefficients need only be done once, though not all of them. By using the integration technique from subsection 3.3.1 we need only calculate all the $\Theta(n, l, \kappa, l')$ values once. However, as the variable t in equations (3.9)-(3.11) is dependent on temperature T_e, we must re-calculate the integral whenever we wish to determine the radiative recombination coefficient for a different temperature. As such, this step is the most critical to optimize. We do this through implementing an OpenCL kernel that performs all parts of the calculation for α_{nl} except that of the $\Theta(n, l, \kappa, l')$ values. These are pre-calculated and loaded into the kernel as a `CL_READ_ONLY` buffer. The number of different, and independent, α_{nl} values that need to be calculated for $n_{max} = 1000$ is:

$$\frac{1000}{2}(1000+1) = 500,500 = 5.005 \cdot 10^5 \tag{3.47}$$

(see equation (3.38)). Hence the overhead of establishing an OpenCL kernel is heavily outweighed by the total time needed to calculate all of the values on the CPU.

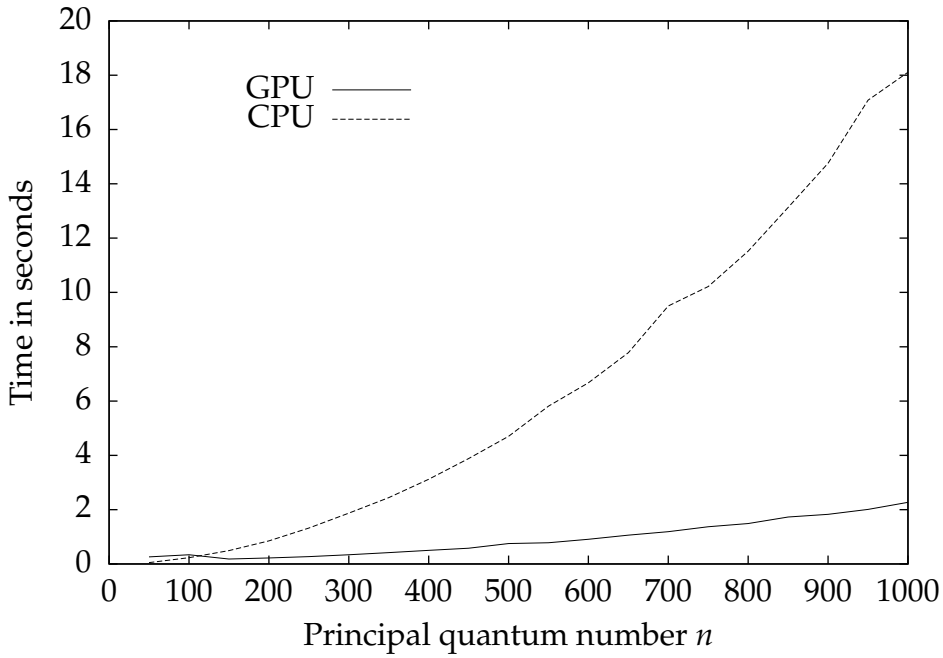

Figure 3.5: Plot of time taken to compute α_{nl} for n_{max} = 1000 for both GPU and CPU.

3.5 Conclusion

Burgess (1965) implemented the recursion relation for n up to 20. In this thesis we computed all n up to 1000. Furthermore, through extending the calculations to also use GPUs, this process could be done in a matter of seconds. Fig. 3.5 shows the difference in runtime for the calculation on a GPU (Nvidia Tesla C2070 - see Appendix B) vs. a CPU (Intel Xeon X5660 - see Appendix B) for values of n from 50-1000 with increments of 50. The graph for the GPU does not change smoothly for lower n. This happens because the most time consuming task at this point is to set up the OpenCL kernel. Hence there is no point in executing a kernel for low n, but a significant difference is seen at high(er) n. From the graph one can see approximately a 10 fold performance increase when n = 1000 when using GPUs instead CPUs and we can therefore conclude that there is a significant gain in implementing the computation of radiative recombinations on GPUs.

Chapter 4

Iterative Computation of b_n Coefficients

Sejnowski and Hjellming (1969) introduced the concept of solving the b_n problem iteratively. They write equation (1.17) as:

$$N_n P_n = N_c P_{cn} + \sum_{m=1}^{\infty} N_m P_{mn} \tag{4.1}$$

where

$$P_n = \sum_{m=1}^{\infty} P_{nm} \tag{4.2}$$

P_{cn} is the combined probability of capturing an electron to level n and $N_c = N_i$ is the ion density, where N_i is defined as in equation (1.11).

Using the expression for b_n as given in equation (1.16) and the formula for N_n^*, as given in equation (1.11), we have:

$$N_n = b_n N_n^* \tag{4.3}$$

where

$$N_n^* = N_c N_e \frac{h^3}{(2\pi m_e k T_e)^{3/2}} n^2 \exp(x_n) \tag{4.4}$$

and

$$x_n = E_n/(kT_e) \tag{4.5}$$

and E_n is the ionization energy of an atom in level n. We can now derive an implicit expression for b_n:

$$\begin{aligned} N_n/N_n^* &= b_n \\ &\Updownarrow \\ b_n &= \frac{N_c P_{nc} + \sum_{m=1}^{\infty} N_m P_{mn}}{P_n} \frac{1}{N_n^*} \\ &= \frac{N_c P_{cn}}{P_n} \frac{1}{N_n^*} + \frac{\sum_{m=1}^{\infty} N_m P_{mn}}{P_n} \frac{1}{N_n^*} \\ &= T_n + \frac{\sum_{m=1}^{\infty} \frac{N_m}{N_m^*} P_{mn}}{P_n} \frac{N_m^*}{N_n^*} \end{aligned}$$

Using equation (4.4), we get:

$$\frac{N_m^*}{N_n^*} = \frac{m^2}{n^2} \exp(x_m - x_n) \tag{4.6}$$

Noting that:

$$\frac{N_m}{N_m^*} = b_m \tag{4.7}$$

we get:

$$b_n = T_n + \sum_{m=1}^{\infty} b_m S_{mn} \tag{4.8}$$

where

$$T_n = \frac{(2\pi m k T_e)^{3/2}}{N_e h} \frac{P_{cn}}{n^2 P_n \exp(x_n)} \tag{4.9}$$

and

$$S_{mn} = \frac{P_{mn}}{P_n} \frac{m^2}{n^2} \exp(x_m - x_n) \tag{4.10}$$

Using equation (4.8) Sejnowski and Hjellming (1969) propose a scheme that iteratively solves equation (4.8) where initially $b_n = 1$ for all n.

Sejnowski and Hjellming (1969) give the following definitions of the population and depopulation probabilities:

$$P_{mn}{}^{R} = A_{mn}\left(1 + \frac{c^2 J_\nu}{2h\nu^3}\right) \tag{4.11}$$

$$P_{nm}{}^{R} = A_{mn}\frac{\omega_m}{\omega_n}\frac{c^2 J_\nu}{2h\nu^3} \tag{4.12}$$

$$P_{nc}{}^{R} = \int_0^\infty \frac{4\pi J_\nu}{h\nu}\alpha_n(\nu)d\nu \tag{4.13}$$

$$P_{cn}{}^{R} = N_e \int_0^\infty \sigma(v)\left(1 + \frac{c^2 J_\nu}{2h\nu^3}\right) v f(v) dv \tag{4.14}$$

where ω_n is the statistical weight. J_ν is the average intensity of the radiation field at frequency ν; $\alpha_n(\nu)$ is the cross-section for photo-ionization from level n for a photon at frequency ν; v is the speed of a free electron; $f(v)$ is the Maxwellian velocity distribution for free electrons and $\sigma(v)$ is the cross-section for electrons at speed v to recombine to level n.

In (4.11)-(4.14), index R refers to the radiative probability component. As such, we also have the following collisional probability components:

$$P_{nc}{}^{C} = 7.8 \times 10^{-11} T_e^{1/2} n^3 \exp(-x_n) N_e \tag{4.15}$$

$$P_{nm}{}^{C} = 1.2 \times 10^{-7} f_{nm} \exp(-x_{nm})\left(\frac{E_m - E_n}{E_1}\right)^{-1.1856} N_e \tag{4.16}$$

where f_{nm} is the oscillator strength from level n to m and $x_{nm} = (E_m - E_n)/(kT_e) = x_m - x_n$. These probabilities are for what Sejnowski and Hjellming (1969) refer to as Class II cross sections. We will not deal with Class I cross sections here.

Sejnowski and Hjellming (1969) use the principle of detailed balancing to calculate ${P_{mn}}^C$:

$$n^2 \exp(x_n) {P_{nm}}^C = m^2 \exp(x_m) {P_{mn}}^C \tag{4.17}$$

$$\Updownarrow$$

$$\frac{n^2 \exp(x_n)}{m^2 \exp(x_m)} {P_{nm}}^C = {P_{mn}}^C \tag{4.18}$$

$$\Updownarrow$$

$${P_{mn}}^C = \frac{n^2}{m^2} \exp(x_n - x_m) {P_{nm}}^C \tag{4.19}$$

and ${P_{cn}}^C$:

$${P_{cn}}^C = \frac{N_e h^3}{(2\pi m k T_e)^{3/2}} n^2 \exp(x_n) {P_{nc}}^C \tag{4.20}$$

The total probabilities are then given by:

$$\begin{aligned} P_{mn} &= {P_{mn}}^R + {P_{mn}}^C \\ P_{nm} &= {P_{nm}}^R + {P_{nm}}^C \\ P_{cn} &= {P_{cn}}^R + {P_{cn}}^C \\ P_{nc} &= {P_{nc}}^R + {P_{nc}}^C \end{aligned}$$

In our calculations of b_n for the iterative method, we have made the simplifying assumption that $J_\nu = 0$ for all frequencies ν. As such, stimulated emission or stimulated absorption is not taken into account. Hence (4.11)-(4.14) become:

$${P_{mn}}^R = A_{mn} \tag{4.21}$$

$${P_{nm}}^R = 0 \tag{4.22}$$

$${P_{nc}}^R = 0 \tag{4.23}$$

$${P_{cn}}^R = N_e \int_0^\infty \sigma(v) v f(v) dv \tag{4.24}$$

and the collisional processes remain the same as they are not affected by external radiation. Using Seaton (1959b, p.92) we have the expression for A_{mn}:

$$A_{mn} = \left(\frac{8\alpha^4 c}{3\pi a_0 \sqrt{3}}\right)\frac{Z^4}{m^5} \times \frac{2m^2 g_{mn}}{n(m^2 - n^2)} \tag{4.25}$$

where g_{mn} is the bound-bound Gaunt factor given by equation (2.22) and α is the fine structure constant. As noted in Sejnowski and Hjellming (1969), one can use the Milne relation to relate $\sigma_n(v)$ to $\alpha_n(\nu)$. Hence we express ${P_{cn}}^R$ in terms of $\alpha_n(\nu)$ which we obtain from Seaton (1959a, p.81):

$$\alpha_n(T_e) = D\frac{\lambda^{1/2}}{n}x_n S_n(\lambda) \tag{4.26}$$

and

$$\begin{aligned} D &= \frac{2^6}{3}\left(\frac{\pi}{3}\right)^{1/2}\alpha^4 c a_0^2 && (4.27)\\ &= 5.197 \times 10^{-14}\mathrm{cm}^3\mathrm{sec}^{-1} && (4.28)\\ \lambda &= \frac{hRc}{kT_e} && (4.29)\\ &= 15.789 \times 10^4 \frac{1}{T_e} && (4.30)\\ x_n &= \frac{\lambda}{n^2} && (4.31)\\ S_n(\lambda) &= \int_0^\infty \frac{g_{II}(n,\epsilon)e^{x_n u}}{(1+u)}du && (4.32) \end{aligned}$$

where $g_{II}(n,\epsilon)$ is the free-bound Gaunt factor and $u = n^2\epsilon$. Furthermore, we have:

$$g_{II}(n,\epsilon) = 1 + 0.1728n^{-2/3}(u+1)^{-2/3}(u-1) - 0.0496n^{-4/3}(u+1)^{-4/3}(u^2 + \frac{4}{3}u + 1) + \ldots \tag{4.33}$$

and substituting (4.33) in (4.32) we have:

$$S_n(\lambda) = S^{(0)}(x_n) + \lambda^{-1/3}S^{(1)} + \lambda^{-2/3}S^{(2)} + \ldots \tag{4.34}$$

For simplicity, we will approximate $S_n(\lambda)$ by using only the first term, $S^{(0)}(x_n)$, defined by:

$$S^{(0)}(x) = e^x \int_x^\infty \frac{e^{-v}}{v}dv \tag{4.35}$$

This approximation will have an effect, but not dominate, for levels $n = 1, 2, 3$.

As we are interested in comparing with Sejnowski and Hjellming (1969), whose solutions are only given for $n > 20$, this should not cause any noticable difference. Hence our final expression for α_n is:

$$\alpha_n = 5.197 \times 10^{-14} x_n^{3/2} S_0(x_n) \tag{4.36}$$

Using the Milne relation, as given by Osterbrock and Ferland (2006, p.401) we have:

$$\alpha_n = \int_0^\infty \sigma(v) v f(v) dv \tag{4.37}$$

which is the same as in equation (4.24). We can thus do a direct substitution and finally obtain:

$$P_{cn}{}^R = N_e \alpha_n \tag{4.38}$$

4.1 Implementation

The b_n calculations were done in standard C. Furthermore, the *GNU Scientific Library* (Gough, 2009) was used in order to perform integration of the exponential integral in equation (4.35). When compiling, one must include "`constants.h`", "`oscillator_strength_gaunt_final.c`" and "`expint.c`" which define the physical constants needed (in ESU units); the bound-bound Gaunt factors and the setup needed to calculate the exponential integral. From equations (4.8)-(4.10) it becomes apparent that most calculations need only be performed once. A scheme was developed where, for each n; P_n, P_{cn} and T_n were calculated. As these are only dependent on n, 1-dimensional arrays sufficed. Furthermore, for each value of n, an array of values was needed for S_{mn} as m takes on all possible values of n. Hence a 2-dimensional array to hold the values of S_{mn} was needed. After calculations of these arrays, each b_n was calculated according to equation (4.8). To ensure consistent results, all b_n's were calculated for each iteration before continuing.

To avoid branching when calculating the S_{mn} terms we defined two loops in the function `calc_S_mn_term`:

```
1    int i;
2    for(i=CASE;i<n;i++)
3    {
4        S_mn[n-CASE][i-CASE] = s_mn_term_n_greater(P_n, n, i, beta, T_e,
             N_e);
5    }
6    for(i=n+1;i<=num_of_b_n;i++)
7    {
8        S_mn[n-CASE][i-CASE] = s_mn_term_m_greater(P_n, n, i, beta, T_e,
             N_e);
9    }
```

Figure 4.1: `calc_S_mn_term` algorithm

We then let `s_mn_term_m_greater` and `s_mn_term_n_greater` in turn call `p_mn_term_m_greater` and `p_mn_term_n_greater`. This is needed as when m is greater we must include spontaneous radiation (as m is the upper level and can thus spontaneously transition downwards) and apply detailed balancing to our collisional term. By writing out two different functions we avoid making any branching which will decrease performance for a GPU implementation.

4.1.1 Output

When calculating the b_n coefficients we are very much interested in knowing not only the b_n coefficients themselves but also the logarithmic derivative and the β value as defined in equation (1.15). However, as Gordon and Sorochenko (2002) use the following form for β, we redefine it as:

$$\beta \equiv \left(1 - \frac{kT_e}{h\nu}\frac{d\ln(b_{n_2})}{dn}\Delta n\right) \tag{4.39}$$

Secondly, as we are establishing discrete values of b_n we define the logarithmic derivative as:

$$\frac{d\ln(b_n)}{dn} \equiv \frac{b_{n+1} - b_n}{b_n} \tag{4.40}$$

The function `write_output` thus takes the final values of b_n and writes to a text file in the format of:

n	b_n	$\frac{d\ln(b_n)}{dn}$	β
1	$\cdots$	$\cdots$	$\cdots$
2	$\cdots$	$\cdots$	$\cdots$
$\vdots$	$\cdots$	$\cdots$	$\cdots$

Below is given an example of the output of the first 9 values from a calculation ('#' is needed to ignore the first line when reading the file using *GNUPlot*):

```
# n  b_n              db/b            beta
2    26218526584.933... nan           1.0000000000
3    2.3226056803     nan             1.0605858215
4    0.6572058324     nan             1.0963419014
5    0.5184360528     nan             1.0278584810
6    0.5073570940     -1.5870106201   0.9271052771
7    0.5204882547     -1.4607079426   0.8205137810
8    0.5385061057     -1.4759779291   0.7126133918
9    0.5565036273     -1.5238207438   0.6033994807
10   0.5731625631     -1.5803451843   0.4921211652
```

Figure 4.2: Example output for b_n, $\frac{d\ln(b_n)}{dn}$ and β

4.2 GPU Optimization

The benefit of an iterative scheme is that it lends itself well to be parallelized. If one can manage to make each calculation independent of the others, GPUs can greatly improve performance.

The nature of the iterative approach is to solve each b_n independently for any given iteration. As we are calculating in the vicinity of $n \sim 1000$ this has the potential to use 1000 GPU compute units independently for each iteration. However, in order for this to be feasible, there are two criteria that must be met. We must ensure:

1. That there are two buffers to hold the value of the b_n's
2. That the calculation of b_n for each n is complete before the next iteration begins i.e. we must ensure that the calculation of b_n in iteration i uses the value of b_n from iteration $i-1$.

Criterion 1 ensures that when calculating a given b_n value in iteration i, following equation (4.8), it can write to an output buffer that will only be read in iteration $i+1$. Thus when calculating other b_n values in iteration i, they will not see this newly updated value, but only use values calculated in iteration $i-1$.

Criterion 2 ensures that *all* values calculated in iteration i will be updated, stored in the output buffer and be available for reading in iteration $i+1$.

These two criteria have the potential of decreasing performance gains when using GPUs. However, criterion 1 is not an issue since in iteration i, buffer one can act as a read-only buffer whereas buffer two can act as a write-only buffer. In iteration $i+1$ they switch roles and become a write-only and read-only buffer respectively. As such there will be no need for copying any data and hence no overhead.

Criterion 2 becomes an issue for performance. The nature of GPUs means that one should try to avoid synchronization in order to gain optimal performance. However, in order to ensure Criterion 2 we must synchronize after each iteration. This is typically done using CPU synchronization. However, this is not the most effective method as it still involves communication back and forth

with the CPU. Hence, Stuart and Owens (2011) propose various techniques for performing synchronization on the GPU rather than the CPU with significant speedups. The XF barrier (Xiao & Feng, 2010), which is an appropriate synchronization mechanism for our approach, appears to be the most effective (Stuart & Owens, 2011). Implementation on the GPU for the iterative method is outside the scope of this thesis. However, due to the nature of the iterative scheme there is no doubt that this can be done, which would be expected to greatly increase performance.

4.3 Results

There were several complications with reproducing the results of Sejnowski and Hjellming (1969). They specify that:

> "...the b_n solutions we will discuss will depend only on N_e, T_e, and assumptions concerning cross-sections and the techniques of calculation".

However, they do not specify how J_ν should be calculated. Hence, as stated earlier, we assume J_ν to be equal to zero - the assumption that was normally made in early b_n calculations (Gordon & Sorochenko, 2002). Furthermore, the alternative recursion scheme suggested (Sejnowski & Hjellming, 1969, p.919) did not work and as such we reverted to using the standard approach. However, as Sejnowski and Hjellming (1969) note, this will take many iterations in order to converge. Figs. 4.3 and 4.4 show this clearly as they do not convergence until the number of iterations becomes ~ 5000. This is far more iterations than the 150 iterations needed for convergence by the recursion method of Sejnowski and Hjellming (1969, p.921).

In Figs. 4.5 and 4.6 we see the importance of choosing n_{max} large enough. Sejnowski and Hjellming (1969) give examples where they have truncated at $n = 240$ after which an unspecified analytic continuation is performed. From Fig. 4.5 it is evident that divergence between $n_{max} = 250$ and $n_{max} > 250$ becomes apparent at approximately $n = 225$ i.e. to obtain a correct value of b_n or the logarithmic derivative, for $n = 250$ we must have $n_{max} - n > 25$. Sejnowski and Hjellming (1969) state that $n_{max} - n > 10$ should suffice. As they truncate at

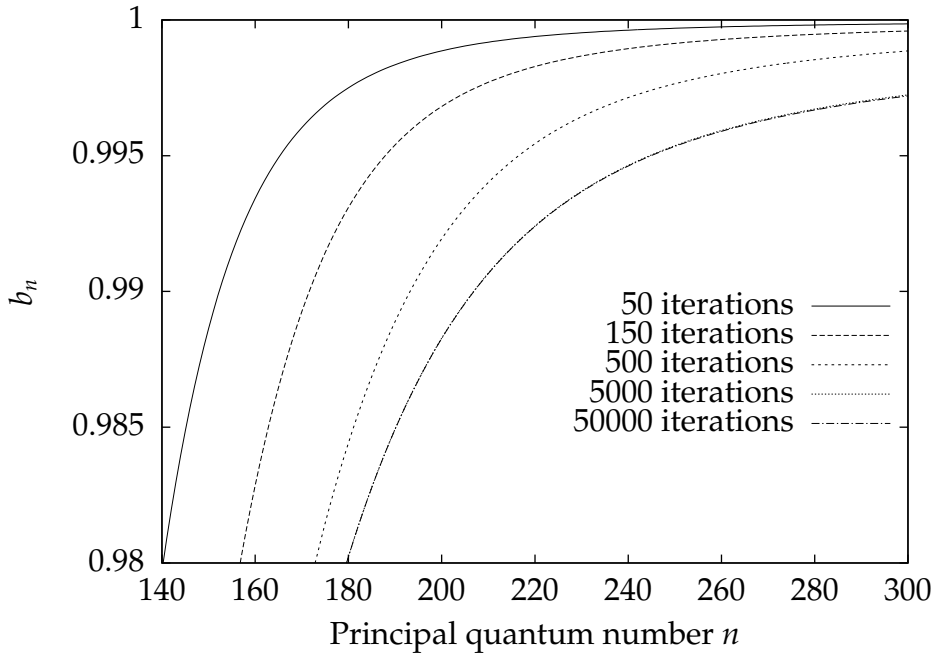

Figure 4.3: Plot of b_n at various iterations for fixed density $N_e = 10$ and temperature $T_e = 10^4$.

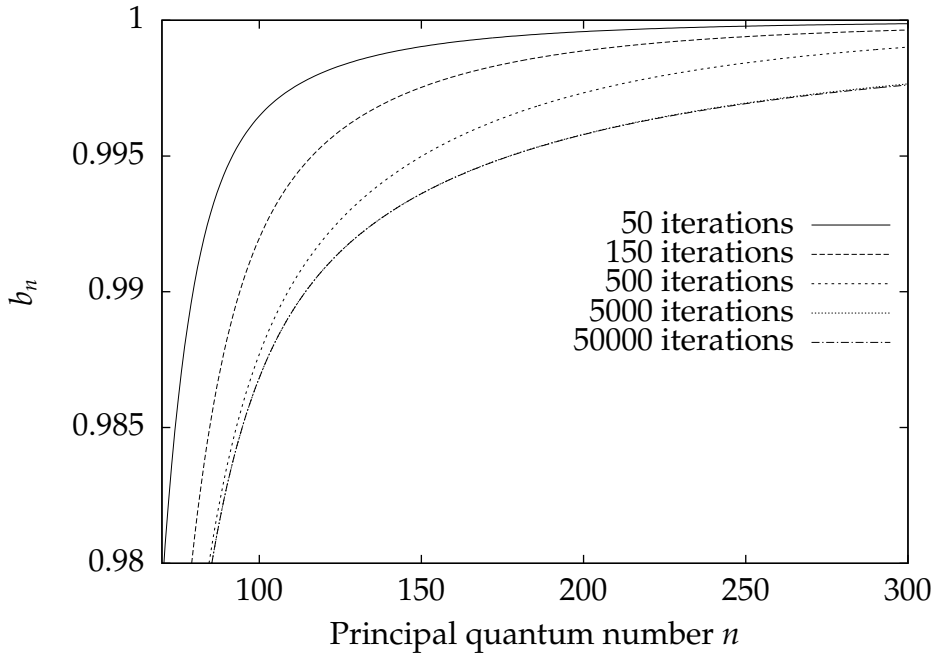

Figure 4.4: Plot of b_n at various iterations for fixed density $N_e = 10^4 \mathrm{cm}^{-3}$ and temperature $T_e = 10^4$.

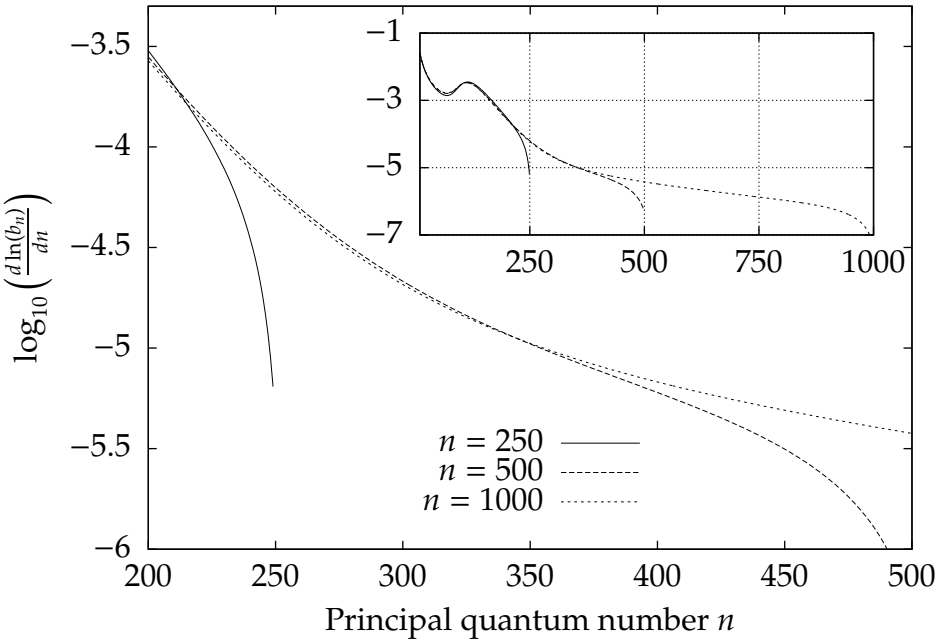

Figure 4.5: Plot of $\log_{10}\left(\frac{d\ln(b_n)}{dn}\right)$ for fixed density $N_e = 10\text{cm}^{-3}$, temperature $T_e = 10^4$ and a fixed number of iterations of 5000.

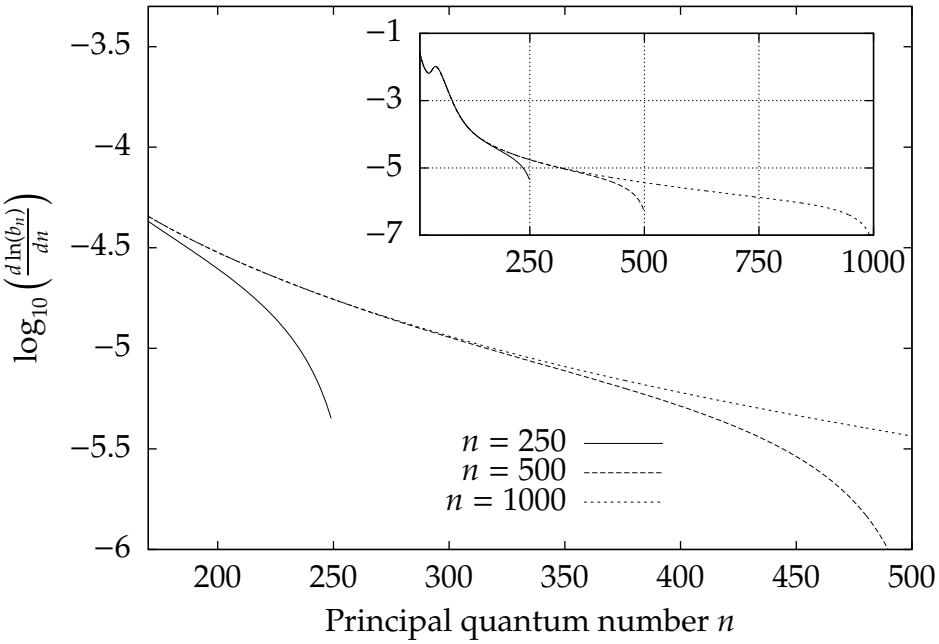

Figure 4.6: Plot of $\log_{10}\left(\frac{d\ln(b_n)}{dn}\right)$ for fixed density $N_e = 10^4\text{cm}^{-3}$, temperature $T_e = 10^4$ and a fixed number of iterations of 5000.

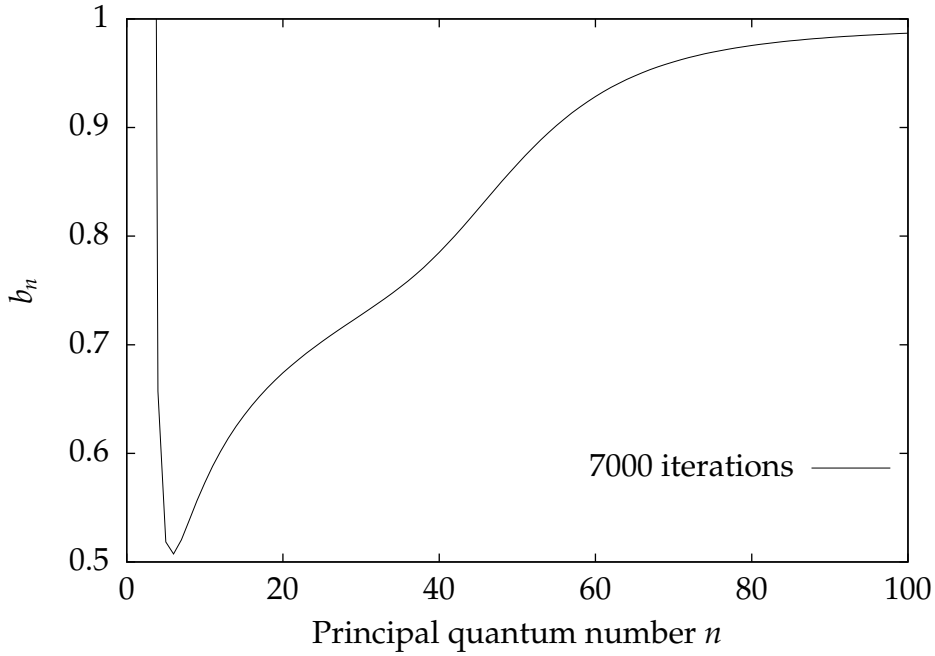

Figure 4.7: Plot of b_n showing spurious results for $n \leq 6$. Number of iterations is 7000 and $N_e = 10^4$ and $T_e = 10^4$.

$n = 240$ this seems a fair assumption. However, when calculating for higher density, as in Fig. 4.6, this is not the case as divergence starts already at $n_{max} - 50$.

Secondly, when calculating for larger n, both figures clearly depict a much larger discrepancy between $n_{max} = 500$ and $n_{max} = 1000$. For Fig. 4.6 this discrepancy starts at 350 i.e. at $n_{max} - n = 150$.

Lastly, it should be pointed out, as shown in Fig. 4.7 that spurious results are obtained for values $n \leq 6$. As Sejnowski and Hjellming (1969) have no graphs where b_n's with $n < 20$ are shown, we cannot say if this is a fault in the program or a result from the recursion scheme. However, this result should not be expected, even when neglecting J_ν.

4.3.1 Performance

In Fig. 4.8 we present the time taken to calculate b_n for $n_{max} = 250, n_{max} = 500, n_{max} = 1000$ and $n_{max} = 2000$. The number of iterations used for the calculations are between 50 and 5000, with increments of 50. Although there are peaks on both graphs, the relationship between the runtime and the number

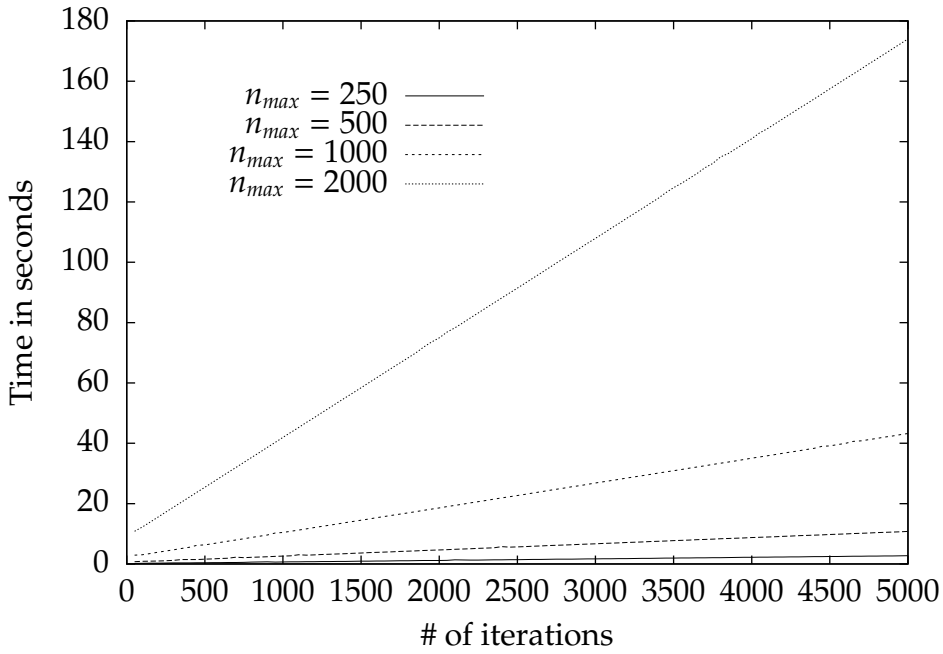

Figure 4.8: Plot of time taken to compute b_n for $n_{max} = 250, n_{max} = 500$ and $n_{max} = 1000$ vs. number of iterations.

of iterations is clearly linear in both cases. When calculating the gradients, $m_{n_{max}}$, for $n_{max} = 250, n_{max} = 500, n_{max} = 1000$ and $n_{max} = 2000$, we choose two points for each n_{max} that are neither on a peak, nor where the number of iterations is low. The latter must be done to avoid counting time that is spent initialising variables etc. as this has more effect on lower values for number of iterations but becomes negligible when the number of iterations is large.

We now select two random points for each n_{max} as described above and thus obtain the following gradients:

$$m_{250} = \left(\frac{2.67 - 0.91}{4900 - 1500}\right) = \frac{1.76}{3400} = 0.00052 \tag{4.41}$$

$$m_{500} = \left(\frac{10.07 - 3.63}{4650 - 1500}\right) = \frac{6.44}{3150} = 0.00204 \tag{4.42}$$

$$m_{1000} = \left(\frac{42.84 - 14.51}{4950 - 1500}\right) = \frac{28.33}{3450} = 0.00821 \tag{4.43}$$

$$m_{2000} = \left(\frac{167.39 - 58.41}{4800 - 1500}\right) = \frac{108.98}{3300} = 0.03302 \tag{4.44}$$

We thus have:

$$\frac{m_{500}}{m_{250}} = \frac{0.00204}{0.00052} = 3.9 \tag{4.45}$$

$$\frac{m_{1000}}{m_{500}} = \frac{0.00821}{0.00204} = 4.0 \tag{4.46}$$

$$\frac{m_{2000}}{m_{1000}} = \frac{0.03302}{0.00821} = 4.0 \tag{4.47}$$

Equations (4.45)-(4.47) clearly suggest that when there is an increase from n to $2^x n$, the time, t, taken to compute b_n increases as $4^x t$.

Chapter 5

Matrix Computation of b_n Coefficients

In this chapter we solve the b_n problem in a somewhat straightforward way. As for the iterative approach in Chapter 4, we define a finite n_{max}, instead of infinity, as the upper limit for the infinite sums in equation (1.26). Rewriting equation (1.26) we then get a set of linear equations of the form:

$$N_2X_2 - N_3Y_{23} - \ldots - N_mY_{2m} - \ldots - N_{n_{max}}Y_{2n_{max}} = Z_2 \tag{5.1}$$

$$-N_2Y_{32} + N_3X_3 - \ldots - N_mY_{3m} - \ldots - N_{n_{max}}Y_{3n_{max}} = Z_3 \tag{5.2}$$

$$\vdots \tag{5.3}$$

$$-N_2Y_{m2} - N_3Y_{m3} - \ldots + N_mX_m - \ldots - N_{n_{max}}Y_{mn_{max}} = Z_m \tag{5.4}$$

$$\vdots \tag{5.5}$$

$$-N_2Y_{n_{max}2} - N_3Y_{n_{max}3} - \ldots - N_mY_{n_{max}m} - \ldots + N_{n_{max}}X_{n_{max}} = Z_{n_{max}} \tag{5.6}$$

where:

$$X_n = \sum_{m=n_0, m\neq n}^{n_{max}} (C_{nm} + B_{nm}\rho_\nu) + \sum_{m=n_0, m\neq n}^{n-1} A_{nm} + C_{ni} + B_{ni}\rho_\nu \tag{5.7}$$

$$Y_{nm} = C_{mn} + B_{mn}\rho_\nu + A_{mn} \tag{5.8}$$

$$Z_n = N_eN_i(\alpha_n + C_{in}) + B_{in}\rho_\nu \tag{5.9}$$

and $A_{mn} = 0$ if $m < n$.

Putting this into matrix notation we have:

$$\begin{pmatrix} X_2 & -Y_{23} & \dots & -Y_{2m} & \dots & -Y_{2n_{max}} \\ -Y_{32} & X_3 & \dots & -Y_{3m} & \dots & -Y_{3n_{max}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -Y_{m2} & -Y_{m3} & \dots & +X_m & \dots & Y_{mn_{max}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -Y_{n_{max}2} & -Y_{n_{max}3} & \dots & -Y_{n_{max}m} & \dots & X_{n_{max}} \end{pmatrix} \times \begin{pmatrix} N_2 \\ N_3 \\ \vdots \\ N_m \\ \vdots \\ N_{n_{max}} \end{pmatrix} = \begin{pmatrix} Z_1 \\ Z_2 \\ \vdots \\ Z_m \\ \vdots \\ Z_{n_{max}} \end{pmatrix}$$

From this it is clear that the first matrix can be written as the sum of the two matrices:

$$Y_M = \begin{pmatrix} 0 & -Y_{23} & \dots & -Y_{2m} & \dots & -Y_{2n_{max}} \\ -Y_{32} & 0 & \dots & -Y_{3m} & \dots & -Y_{3n_{max}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -Y_{m2} & -Y_{m3} & \dots & 0 & \dots & Y_{mn_{max}} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ -Y_{n_{max}2} & -Y_{n_{max}3} & \dots & -Y_{n_{max}m} & \dots & 0 \end{pmatrix}$$

and

$$X_M = \begin{pmatrix} X_2 & 0 & \dots & 0 & \dots & 0 \\ 0 & X_3 & \dots & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & X_m & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ 0 & 0 & \dots & 0 & \dots & X_{n_{max}} \end{pmatrix}$$

Furthermore, if we remove the terms C_{ni} and $B_{ni}\rho_\nu$ from the X_n term, it becomes evident from equations (5.7) and (5.8) that each X_n term is in fact the sum of all the Y_{nm} terms in row n. Secondly, due to detailed balancing, as used in Chapter 4, we have symmetry between each entry Y_{nm} and Y_{mn} and thus it suffices to compute just one of these terms and apply detailed balancing to obtain the other.

Below, we define the terms used in equations (5.7)-(5.9).

For bound-bound collisional transitions we have:

$$C_{nm} = n^4(J_1 + J_2 + J_3)/T_e^{3/2} \tag{5.10}$$

where

$$s = m - n > 0 \tag{5.11}$$

$$\beta = 10^5 \times 1.58/T \tag{5.12}$$

$$\beta_1 = 1.4(nm)^{1/2} \tag{5.13}$$

$$E = 0.85/\beta \tag{5.14}$$

$$A = \frac{8}{3s}\left(\frac{m}{sn}\right)^3 (0.184 - 0.04/s^{2/3})\left(1 - \frac{0.2s}{nm}\right)^{1+2s} \tag{5.15}$$

$$L = \ln\left(\frac{1 + 0.53E^2nm}{1 + 0.4E}\right) \tag{5.16}$$

$$J_1 = \frac{4}{3}AL\left(\frac{0.85}{\beta}\right)\left(\frac{1}{\beta} - \frac{1}{\beta + \beta_1}\right) \tag{5.17}$$

$$J_2 = \frac{16}{9}\frac{F_1m^3}{y_1}\frac{(\sqrt{2 - n^2/m^2} + 1)^3}{(n + m)^3s^3}\frac{\exp(-\beta/\beta_1)}{\beta} \tag{5.18}$$

$$F_1 = \left(1 - \frac{0.3s}{nm}\right)^{1+2s} \tag{5.19}$$

$$y_1 = \left[1 - \frac{\ln(18s)}{4}s\right]^{-1} \tag{5.20}$$

$$J_3 = \frac{1}{4}\left(\frac{n^2\xi_-}{m}\right)^3 \frac{J_4(z)}{\beta + \beta_1}\ln(1 + 0.5\beta\xi_-) \tag{5.21}$$

$$\xi_- = 2/\left[n^2\left(\sqrt{2 - n^2/m^2} - 1\right)\right] \tag{5.22}$$

$$z = 0.75\xi_-(\beta + \beta_1) \tag{5.23}$$

$$J_4(z) = \frac{2}{z}\frac{1}{2 + z(1 + e^{-z})} \tag{5.24}$$

and

$$C_{mn} = C_{nm} \times \left(\frac{m}{n}\right)^2 \exp\left(-\beta\left[\frac{1}{n^2} - \frac{1}{m^2}\right]\right) \tag{5.25}$$

(Gee, Percival, Lodge, & Richards, 1976).

For spontaneous radiation we have:

$$A_{nm} = 1.574 \cdot 10^{10}\frac{n^{-5}m^{-3}}{m^{-2} - n^{-2}}g^I_{nm} \tag{5.26}$$

(Shaver, 1975, p.5) where g^I_{nm} is the Gaunt factor.

For stimulated radiation we have:

$$B_{nm}\rho_\nu = WA_{nm}/\left[\exp(-h\nu/[kT_r]) + 1\right] \tag{5.27}$$

and

$$B_{mn}\rho_\nu = B_{nm}\rho_\nu \left(\frac{n}{m}\right)^2 \tag{5.28}$$

where T_r is the radiation temperature in the observed nebula and W is the dilution factor for the source of radiation.

For collisional ionization:

$$C_{ni} = N_e \cdot 3.45 \cdot 10^{-5} \frac{n^2}{\sqrt{T_e}} \exp(-x_n) \tag{5.29}$$

where

$$x_n = I_n/(kT_e) \tag{5.30}$$

and

$$I_n = 2.179 \cdot 10^{-11}/n^2 \tag{5.31}$$

Hence for three-body recombination:

$$N_e N_i C_{in} = N_e N_i \frac{N_n^*}{N_e N_i} C_{ni} = N_n^* C_{ni} \tag{5.32}$$

(Dupree, 1969, p.494) and (Shaver, 1975, p.8).

For stimulated radiative ionization we have:

$$B_{ni}\rho_\nu = z_0 \frac{W}{n^5} \int_{I_n/(kT_r)}^{\infty} \frac{1}{x(e^x - 1)} dx \tag{5.33}$$

where

$$z_0 = \frac{8\alpha^4 c}{3\sqrt{3}\pi a_0} \tag{5.34}$$

For stimulated recombination we have:

$$B_{in}\rho_\nu = \frac{z_1}{T_e^{3/2}} z_0 \frac{W}{n^3} \frac{\exp(-T_r x/T_e)}{x(\exp(x) - 1)} dx \tag{5.35}$$

where

$$z_1 = 8(\pi a_0^2 I_n/k)^{3/2} \tag{5.36}$$

Finally, for radiative recombination we have:

$$N_e N_i \alpha_n = N_e N_i \cdot 5.197 \times 10^{-14} x_n^{3/2} S_0(x_n) \tag{5.37}$$

[see equation (4.36)].

5.1 Implementation

The implementation of this program relies on an unpublished Matlab program by Professor Sergei Gulyaev, that solves the b_n problem. The implementation given in this program was written in standard C. As in Chapter 4, when compiling, we must include "`constants.h`", "`oscillator_strength_gaunt_final.c`" and "`expint.c`".

When implementing our program, we first populate the matrix Y_M from which we can create a second matrix, X_M, obtained from Y_M. Finally, we add these two matrices. We then use LU factorization (Serre, 2010, p.208) to solve the resulting system of linear equations. For this purpose, we make use of the Meschach library which supports various matrix operations as well as LU factorization. Stewart and Leyk (1994) describe Meschach and outline its advantages over the very well-known linear algebra libraries LINPACK, EISPACK and LAPACK.

When populating the matrix Y_M we make extensive use of detailed balancing for collisional transition (shown in equation (5.25)) and stimulated radiation (shown in equation (5.28)). Hence we need only calculate values below the diagonal for the matrix Y_M and apply detailed balancing to obtain the values above the diagonal. The function that performs this calculation is shown in Fig. 5.1.

```
// 'beta' = 1.57e5/T_e
// 'W' is the dilution factor of the radiation field
// 'T_r' is the temperature of the radiaion field
// 'nu' is the frequency of radiation
for(n=N0;n<=N;n++)
{
    for(m=N0;m<n;m++)
    {
        // m<n
        spon_rad         =
            spontaneous_radiation(n,m)*gaunt_approximation(m,n);
        col_trans        = collisional_transition(m,n,beta,T_e,N_e);
        col_trans_db     =
            col_trans*pow((m+0.0)/n,2)*exp(-beta*(1/pow(n,2)-1/pow(m,2)));
        ind_rad          = W*spon_rad/(exp(h*nu(n,m)/k/T_r)+1);
        ind_rad_db       = ind_rad*pow((n+0.0)/m,2);
        C[n-N0][m-N0]    = -(spon_rad+col_trans_db+ind_rad);
        C[m-N0][n-N0]    = -(col_trans+ind_rad_db);
        C[n-N0][n-N0]    = 0.0;
        C[m-N0][m-N0]    = 0.0;
    }
}
```

Figure 5.1: `Y_m_population.c`

5.2 GPU Optimization

In this chapter we solve the b_n problem by adding matrices and using LU factorization. There are three major steps in solving the system of linear equations in equations (5.1) - (5.6):

1. We must populate the matrices Y_M and X_M and add them together.
2. We must populate the matrix $Z_M = (Z_1 \quad Z_2 \quad \cdots \quad Z_{n_{max}})^\intercal$.
3. We must apply LU factorization to the resulting system of linear equations in order to solve it.

Step 1 will be a bottleneck as it involves populating an entire matrix of dimension $n_{max} \times n_{max}$. As we want to push the limits for what levels can be computed, we let n go from $10^3 - 10^4$. This will result in matrices with number of entries between $10^6 - 10^8$. From the nature of matrices X_M and Y_M it is clear that the population of an entry i, j can be done independently of all other entries. Also, the process is the same for each entry as shown in Fig. 5.1. Hence this problem is very well suited for GPU parallelization, as it obeys both property 1 and property 2 for GPUs (see section 1.4). Furthermore, the kernel space will have size $\sim 10^6 - 10^8$ which will heavily outweigh the cost of setting up a kernel.

Step 2 will most likely not be a bottleneck as the number of terms will only be between $10^3 - 10^4$ and hence the cost of setting up a kernel may outweigh gain in computation speed.

Step 3 will result in the greatest speed up if utilising a GPU as solving the system of linear equations will be the most intensive task due to the nature of LU factorization.

5.3 Results

The scope of this thesis only allowed for step 1 to be completed. However, it is clear from Fig. 5.2 that the GPU approach has a significant impact on speed. For $n = 1000$ there is only a two fold speed increase but at $n = 10000$ this increases to approximately 30 times.

Although step 3 was not carried out, Mukunoki and Takahashi (2012) have shown a 30 fold increase in performance when performing quadruple precision

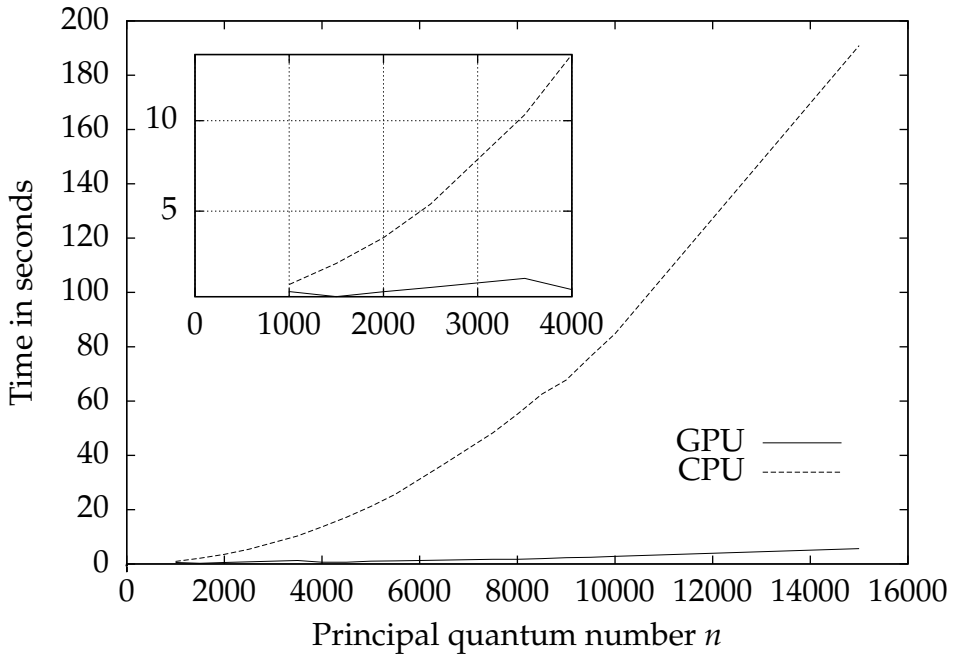

Figure 5.2: Plot of time taken to compute Y_M for $n_{max} = 1000 - 10000$.

BLAS calculations on a Tesla C1060 GPU as opposed to using an Intel Core i7 920. Secondly, a test performed on an AMD Opteron with an Nvidia GeForce GTX 280 has shown a similar performance increase when performing LU factorization (Lezar & Davidson, 2010). Hence we would expect a significant performance increase when carrying out step 3 on the GPU.

Finally, we produce a graph of b_n versus n (Fig. 5.3) and of the logarithmic derivative (5.4) to show the calculation of b_n for $n_{max} = 1000$.

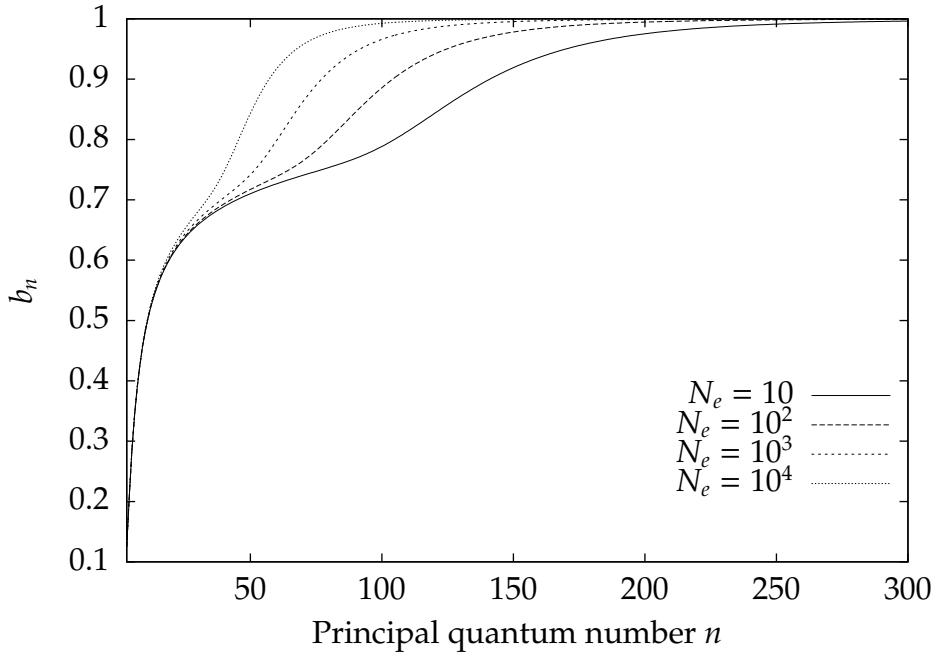

Figure 5.3: Plot of b_n for four different densities and temperature $T_e = 10^4$.

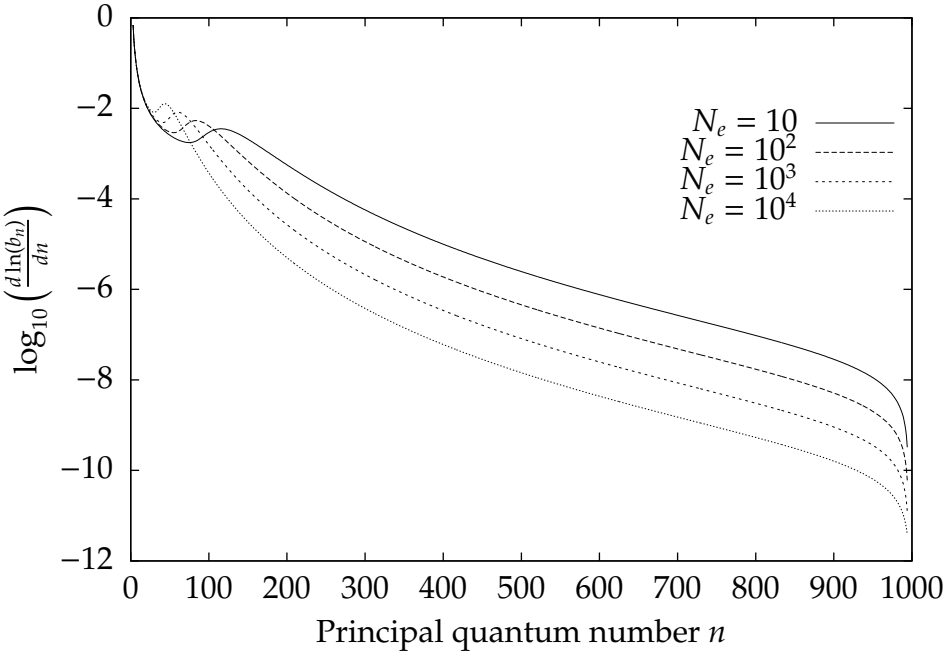

Figure 5.4: Plot of $\log_{10}\left(\frac{d\ln(b_n)}{dn}\right)$ for four different densities and temperature $T_e = 10^4$.

5.4 Comparison of Matrix Computation and Iterative Computation of b_n Coefficients

In Fig. 5.5 we have plotted a calculation of b_n for $n_{max} = 2000$ in the case of the iterative approach and $n_{max} = 1000$ for the matrix approach to avoid any spurious results. Secondly, we have shown values of b_n between $n = 20$ and $n = 250$. The lower limit is due to the spurious results for the iterative method as shown in Chapter 4.

There is clearly some difference in the results between the two methods. However, this is partly due to differences in the definitions of the processes. It was discovered that when only considering radiative processes, the iterative method and the matrix method agreed fairly well. However, the way in which collisional transitions behave differs between the two methods which is evident from Fig. 5.5. Overall, the matrix method appears to be more stable in the way at which it provides results that are not dependent on the number of iterations.

When comparing speed, it will be most likely that the matrix method will benefit mostly from the use of GPUs as there will be no need for synchronization and hence one can avoid costly call-backs to the CPU.

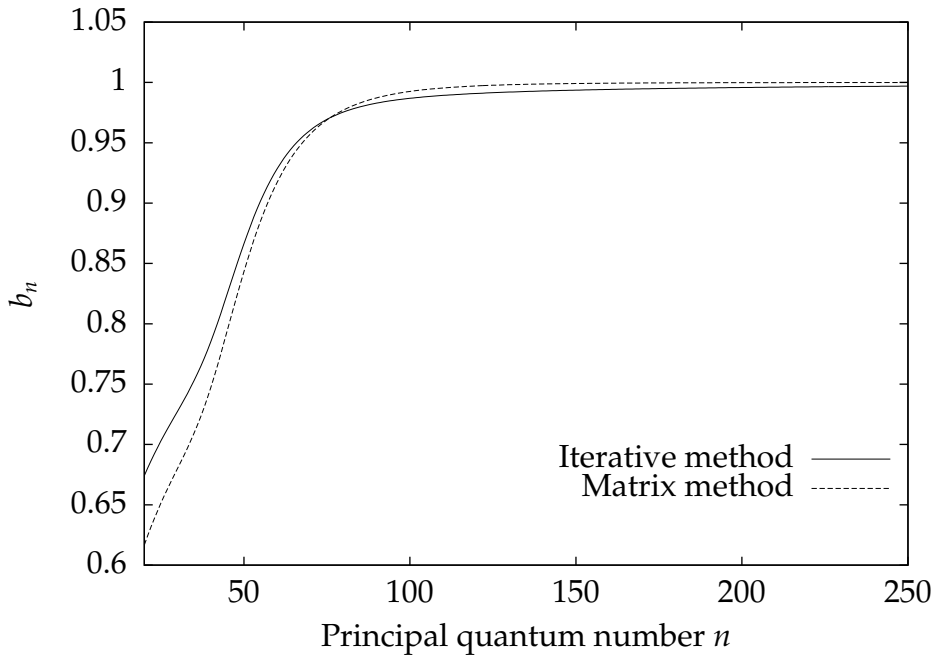

Figure 5.5: Plot of b_n for $N_e = 10^4$ and temperature $T_e = 10^4$ for the iterative method and the matrix method.

Chapter 6

Conclusion

In this thesis we have explored the computationally intensive problems of spectroscopy and the physics of the interstellar medium, in particular those relating to the problem of departure coefficients. Specifically, we have looked at these problems with the understanding that the methods of computation have changed drastically since the 1960s and 1970s, which was when a majority of the work in this area was published.

The importance of departure coefficients was understood in the 1960s when the interpretation of radio recombination lines and cosmic masers required consideration of plasmas far from the thermodynamic equilibrium (Goldberg, 1966). Observations of radio recombination lines dealt with high principal quantum numbers — up to 1000. Hence, solving the departure coefficient problem required simultaneous consideration of thousands of quantum levels, resulting in thousands of simultaneous equations of population balances. Some of these coefficients in turn required special consideration from a computational point of view — namely those involving hypergeometric functions, with terms of order 10^{1000} and higher. Approximate methods were developed for dealing with huge matrices and numbers, such as the approximation formulae used to compute the Gaunt factors and oscillator strengths, and the matrix condensation technique (Brocklehurst, 1970).

The latter was used in the solution to the b_n problem proposed by Brocklehurst and Salem (1977). They present a solution that relies on the use of the matrix condensation technique, which was based on the technique developed by Burgess and Summers (1969). This technique can condense a matrix of dimensions 1000×1000 to one of dimensions 30×30. The main reason for this

was:

> "It is obviously impracticable to invert a 1000×1000 matrix, D, so a procedure of matrix condensation based on Lagrange interpolation and extrapolation of b_n (which vary smoothly except for small n) was adopted" (Burgess & Summers, 1969, p.1010).

It is no longer the case that it is impractical to invert a 1000×1000 matrix and hence we have chosen to solve the entire system of linear equations given by the b_n problem rather than use an approximation technique that is no longer required.

Seminal papers by Storey and Hummer [(Hummer & Storey, 1987), (Hummer & Storey, 1992) and (Storey & Hummer, 1995)] which discuss a solution for n, l, also use a matrix condensation technique. Furthermore, it starts by assuming that for n, l, the l-sublevels have populations proportional to $(2l + 1)$ such that $b_n = b_{n,l}$. It then goes on to correct, through an iterative approach, for the levels of n below some n_c where this assumption is not correct. It then uses the calculated $b_{n,l}$'s for $n \leq n_c$ to calculate b_n through the equation:

$$b_n = \sum_l \frac{2l+1}{n^2} b_{n,l} \tag{6.1}$$

It is our opinion, as for the case of b_n, that approximation techniques are no longer needed, due to the increasing performance of computers. Hence we did not use this method to solve the $b_{n,l}$ problem.

The goals of this thesis were to use, for the first time, arbitrary arithmetic to solve problems involving the hypergeometric function as well as optimizing the solutions to the aforementioned problems using modern multi-core and high-performance computing architectures.

We managed to implement the exact solution to the b_n problem. Although Gaunt factors were used in this calculation, we compared them to the exact solutions we computed for the oscillator strengths, using arbitrary precision, and determined that they were very close approximations. Furthermore, we implemented exact solutions for both the Einstein coefficients and the radiative recombination coefficients for n, l.

We also managed to optimize some of the procedures to use high performance techniques and drastically increase speed, some by a factor of 30.

Future Work

Due to the large scope of both the b_n and $b_{n,l}$ problems, it was not possible to fully cover either in this thesis. Hence it would be greatly desirable to continue calculations of collisional transitions for n, l and solve the entire $b_{n,l}$ problem as well as improve the matrix method for calculation of b_n. Furthermore, a comparison of the output of these methods should be made with the methods developed by Brocklehurst and Salem (1977) and Storey and Hummer (1995) to show if the various assumptions, such as matrix condensation, are in fact valid. A large, modern area of applications for the b_n problem is connected with the development of the thermonuclear controlled fusion reactors, such as the Tokamak and ITER (Lisitsa et al., 2012). Recombination lines proved to be the only reliable tool for diagnostics of the high temperature plasmas of these power stations of the future. Consideration of the methods developed in this thesis in terms of this very important application would be highly desirable.

References

Bethe, H. A., & Salpeter, E. E. (1957). *Quantum Mechanics of One- and Two-Electron Atoms*.

Black, P. E. (2013, May). *factorial.* in *Dictionary of Algorithms and Data Structures* [online], Paul E. Black, ed., U.S. National Institute of Standards and Technology. Retrieved from `http://www.nist.gov/dads/HTML/factorial.html`

Bolton, J. G., & Stanley, G. J. (1948, February). Variable Source of Radio Frequency Radiation in the Constellation of Cygnus. *Nature*, *161*, 312-313. doi: 10.1038/161312b0

Brocklehurst, M. (1970). Level populations of hydrogen in gaseous nebulae. *Monthly Notices of the Royal Astronomical Society*, *148*, 417.

Brocklehurst, M. (1971). Calculations of level populations for the low levels of hydrogenic ions in gaseous nebulae. *Monthly Notices of the Royal Astronomical Society*, *153*, 471-490.

Brocklehurst, M., & Salem, M. (1977). Radio recombination lines from H^+ regions and cool interstellar clouds: computation of the bn factors. *Computer Physics Communications*, *13*, 39-48. doi: 10.1016/0010-4655(77)90025-X

Burgess, A. (1965). Tables of hydrogenic photoionization cross-sections and recombination coefficient. *Memoirs of the Royal Astronomical Society*, *69*, 1.

Burgess, A., & Summers, H. P. (1969, August). The Effects of Electron and Radiation Density on Dielectronic Recombination. *Astrophysical Journal*, *157*, 1007. doi: 10.1086/150131

Burgess, A., & Summers, H. P. (1976, February). The recombination and level populations of ions. I - Hydrogen and hydrogenic ions. *Monthly Notices of the Royal Astronomical Society*, *174*, 345-391.

Chandrasekhar, S. (1960). *Radiative transfer*. Dover Publ. Retrieved from `http://books.google.co.nz/books?id=zPVQAAAAMAAJ`

Cormen, T. (2009). *Introduction to algorithms*. Mit Press. Retrieved from http://books.google.co.nz/books?id=h2xRPgAACAAJ

Dopita, M. A., & Sutherland, R. S. (2003). *Astrophysics of the diffuse universe*.

Dupree, A. K. (1969, November). Radiofrequency Recombination Lines from Heavy Elements: Carbon. *Astrophysical Journal*, *158*, 491. doi: 10.1086/150213

Encyclopaedia britannica. (2013, Feb 06). Retrieved from http://www.britannica.com/EBchecked/topic/488865/radio-astronomy

Fastfactorialfunctions. (2013, Mar 12). Retrieved from http://www.luschny.de/math/factorial/FastFactorialFunctions.htm

Fousse, L., Hanrot, G., Lefèvre, V., Pélissier, P., & Zimmermann, P. (2007, June). Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw.*, *33*(2). Retrieved from http://doi.acm.org/10.1145/1236463.1236468 doi: 10.1145/1236463.1236468

Gcc 4.3 release series changes, new features, and fixes. (2013, March 04). Retrieved from http://gcc.gnu.org/gcc-4.3/changes.html

Gee, C. S., Percival, L. C., Lodge, J. G., & Richards, D. (1976, April). Theoretical rates for electron excitation of highly-excited atoms. *Monthly Notices of the Royal Astronomical Society*, *175*, 209-216.

Goldberg, L. (1966, June). Stimulated Emission of Radio-Frequency Lines of Hydrogen. *Astrophysical Journal*, *144*, 1225-1231. doi: 10.1086/148723

Gordon, M., & Sorochenko, R. (2002). *Radio recombination lines: Their physics and astronomical applications*. London. Retrieved from http://books.google.co.nz/books?id=Q-iGoBWLXIoC

Gordon, W. (1929). Zur Berechnung der Matrizen beim Wasserstoffatom. *Annalen der Physik*, *394*, 1031-1056. doi: 10.1002/andp.19293940807

Gough, B. (2009). *Gnu scientific library: Reference manual*. Network Theory. Retrieved from http://books.google.ca/books?id=CUuNPgAACAAJ

Granlund, T., & the GMP development team. (2013). GNU MP: The GNU Multiple Precision Arithmetic Library (5.1.1 ed.) [Computer software manual]. (http://gmplib.org/)

Hummer, D. G., & Storey, P. J. (1987, February). Recombination-line intensities

for hydrogenic ions. I - Case B calculations for H I and He II. *Monthly Notices of the Royal Astronomical Society*, *224*, 801-820.

Hummer, D. G., & Storey, P. J. (1992, January). Recombination line intensities for hydrogenic ions. III - Effects of finite optical depth and dust. *Monthly Notices of the Royal Astronomical Society*, *254*, 277-290.

Ieee standard for floating-point arithmetic. (2008, Aug 29). *IEEE Std 754-2008*, 1 -58. doi: 10.1109/IEEESTD.2008.4610935

Knuth, D. E. (1992). Two notes on notation. *The American Mathematical Monthly*, *99*(5), pp. 403-422. Retrieved from http://www.jstor.org/stable/2325085

Lang, K. R. (1975). *Astrophysical formulae: A compendium for the physicist and astrophysicist*. Not Avail. Retrieved from http://www.amazon.com/Astrophysical-Formulae-Compendium-Physicist-Astrophysicist/dp/3540066055%3FSubscriptionId%3D0JYN1NVW651KCA56C102%26tag%3Dtechkie-20%26linkCode%3Dxm2%26camp%3D2025%26creative%3D165953%26creativeASIN%3D3540066055

Lezar, E., & Davidson, D. (2010). Gpu-based lu decomposition for large method of moments problems. *Electronics Letters*, *46*(17), 1194-1196. doi: 10.1049/el.2010.1680

Lisitsa, V. S., Bureyeva, L. A., Kukushkin, A. B., Kadomtsev, M. B., Krupin, V. A., Levashova, M. G., . . . Vukolov, K. Y. (2012). Spectroscopic problems in iter diagnostics. *Journal of Physics: Conference Series*, *397*(1), 012015. Retrieved from http://stacks.iop.org/1742-6596/397/i=1/a=012015

Lu, M., He, B., & Luo, Q. (2010). Supporting extended precision on graphics processors. In *Proceedings of the sixth international workshop on data management on new hardware* (pp. 19–26). New York, NY, USA: ACM. Retrieved from http://doi.acm.org/10.1145/1869389.1869392 doi: 10.1145/1869389.1869392

Menzel, D. H., & Pekeris, C. L. (1935, November). Absorption coefficients and hydrogen line intensities. *Monthly Notices of the Royal Astronomical Society*, *96*, 77.

Mukunoki, D., & Takahashi, D. (2012). Implementation and evaluation of quadruple precision blas functions on gpus. In K. Jonasson (Ed.), *Applied parallel and scientific computing* (Vol. 7133, p. 249-259). Springer Berlin Heidelberg. Retrieved from

http://dx.doi.org/10.1007/978-3-642-28151-8_25 doi: 10.1007/978-3-642-28151-8_25

Murthy, G., Ravishankar, M., Baskaran, M., & Sadayappan, P. (2010, April). Optimal loop unrolling for gpgpu programs. In *Parallel distributed processing (ipdps), 2010 ieee international symposium on* (p. 1-11). doi: 10.1109/IPDPS.2010.5470423

Nakayama, T., & Takahashi, D. (2011, December). Implementation of multiple-precision floating-point arithmetic library for gpu computing. In T. Gonzalez (Ed.), *Proceedings of the 23rd iasted international conference on parallel and distributed computing and systems* (p. 343-349). Retrieved from http://www.actapress.com/PaperInfo.aspx?paperId=453078 doi: 10.2316/P.2011.757-041

Osterbrock, D., & Ferland, G. (2006). *Astrophysics of gaseous nebulae and active galactic muclei*. Univ Science Books. Retrieved from http://books.google.co.nz/books?id=6GIXMFpET4cC

Pradhan, A., & Nahar, S. (2011). *Atomic astrophysics and spectroscopy*. Cambridge University Press. Retrieved from http://books.google.co.nz/books?id=5948JMEGzm8C

Seaton, M. J. (1959a). Radiative recombination of hydrogenic ions. *Monthly Notices of the Royal Astronomical Society*, *119*, 81.

Seaton, M. J. (1959b). The solution of capture-cascade equations for hydrogen. *Monthly Notices of the Royal Astronomical Society*, *119*, 90.

Sejnowski, T. J., & Hjellming, R. M. (1969, June). The General Solution of the b_{n} Problem for Gaseous Nebulae. *Astrophysical Journal*, *156*, 915. doi: 10.1086/150024

Serre, D. (2010). *Matrices: Theory and applications*. Springer Science+Business Media, LLC. Retrieved from http://books.google.co.nz/books?id=pKKrFnINccIC

Serway, R., & Jewett, J. (2008). *Physics for scientists and engineers with modern physics: Chapters 1-46*. Brooks/Cole. Retrieved from http://books.google.co.nz/books?id=eOU9AQAAIAAJ

Shaver, P. A. (1975, July). Theoretical intensities of low frequency recombination lines. *Pramana*, *5*, 1-28. doi: 10.1007/BF02875147

Singer, J. (1959). *Masers*. Wiley. Retrieved from http://books.google.co.nz/books?id=FhMjAAAAMAAJ

Stewart, D., & Leyk, Z. (1994). *Meschach: Matrix computations in c : Version 1.2*. CMA. Retrieved from http://books.google.co.nz/books?id=4HSBQgAACAAJ (Contributor: Australian National University. Centre for Mathematics and Its Applications)

Storey, P. J., & Hummer, D. G. (1995, January). Recombination line intensities for hydrogenic ions-IV. Total recombination coefficients and machine-readable tables for Z=1 to 8. *Monthly Notices of the Royal Astronomical Society*, *272*, 41-48.

Strzodka, R., Doggett, M., & Kolb, A. (2005). Scientific computation for simulations on programmable graphics hardware. *Simulation Modelling Practice and Theory*, *13*(8), 667 - 680. Retrieved from http://www.sciencedirect.com/science/article/pii/S1569190X05000833 (<ce:title>Programmable Graphics Hardware</ce:title>) doi: 10.1016/j.simpat.2005.08.001

Stuart, J. A., & Owens, J. D. (2011). Efficient synchronization primitives for gpus. *CoRR*, *abs/1110.4623*.

System/360 model 50. (2013, May 21). Retrieved from http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP2050.html

The MPFR Team. (2013). *The mpfr library: Algorithms and proofs.* (http://www.mpfr.org/algorithms.pdf)

van de Hulst, H. C. (1951, October). Observations of the interstellar hydrogen line of wave length 21 cm made at Kootwijk, Netherlands. *Astronomical Journal*, *56*, 144-144. doi: 10.1086/106564

Xiao, S., & Feng, W. (2010). Inter-block gpu communication via fast barrier synchronization. In *Parallel distributed processing (ipdps), 2010 ieee international symposium on* (p. 1-12). doi: 10.1109/IPDPS.2010.5470477

Appendix A

Code

A.1 Auxiliary Code

```
#ifndef _BNCONSTANTSGUARD_
#define _BNCONSTANTSGUARD_
/*
* Header file that contains definitions of
* physical constants for use in departure
* coefficient calculations
*/

double const PI = 3.14159265;
double const EXP = 2.71828183;
double const h  = 6.626068e-27;
double const h_bar = (6.626068e-27)/(2*3.14159265); // h_bar = h/(2*PI)
double const k  = 1.3806503e-16;
double const m_e = 9.10938188e-28;
double const m_p = 1.672621637e-24;
double const R_inf = 109737.31568;
double const R_H = (109737.31568)/(1+(9.10938188e-28)/(1.672621637e-24));
    //R_H = R_inf/(1+m/m_p)
double const c  = 2.99792458e10;
double const Z  = 1.0;
double const q_e = -4.8032041e-10; // ESU or statCoulomb units

```

```
#endif
```

constants.h

```
#include <stdio.h>
#include <math.h>
#include <gsl/gsl_integration.h>

double epsabs = 0;
double epsrel = 1e-5;

double expint(double x)
{
  double expint = exp(-x)/x;
  return expint;
}

double ff1(double x)
{
  double ff1 = 1/x/(exp(x)-1);
  return ff1;
}

double ff2(double x, void* params)
{
  double T_r = ((double *)params)[0];
  double T_e = ((double *)params)[1];
  double ff2 = (exp(-T_r*x/T_e)/x)/(exp(x)-1);

  return ff2;
}

double calc_expint(double x_n)
{
  gsl_integration_workspace * w = gsl_integration_workspace_alloc (1000);
  double result, error;

  gsl_function EXPINT;
  EXPINT.function = (void *)&expint;
```

```
36   gsl_integration_qagiu(&EXPINT,x_n,epsabs,epsrel,1000,w,&result,&error);
37   gsl_integration_workspace_free (w);
38   return result;
39 }
40
41 double calc_ff1(double x)
42 {
43   gsl_integration_workspace* w = gsl_integration_workspace_alloc(1000);
44   double result, error;
45
46   gsl_function FF1;
47   FF1.function = (void *)&ff1;
48   gsl_integration_qagiu(&FF1,x,epsabs,epsrel,1000,w,&result,&error);
49   gsl_integration_workspace_free(w);
50
51   return result;
52 }
53
54 double calc_ff2(double x, double T_r, double T_e)
55 {
56   gsl_integration_workspace* w = gsl_integration_workspace_alloc(1000);
57
58   double result, error;
59   double params[2];
60
61   params[0] = T_r;
62   params[1] = T_e;
63
64   gsl_function FF2;
65   FF2.function = (void *)&ff2;
66   FF2.params = &params[0];
67   gsl_integration_qagiu(&FF2,x,epsabs,epsrel,1000,w,&result,&error);
68   gsl_integration_workspace_free(w);
69
70   return result;
71 }
```

expint.c

```
#include <stdlib.h>
/*
 * struct definitions
 */
// struct Program_kernel stores a kernel along with its size
typedef struct
{
  char**  kernel;
  int size;
} Program_kernel;

/*
 * method declarations
 */
Program_kernel* loadKernel(const char* filename);
void readKernelFromTextFile(FILE* pFile, Program_kernel* pgmKernel);
```

kernelReader.h

```
/******************
 * kernelReader.c:
 * - 'loadKernel' uses 'readKernelFromTextFile' to load an OpenCL
 *    kernel into dynamic memory from a text file.
 * - The kernel is stored in the typdef struct Program_kernel
 *    defined in 'kernelReader.h'.
 * - Returns a Program_kernel*.
 ******************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "kernelReader.h"

Program_kernel* loadKernel(const char* filename)
{
  FILE* pFile;

  pFile = fopen(filename, "r");

```

```
21   Program_kernel* pgmKernel =
         (Program_kernel*)malloc(sizeof(Program_kernel));
22
23   if(pFile != NULL)
24   {
25     readKernelFromTextFile(pFile, pgmKernel);
26   }
27   else
28   {
29     puts("Error loading kernel");
30   }
31
32   fclose(pFile);
33
34   return pgmKernel;
35 }
36
37 void readKernelFromTextFile(FILE* pFile, Program_kernel* pgmKernel)
38 {
39   int size = 0;
40   char** textMatrix = (char**)malloc(size*sizeof(char*));
41   int index = 0;
42   int lineSize = 1024; // Expected maximum size of each line
43                        // in text file
44   char currentString[lineSize];
45
46   while(!feof(pFile))
47   {
48   fgets(currentString, lineSize, pFile);
49
50        size += sizeof(char*);
51
52        textMatrix = (char**)realloc(textMatrix, size);
53        textMatrix[index] = strdup(currentString);
54        index++;
55   }
56
57   pgmKernel->kernel = textMatrix;
```

```
58   // HACK: must remove last char* pointer as loop goes one line further
         than it should i.e. goes one line past end of file.
59   pgmKernel->size = (size-sizeof(char*));
60 }
```

kernelReader.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <math.h>
4  #include "constants.h"
5
6  double oscillator_strength(int n, int m);
7  double gaunt_approximation(int n, int m);
8  double f_kramer_func(int n, int m);
9
10 double oscillator_strength(int n, int m)
11 {
12         double gaunt_factor = gaunt_approximation(n,m);
13         double f_kramer  = f_kramer_func(n,m);
14         double f_real    = f_kramer*gaunt_factor;
15
16         return f_real;
17 }
18
19 double gaunt_approximation(int n, int m)
20 {
21         double G_1 = (0.203+0.256/pow(m,2)+0.257/pow(m,4))*m;
22         double G_2 = 0.170*m+0.18;
23         double G_3 = (0.2214+0.1554/pow(m,2)+0.370/pow(m,4))*m;
24         double T_1 = (2*n-m)*(n-m+1);
25         double T_2 = 4.0*(n-1)*(m-n-1);
26         double T_3 = (2*n-m-0.001)*(n-0.999);
27         double T_4 = (1/pow(m-1.999,2))*(1/(m*pow(n,(2.0/3))))
28                       *pow((m-1.0)/(m-n),(2.0/3));
29
30         double gaunt_factor = 1.0-T_4*(T_1*G_1+T_2*G_2+T_3*G_3);
31
32         return gaunt_factor;
```

```
}

double f_kramer_func(int n, int m)
{
        return 1.9603/pow(m*m-n*n,3)*pow(m,3)*n;
}
```

oscillator_strength_gaunt_final.c

A.2 Einstein Coefficients

```
/*******************************************
 * file: 'einstein_coefficient_calc_mpfr.c' *
 *******************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <mpfr.h>
#include <gmp.h>
#include "constants.h"

// Global variables
FILE* fp;
const char* FILENAME = "A.dat";

// Function declarations
void calc_A(mpfr_t* A, int n, int m, int l);
void calc_B(mpfr_t* B, int n, int m, int l);
void calc_C(mpfr_t* C, int n, int m);
void calc_2F1(int a, int b, int g, double X, mpfr_t* hyp_ser_ptr);
double calc_A_nl_ml_p(int n, int l, int m, int l_p, int write);
double calc_A_nl_ml_p_f(int n, int l, int m, int l_p);
double calc_A_nl_ml_p_no_file(int n, int l, int m, int l_p);
double calc_A_nl_ml_p_w_file(int n, int l, int m, int l_p);
double calc_A_nl_ml_p_r_file(int n, int l, int m, int l_p);
double calc_rho_squared(int n, int m, int l);
```

```
28
29 int open_file(const char* FILENAME, FILE** fp)
30 {
31     int write = -2;
32
33     printf("Trying to open '%s'\n",FILENAME);
34     if((*fp = fopen(FILENAME,"r")))
35     {
36             //File exists - set for reading
37             write = 0;
38             printf("File is ready for reading\n");
39     }
40     else if((*fp = fopen(FILENAME,"w")))
41     {
42             //File does not exist - set for writing
43             write = 1;
44             printf("File is ready for writing\n");
45     }
46     else
47     {
48             printf("Could not open file for ");
49             if(write)
50             {
51                     printf("writing.\n");
52             }
53             else
54             {
55                     printf("reading.\n");
56             }
57             printf("Will calculate function without saving\n");
58             write = -1;
59     }
60
61     return write;
62 }
63
64 int input_accepter()
65 {
```

```
66    char input[256];
67    gets(input);
68    int n = atoi(input);
69
70    while(!(n+1))
71    {
72
73          printf("You did not enter a number correctly, please enter
                 again:\n");
74          gets(input);
75          n = atoi(input);
76    }
77    return n;
78 }
79
80
81 int main()
82 {
83   int write;
84
85   printf("Please enter desired precision:\n");
86   int prec = input_accepter();
87
88   printf("Please enter n:\n");
89   int n = input_accepter();
90
91     mpfr_set_default_prec(prec);
92
93   system("clear");
94
95
96   time_t t0, t1; // time_t is defined in <time.h> and <sys/types.h>
97                  // as long
98   clock_t c0, c1; // clock_t is defined in <time.h> and <sys/types.h>
99                   // as int
100
101  double A;
102  int i,j,k;
```

```
103
104
105 // Open file
106   printf("Write to file?\n");
107   printf("Press 1 to write or read\n");
108   printf("Press 0 to re-calculate\n");
109   if(input_accepter())
110   {
111         write = open_file(FILENAME,&fp);
112   }
113   else
114   {
115         write = -1;
116   }
117
118   printf("Starting calculation of A_%d...\n",n);
119   // Start timing
120   t0 = time(NULL);
121   c0 = clock();
122
123   // Start of Calculation
124   //=======================================================
125
126 double    A_val = -1.0;
127 int           m = n;
128 char append_flag = 0;
129
130 // Loop calculates all allowed Einstein coefficients
131 // according to quantum selection rules.
132 for(n=1;n<=m;n++)
133 {
134   // Print time taken to calculate n for multiples of 10
135   printf("n = %d\n",n);
136   if(n%10==0)
137   {
138     c1 = clock();
139     printf("%f\n",(float)(c1 - c0)/CLOCKS_PER_SEC);
140   }
```

```
141     for(i=1;i<n;i++)
142     {
143       for(j=0;j<n;j++)
144       {
145         if(j+1<i && j+1>=0)
146         {
147           A_val = calc_A_nl_ml_p(n,j,i,j+1,write);
148 
149           // We must check end of file AFTER calc_A has
150           // tried to write
151                   if(feof(fp))
152                   {
153                           printf("Appending\n");
154                           fclose(fp);
155                           fp = fopen(FILENAME,"a+");
156                           write = 1;
157                           j--;
158                   }
159 
160           if(j-1<i && j-1>=0)
161           {
162             A_val = calc_A_nl_ml_p(n,j,i,j-1,write);
163           }
164         }
165         else if(j-1<i && j-1>=0)
166         {
167           A_val = calc_A_nl_ml_p(n,j,i,j-1,write);
168         }
169 
170         // We must check end of file AFTER calc_A has tried to write
171         if(feof(fp))
172         {
173           printf("Appending\n");
174           fclose(fp);
175           fp = fopen(FILENAME,"a+");
176           write = 1;
177           j--;
178         }
```

```
179
180       }
181     }
182   }
183
184     //=====================================================
185     // End  of Calculation
186
187     // Measure elapsed time
188     t1 = time(NULL);
189     c1 = clock();
190
191     printf("\n");
192     printf("\n");
193     printf("\n");
194     printf("Done calculating A_%d:\n",n);
195     printf("Total time was:\n");
196     printf ("\telapsed wall clock time: %ld\n", (long) (t1 - t0));
197     printf ("\telapsed CPU time:   %f\n", (float) (c1 -
            c0)/CLOCKS_PER_SEC);
198     printf("\n");
199
200     //close file - if open
201     if(write != -1)
202           fclose(fp);
203 }
204
205 double calc_A_nl_ml_p(int n, int l, int m, int l_p, int write)
206 {
207     double A = 0.0;
208     if(write == 1)
209     {
210         // Calculate Einstein Coefficient and save in dat file
211     A = calc_A_nl_ml_p_w_file(n,l,m,l_p);
212     }
213     else if(write == 0)
214     {
215         // Read Einstein Coefficient from dat file
```

```
216     A = calc_A_nl_ml_p_r_file(n,l,m,l_p);
217     }
218     else
219     {
220         // Calculate Einstein Coefficient without saving value
221     A = calc_A_nl_ml_p_no_file(n,l,m,l_p);
222     }
223 
224     return A;
225 }
226 
227 double calc_A_nl_ml_p_no_file(int n, int l, int m, int l_p)
228 {
229     double A = calc_A_nl_ml_p_f(n,l,m,l_p);
230     return A;
231 }
232 
233 
234 double calc_A_nl_ml_p_r_file(int n, int l, int m, int l_p)
235 {
236     double A;
237     fscanf(fp,"%lfe\n",&A);
238     return A;
239 }
240 
241 /*
242  * Function that, along with calculating Theta, also stores in a
243  * file to avoid costly recalculation. NOTE: File-pointer must
244  * point to an open and ready file.
245  */
246 double calc_A_nl_ml_p_w_file(int n, int l, int m, int l_p)
247 {
248     // fp is open and ready to avoid overhead of re-opening
249     // file with every call to calc_Theta_w_file
250     double A = calc_A_nl_ml_p_f(n,l,m,l_p);
251     fprintf(fp, "%fe ",A);
252 
253     return A;
```

```
254 }
255
256 double calc_A_nl_ml_p_f(int n, int l, int m, int l_p)
257 {
258   // Must swap n,m,l and l' according to quantum selection rules
259   int n_rho = n;
260   int m_rho = m;
261   double l_max;
262   double rho_2;
263   if(l>=l_p)
264   {
265     l_max = l;
266   }
267   else
268   {
269     l_max = l_p;
270     n_rho = m;
271     m_rho = n;
272   }
273
274   rho_2        = calc_rho_squared(n_rho,m_rho,l_max);
275   double a_nl_md = pow(1/pow(m,2)-1/pow(n,2),3)*l_max/(2*l+1)*rho_2;
276   double A_nl_md = 2.6774e9*a_nl_md;
277
278   return A_nl_md;
279 }
280
281 double calc_rho_squared(int n, int m, int l)
282 {
283   mpfr_t A,B,C,F1,F2,temp1,answer;
284   mpfr_init(A);
285   mpfr_init(B);
286   mpfr_init(C);
287   mpfr_init(F1);
288   mpfr_init(F2);
289   mpfr_init(temp1);
290   mpfr_init(answer);
291
```

```
292   calc_A(&A,n,m,l);
293   calc_B(&B,n,m,l);
294   calc_C(&C,n,m);
295   calc_2F1(-n+l+1,-m+l,2*l,-4*n*m/pow(n-m,2),&F1);
296   calc_2F1(-n+l-1,-m+l,2*l,-4*n*m/pow(n-m,2),&F2);
297
298   mpfr_mul(answer,C,F2,MPFR_RNDN);
299   mpfr_sub(answer,F1,answer,MPFR_RNDN);
300   mpfr_mul(answer,answer,B,MPFR_RNDN);
301   mpfr_mul(answer,answer,A,MPFR_RNDN);
302   // answer = A*B*[F1-C*F2]
303
304   mpfr_pow_ui(answer,answer,2,MPFR_RNDN);
305
306   double answer_d = mpfr_get_d(answer,MPFR_RNDN);
307
308   // Clean-up
309   mpfr_clear(A);
310   mpfr_clear(B);
311   mpfr_clear(C);
312   mpfr_clear(F1);
313   mpfr_clear(F2);
314   mpfr_clear(temp1);
315   mpfr_clear(answer);
316
317   return answer_d;
318 }
319
320 void calc_A(mpfr_t* A, int n, int m, int l)
321 {
322   mpfr_t temp1;
323   mpfr_t temp2;
324   mpfr_t temp3;
325   mpfr_t temp4;
326   mpfr_t temp5;
327
328   mpfr_init(temp1);
329   mpfr_init(temp2);
```

```
330  mpfr_init(temp3);
331  mpfr_init(temp4);
332  mpfr_init(temp5);
333  mpfr_fac_ui(temp1, n+l, MPFR_RNDN);
334  mpfr_fac_ui(temp2, m+l-1, MPFR_RNDN);
335  mpfr_fac_ui(temp3, n-l-1, MPFR_RNDN);
336  mpfr_fac_ui(temp4, m-l, MPFR_RNDN);
337  mpfr_div(temp1,temp1,temp3,MPFR_RNDN);
338  mpfr_div(temp2,temp2,temp4,MPFR_RNDN);
339  mpfr_mul(temp5,temp1,temp2,MPFR_RNDN);
340  mpfr_sqrt(temp5,temp5,MPFR_RNDN);
341  // temp5 = sqrt([(n+l)!(m+l-1)!]/[(n-l-1)!(m-l)!])
342
343  mpfr_set_ui(temp1,4*n*m,MPFR_RNDN);
344  mpfr_pow_ui(temp1,temp1,l+1,MPFR_RNDN);
345  mpfr_set_ui(temp2,n+m,MPFR_RNDN);
346  mpfr_pow_ui(temp2,temp2,n+m,MPFR_RNDN);
347  mpfr_div(temp1,temp1,temp2,MPFR_RNDN);
348
349  // temp1 = [(4*n*m)^(l+1)]/[(n+m)^(n+m)]
350  mpfr_set_si(temp2,-1,MPFR_RNDN);
351  mpfr_pow_ui(temp2,temp2,m-l,MPFR_RNDN);
352  // temp2 = (-1)^(m-l)
353  mpfr_fac_ui(temp3, 2*l-1, MPFR_RNDN);
354  mpfr_mul_ui(temp3, temp3, 4, MPFR_RNDN);
355  // temp3 = 4(2l-1)!
356  mpfr_div(temp2, temp2, temp3, MPFR_RNDN);
357  // temp2 = [(-1)^(m-1)]/[4(2l-1)!]
358  mpfr_mul(temp1, temp1, temp2, MPFR_RNDN);
359  mpfr_mul(temp1, temp1, temp5, MPFR_RNDN);
360  // temp1 = [(-1)^(m-1)]/[4(2l-1)!]*sqrt([(n+l)!(m+l-1)!]
361  //        /[(n-l-1)!(m-l)!])*[(4*n*m)^(l+1)]/[(n+m)^(n+m)]
362
363  mpfr_set(*A,temp1,MPFR_RNDN);
364
365  // Clean-up
366  mpfr_clear(temp1);
367  mpfr_clear(temp2);
```

```
368   mpfr_clear(temp3);
369   mpfr_clear(temp4);
370   mpfr_clear(temp5);
371 }
372
373 void calc_B(mpfr_t* B, int n, int m, int l)
374 {
375   mpfr_t temp1;
376   mpfr_init(temp1);
377   double exponent;
378   double base;
379
380   exponent = n+m-2*l-2;
381   base     = n-m;
382
383   mpfr_set_si(temp1,n-m,MPFR_RNDN);
384   mpfr_pow_ui(temp1,temp1,exponent,MPFR_RNDN);
385
386   mpfr_set(*B,temp1,MPFR_RNDN);
387
388   // Clean-up
389   mpfr_clear(temp1);
390 }
391
392 void calc_C(mpfr_t* C, int n, int m)
393 {
394   mpfr_t temp1, temp2;
395   mpfr_init(temp1);
396   mpfr_init(temp2);
397
398   mpfr_set_si(temp1,n-m,MPFR_RNDN);
399   mpfr_set_ui(temp2,n+m,MPFR_RNDN);
400   mpfr_div(temp1,temp1,temp2,MPFR_RNDN);
401   mpfr_pow_ui(temp1,temp1,2,MPFR_RNDN);
402
403   mpfr_set(*C,temp1,MPFR_RNDN);
404
405   // Clean-up
```

```
406   mpfr_clear(temp1);
407   mpfr_clear(temp2);
408 }
409
410 void calc_2F1(int a, int b, int g, double X, mpfr_t* hyp_ser_ptr)
411 {
412     long double ab_sum   = a+b;
413     long double ab_sum_prod = a*b-a-b+1;
414     long double l2_sub_1 = g-1;
415
416     mpfr_t chi;
417     mpfr_init(chi);
418     mpfr_set_ld(chi,X,MPFR_RNDN);
419
420     // Find largest, i.e. least negative of a,b
421     int num_iter;
422
423     if(a < 0 && b < 0)
424     {
425         if(a>b)
426         {
427             num_iter = -a;
428         }
429         else
430         {
431             num_iter = -b;
432         }
433     }
434
435   if(a == 0 || b == 0)
436   {
437     num_iter = 0;
438   }
439
440     int i;
441
442     mpfr_t F_1_mpfr[num_iter+1];
443     mpfr_init(F_1_mpfr[0]);
```

```
444     mpfr_set_si(F_1_mpfr[0],1,MPFR_RNDN);
445
446     mpfr_t temp;
447     mpfr_init(temp);
448
449     mpfr_t sum;
450     mpfr_init(sum);
451     mpfr_set(sum,F_1_mpfr[0],MPFR_RNDN);
452
453     mpfr_t numerator;
454     mpfr_t denominator;
455     mpfr_init(numerator);
456     mpfr_init(denominator);
457
458   // Hypergeometric series progresses as:
459   // t1 = 1 =>
460   // 2F1_t1 = t1
461   // .........
462   // t2 = [(ab)/(1!2l)]X = [(ab)/(2l)]X =>
463   // 2F1_t2 = t1+t1*t2
464   // .........
465   // t3 = [a(a+1)b(b+1)/(2!(2l+1)]X^2 = [(a+1)(b+1)/(2l+1)]X*[(ab)/(2l)]X
466   // 2F1_t3 = t1+t1*t2+t1*t2*t3
467   // ..... and so on
468     for(i=1;i<=num_iter;i++)
469     {
470         mpfr_init(F_1_mpfr[i]);
471
472         mpfr_set_ld(numerator,ab_sum_prod+i*(ab_sum)-2*i+i*i,MPFR_RNDN);
473     // numerator = ab-a-b+1+i(a+b)-2i+i^2
474         mpfr_set_ld(denominator,i*(l2_sub_1+i),MPFR_RNDN);
475     // denominator = i(2l-1+i)
476         mpfr_div(temp,numerator,denominator,MPFR_RNDN);
477     // temp = (ab-a-b+1+i(a+b)-2i+i^2)/(i(2l-1+i))
478         mpfr_mul(temp,temp,chi,MPFR_RNDN);
479     // temp = (ab-a-b+1+i(a+b)-2i+i^2)/(i(2l-1+i))*X
480         mpfr_mul(F_1_mpfr[i],F_1_mpfr[i-1],temp,MPFR_RNDN);
481     // F_1_mpfr[i] = F_1_mpfr[i-1]*temp
```

```
482         mpfr_add(sum,sum,F_1_mpfr[i],MPFR_RNDN);
483     }
484 
485     mpfr_set(*hyp_ser_ptr,sum,MPFR_RNDN);
486 
487     // Clean-up
488     for(i=0;i<num_iter;i++)
489     {
490           mpfr_clear(F_1_mpfr[i]);
491     }
492     mpfr_clear(temp);
493     mpfr_clear(sum);
494     mpfr_clear(numerator);
495     mpfr_clear(denominator);
496     mpfr_clear(chi);
497 }
```

einstein_coefficient_calc_mpfr.c

A.3 Radiative Recombination

```
1  long double g_nl_Kl_p(int n, int l, double K, int l_p);
2  long double G_n_l_K_lp(int n, int l, double K, int l_p);
3  long double G_n_l_K_lg(int n, int l, double K);
4  long double G_n_l_K_ls(int n, int l, double K);
5  long double G_n_n_1_0_n(int n);
6  long double G_n_n_1_K_n(int n, double K);
7  long double G_n_n_2_K_n_1(int n, double K);
8  long double G_n_n_1_K_n_2(int n, double K);
9  long double G_n_n_2_K_n_3(int n, double K);
10 long double calc_A_g(int n, int l, double K);
11 long double calc_B_g(int n, int l, double K);
12 long double calc_A_s(int n, int l, double K);
13 long double calc_B_s(int n, int l, double K);
14 long double factorial_minus_lower(int n, int lower_limit);
15 long double calc_Theta(int n, int l, double K, int l_p, int write, FILE*
       fp);
16 long double calc_Theta_r_file(int n, int l, double K, int l_p, FILE* fp);
```

```
17 long double calc_Theta_w_file(int n, int l, double K, int l_p, FILE* fp);
18 long double calc_Theta_no_file(int n, int l, double K, int l_p);
19 int max(int n1, int n2);
20 int input_accepter();
21 int open_file(const char* FILENAME, FILE** fp);
```

radiative_recombination_long_double.h

```
1 /************************************************
2  * file: 'radiative_recombination_long_double.c' *
3  ************************************************/
4
5 #include <CL/cl.h>
6 #include <errno.h>
7 #include <string.h>
8 #include <sys/types.h>
9 #include <sys/resource.h>
10 #include <time.h>
11 #include <unistd.h>
12 #include <stdio.h>
13 #include <stdlib.h>
14 #include <math.h>
15 #include "radiative_recombination_long_double.h"
16 #include "constants.h"
17 #include "kernelReader.h"
18
19 // Global variables
20 FILE* fp;
21 const char* FILENAME = "dat_rad_rec.dat";
22 const int N0=2; // Case B
23 int N=500;
24 const double T_e = 10000.0;
25 const int num_Iter = 100;
26 const double h_factor = 0.000000025;
27
28 double calc_alpha_nl(int n, int l, int iter, double y, double h_factor,
      double alpha, double* theta);
29 double calc_I_y(int n, int l, int l_p, double y, int iter, double
      h_factor, double* theta);
```

```
30 double calc_I_integral_y(int n, int l, int l_p, int iter, double y,
       double h_factor, double* theta);
31 // boole's integral as given on:
       http://en.wikipedia.org/wiki/Boole%27s_rule
32 double boole_integral_y(int n, double x1, double x5, double h, double
       (*i_f)(int n, double K, double y, double* theta, int theta_index),
       double y, double* theta, int theta_index);
33 double integrand_y(int n, double K2, double y, double* theta, int
       theta_index);
34 void execute_open_cl_kernel(int N0, double y, double h_factor, double
       alpha, int num_Iter, double* theta, int theta_size, double* alpha_nl,
       int alpha_nl_size, int* alpha_n_index);
35 void execute_cpu_kernel(double y, double h_factor, double alpha, int
       num_Iter, double* theta, int alpha_nl_size, double* alpha_nl_array);
36 void print_matrix(double* MAT, int alpha_nl_size);
37
38 int main()
39 {
40   // SETUP TIMER
41   clock_t gc0, gc1;
42   clock_t c0, c1;
43   gc0 = clock();
44
45   // Increase stack size
46   struct rlimit old_lim;
47   getrlimit(RLIMIT_STACK,&old_lim);
48   printf("old_lim_cur = %lld\n",(long long)old_lim.rlim_cur);
49   printf("old_lim_max = %lld\n",(long long)old_lim.rlim_max);
50   long long newLim = 10000*old_lim.rlim_cur;
51   struct rlimit new_lim = {newLim,newLim};
52
53   // Set new limit in BYTES (4 times the original stack size)
54   setrlimit(RLIMIT_STACK,&new_lim);
55   printf("new_lim_max = %lld\n",(long long)new_lim.rlim_max);
56
57   printf("Please enter N:\n");
58   N = input_accepter();
59
```

```
60  int write;
61
62  // Open file
63  printf("Write to file?\n");
64  printf("Press 1 to write or read\n");
65  printf("Press 0 to re-calculate\n");
66  if(input_accepter())
67  {
68    write = open_file(FILENAME,&fp);
69  }
70  else
71  {
72    write = -1;
73  }
74
75  int theta_index = 0; // This is the base offset in the theta_array.
      num_Iter values of theta will be needed.
76  int n,l,l_counter,k_index,intg_index;
77  double K2 = 0.0;
78  double h_m = 0.0;
79  int l_p;
80
81  int alpha_nl_size = ((N-1)*(N+2))/2;
82  int theta_size = N*(N-1)*num_Iter*5;
83
84  double alpha_nl_gpu[alpha_nl_size];
85  double alpha_nl_cpu[alpha_nl_size];
86  double theta[theta_size];
87
88  // Population of Theta
89  for(n=N0;n<=N;n++)
90  {
91    //printf("n = %d\n",n);
92    for(l=0;l<n;l++)
93    {
94      for(l_counter=0;l_counter<2;l_counter++)
95      {
96        // Reset K every time as this is the integration variable.
```

```
97        K2 = 0.0;
98        h_m = h_factor/n;
99
100       if(l_counter == 0)
101         l_p = l-1;
102       else
103         l_p = l+1;
104
105       // Ensure quantum selection rules are upheld i.e.
106       // l_p >= 0 and l_p < n
107       if(l_p>=0 && l_p<n)
108       {
109         theta[theta_index] = (double)calc_Theta(n,l,0.0,l_p,write,fp);
110         theta_index++;;
111         K2 = h_m;
112
113         for(k_index=0;k_index<num_Iter;k_index++)
114         {
115           for(intg_index=1;intg_index<=4;intg_index++)
116           {
117             theta[theta_index] =
                  (double)calc_Theta(n,l,sqrt(K2),l_p,write,fp);
118             K2 += h_m;
119             theta_index++;
120           }
121           K2 += h_m;
122           h_m = 2*h_m;
123         }
124       }
125     }
126   }
127 }
128
129 // Variables needed to calculate alpha_nl
130 double y    = 15.778e4/T_e;
131 double alpha = 1/137.035999074; // Fine-structure constant
132
133 int i;
```

```
134  n = 2; // n = N0
135  int alpha_n_index[alpha_nl_size];
136
137  for(i=0;i<alpha_nl_size;)
138  {
139    for(l=0;l<n;l++)
140    {
141      alpha_n_index[i] = n;
142      i++;
143    }
144    n++;
145  }
146
147  printf("Calculating GPU\n");
148  // Start TIMER
149  c0 = clock();
150
151    execute_open_cl_kernel(N0,y,h_factor,alpha,num_Iter,theta,theta_size,
152      alpha_nl_gpu,alpha_nl_size, alpha_n_index);
153
154  c1 = clock();
155  printf ("\tGPU total time:   %f\n", (float) (c1 - c0)/
156    CLOCKS_PER_SEC);
157
158  printf("Calculating CPU\n");
159  // Start TIMER
160  c0 = clock();
161
162    execute_cpu_kernel(y,h_factor,alpha,num_Iter,theta,alpha_nl_size,
163      alpha_nl_cpu);
164
165  c1 = clock();
166  printf ("\tCPU total time:   %f\n", (float) (c1 - c0)/CLOCKS_PER_SEC);
167
168  gc1 = clock();
169  printf ("Total global time:   %f\n", (float) (gc1 - gc0)/
170    CLOCKS_PER_SEC);
171  printf("Done\n");
```

```
172 }
173
174 void execute_cpu_kernel(double y, double h_factor, double alpha, int
        num_Iter, double* theta, int alpha_nl_size, double* alpha_nl_array)
175 {
176   int i,l;
177   int n = 2;
178   //printf("\n");
179   for(i=0;i<alpha_nl_size;)
180   {
181     for(l=0;l<n;l++)
182     {
183           alpha_nl_array[i] =
                calc_alpha_nl(n,l,num_Iter,y,h_factor,alpha,theta);
184           i++;
185     }
186     n++;
187   }
188 }
189
190 void print_matrix(double* MAT, int alpha_nl_size)
191 {
192   int i,l;
193   int n = 2;
194   printf("\n");
195   for(i=0;i<alpha_nl_size;)
196   {
197     for(l=0;l<n;l++)
198     {
199       if(i>=alpha_nl_size-N)
200       printf("%e\n",MAT[i]);
201       i++;
202     }
203     n++;
204   }
205   printf("\n");
206 }
207
```

```
208 void execute_open_cl_kernel(int N0, double y, double h_factor, double
      alpha, int num_Iter, double* theta, int theta_size, double* alpha_nl,
      int alpha_nl_size, int* alpha_n_index)
209 {
210   // DATA INIT
211   int err;
212   size_t global[1];
213   cl_device_id device_id[100];
214   cl_context context;
215   cl_command_queue commands;
216   cl_program program;
217   cl_kernel kernel;
218   cl_uint nd;
219   cl_mem alpha_n_index_in, theta_in, alpha_nl_out;
220
221   // PLATFORM SETUP
222   cl_platform_id platforms[100];
223   cl_uint platforms_n = 0;
224   cl_uint devices_n = 0;
225   clGetPlatformIDs(100, platforms, &platforms_n);
226   if(platforms_n == 0)
227         puts("no devices found");
228   err = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, 100, device_id,
        &devices_n);
229
230   //context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
231   context = clCreateContext(NULL, 1, device_id, NULL, NULL, &err);
232   commands = clCreateCommandQueue(context, device_id[0], 0, &err);
233
234   // SETUP buffers and write "alpha_n_index" and "theta_in"
235   // to the device memory
236   alpha_n_index_in        = clCreateBuffer(context, CL_MEM_READ_ONLY ,
        sizeof(int) * alpha_nl_size , NULL, NULL);
237   theta_in                = clCreateBuffer(context, CL_MEM_READ_ONLY ,
        sizeof(double) * theta_size , NULL, NULL);
238   alpha_nl_out            = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
        sizeof(double) * alpha_nl_size, NULL, NULL);
```

```
239   err             = clEnqueueWriteBuffer(commands, alpha_n_index_in,
          CL_TRUE, 0, sizeof(int) * alpha_nl_size, alpha_n_index, 0, NULL,
          NULL);
240   err             = clEnqueueWriteBuffer(commands, theta_in , CL_TRUE, 0,
          sizeof(double) * theta_size, theta , 0, NULL, NULL);
241
242   // BUILD the program, define the kernel and setup arguments
243   Program_kernel* pgmKernel = loadKernel("rad_kernel.cl");
244   const char** program_source = (const char**)pgmKernel->kernel;
245   int pgmSize             = pgmKernel->size;
246   program                 = clCreateProgramWithSource(context,
          pgmSize/sizeof(*program_source), program_source, NULL, &err);
247   err                     = clBuildProgram(program, 0, NULL, NULL, NULL,
          NULL);
248
249   // SETUP Kernel
250   kernel = clCreateKernel(program, "calc_alpha_nl", &err);
251   err   = clSetKernelArg(kernel, 0, sizeof(int) , &N0);
252   err  |= clSetKernelArg(kernel, 1, sizeof(double) , &y);
253   err  |= clSetKernelArg(kernel, 2, sizeof(double) , &h_factor);
254   err  |= clSetKernelArg(kernel, 3, sizeof(double) , &alpha);
255   err  |= clSetKernelArg(kernel, 4, sizeof(double) , &PI);
256   err  |= clSetKernelArg(kernel, 5, sizeof(double) , &a_0);
257   err  |= clSetKernelArg(kernel, 6, sizeof(double) , &c);
258   err  |= clSetKernelArg(kernel, 7, sizeof(int) , &num_Iter);
259   err  |= clSetKernelArg(kernel, 8, sizeof(cl_mem), &theta_in);
260   err  |= clSetKernelArg(kernel, 9, sizeof(cl_mem), &alpha_nl_out);
261   err  |= clSetKernelArg(kernel,10, sizeof(cl_mem), &alpha_n_index_in);
262
263   // RUN the kernel and collect results
264   global[0] = (size_t)alpha_nl_size;
265   nd = 1;
266   err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global, NULL,
          0, NULL, NULL);
267   clFinish(commands);
268   err = clEnqueueReadBuffer(commands, alpha_nl_out, CL_TRUE, 0,
          sizeof(double) * alpha_nl_size, alpha_nl, 0, NULL, NULL);
269 }
```

```
270
271 long double calc_Theta(int n, int l, double K, int l_p, int write, FILE*
        fp)
272 {
273   long double Theta = 0.0;
274   if(write == 1)
275   {
276     // Calculate theta and save in dat file
277     Theta = calc_Theta_w_file(n,l,K,l_p,fp);
278   }
279   else if(write == 0)
280   {
281     // Read Theta from dat file
282     Theta = calc_Theta_r_file(n,l,K,l_p,fp);
283   }
284   else
285   {
286     // Calculate Theta without saving value
287     Theta = calc_Theta_no_file(n,l,K,l_p);
288   }
289
290   return Theta;
291 }
292
293 long double calc_Theta_no_file(int n, int l, double K, int l_p)
294 {
295   long double g = g_nl_Kl_p(n,l,K,l_p);
296   return (1+n*n*K*K)*g*g;
297 }
298
299
300 long double calc_Theta_r_file(int n, int l, double K, int l_p, FILE* fp)
301 {
302   long double T;
303   fscanf(fp,"%Le\n",&T);
304   return T;
305 }
306
```

```
307 // Function that, along with calculating Theta, also stores in a file to
        avoid
308 // costly recalculation. NOTE: File-pointer must point to an open and
        ready
309 // file.
310 long double calc_Theta_w_file(int n, int l, double K, int l_p, FILE* fp)
311 {
312   // fp is open and ready to avoid overhead of re-opening
313   // file with every call to calc_Theta_w_file
314   long double T = calc_Theta_no_file(n,l,K,l_p);
315   fprintf(fp, "%Le ",T);
316
317   return T;
318 }
319
320 // Calculates g(n,l;K,l_p)
321 long double g_nl_Kl_p(int n, int l, double K, int l_p)
322 {
323   long double product = 1.0;
324   long double G   = G_n_l_K_lp(n,l,K,l_p);
325   long double fact;
326
327   int s;
328   for(s=1;s<=l_p;s++)
329   {
330     // Take square root to make number smaller
331     product *= sqrtl(1+s*s*K*K);
332   }
333
334   // fact = (n+l)!/(n-l-1)!
335   fact = factorial_minus_lower(n+l,n-l-1);
336   fact = sqrtl(fact);
337
338   // return: g(nl,Kl') =
339   // sqrt[(n+l)!/(n-l-1)!)*\product_{s=0}^{l'}(1+s^2*K^2)]*
340   // (2n)^(l-n)*G(n,l,K,l')
341   return fact*product*powl(2*n,l-n)*G;
342 }
```

```
343
344 long double G_n_l_K_lp(int n, int l, double K, int l_p)
345 {
346   long double G;
347
348   if(l==(n-1) && l_p==n)
349   {
350     if(K==0.0)
351     {
352       G = G_n_n_1_0_n(n);
353     }
354     else
355     {
356       G = G_n_n_1_K_n(n,K);
357     }
358   }
359   else if(l==(n-2) && l_p==(n-1))
360   {
361     G = G_n_n_2_K_n_1(n,K);
362   }
363   else if(l==(n-1) && l_p==(n-2))
364   {
365     G = G_n_n_1_K_n_2(n,K);
366   }
367   else if(l==(n-2) && l_p==(n-3))
368   {
369     G = G_n_n_2_K_n_3(n,K);
370   }
371   else
372   {
373     if(l==(l_p+1))
374     {
375       G = G_n_l_K_lg(n,l,K);
376     }
377     else if(l==(l_p-1))
378     {
379       G = G_n_l_K_ls(n,l,K);
380     }
```

```
381     else
382     {
383       printf("Error - incorrectly formatted l and l'\n");
384       printf("l and l' are not in the format of:\n");
385       printf("l' = l+1 or l' = l-1\n");
386     }
387   }
388
389   return G;
390 }
391
392
393 // Calculates G(n,l,K,l') = G(n,l,K,l-1) i.e. 'g' is 'greater', since l>l'
394 long double G_n_l_K_lg(int n, int l, double K)
395 {
396   long double h1 = G_n_n_1_K_n_2(n,K);
397   long double h2 = (2*n-1)*(4+(n-1)*(1+n*n*K*K))*h1;//= G_n_n_2_K_n_3(n,K);
398
399   int i;
400   // i = 3 as this is the base case (i.e. G(n,n-3,K,n-4)) for l>l' (l'=l_p)
401   for(i=3;i<=n-l;i++)
402   {
403     // l should be given in terms of n (for the argument of A_g and B_g) -
          as the first step
404     // is to calculate h3 = G(n,n-3,K,n-4), A = calc_A_g(n,n-3+1,K),
405     // B = calc_B_g(n,n-3+1,K), we must add 1, as A and B use l rather
          than l-1
406     long double A = calc_A_g(n,n-i+1,K);
407     long double B = calc_B_g(n,n-i+1,K);
408
409     // if i is odd redefine h1, else redefine h2
410     if(i%2 != 0)
411     {
412       h1 = A*h2+B*h1;
413     }
414     else
415     {
416       h2 = A*h1+B*h2;
```

```
417     }
418
419   }
420
421   // As we want the last calculated value,
422   // simply check to see what 'i' is
423   if(i%2==0)
424     val = h2;
425   else
426     val = h1;
427
428   return val;
429 }
430
431 long double calc_A_g(int n, int l, double K)
432 {
433   // A_g = 4n^2-4l^2+l(2l+1)(1+n^2*K^2)
434   return 4*n*n-4*l*l+l*(2*l+1)*(1+n*n*K*K);
435 }
436
437 long double calc_B_g(int n, int l, double K)
438 {
439   // B_g = -4(n)^2[n^2-(l+1)^2](1+l^2*K^2)
440   return -4*n*n*(n*n-(1+l)*(1+l))*(1+l*l*K*K);
441 }
442
443 long double calc_A_s(int n, int l, double K)
444 {
445   // A_s = 4n^2-4l^2+l(2l-1)(1+n^2*K^2)
446   return 4*n*n-4*l*l+l*(2*l-1)*(1+n*n*K*K);
447 }
448
449 long double calc_B_s(int n, int l, double K)
450 {
451   // B_s = -4(n^2)(n^2-l^2)[1+(l+1)^2*K^2]
452   return -4*n*n*(n*n-l*l)*(1+(l+1)*(l+1)*K*K);
453 }
454
```

```
455 // Calculates G(n,l,K,l') = G(n,l,K,l+1) i.e. 's' is 'smaller',
456 // since l<l'
457 long double G_n_l_K_ls(int n, int l, double K)
458 {
459   long double h1 = G_n_n_1_K_n(n,K);
460   long double h2 =(2*n-1)*(1+n*n*K*K)*n*h1; // = G_n_n_2_K_n_1(n,K);
461
462   int i;
463   // i = 2 as this is the base case (i.e. G(n,n-2,K,n-1) for l<l' (l' =
        l_p))
464   for(i=2;i<n-l;i++)
465   {
466     // l should be given in terms of n (for the argument of A_s and B_s) -
          as the first step
467     // is to calculate h3 = G(n,n-2,K,n-1), A = calc_A_s(n,n-2,K),
468     // B = calc_B_s(n,n-2,K), we must add 2, as A and B use l rather than
          l-2
469     long double A = calc_A_s(n,n-i+1,K);
470     long double B = calc_B_s(n,n-i+1,K);
471
472     if(i%2 == 0)
473     {
474           h1 = A*h2+B*h1;
475     }
476     else
477     {
478           h2 = A*h1+B*h2;
479     }
480   }
481
482   // If l == even, the last h1 calculated will be the answer
483   // else,       the last h2 calculated will be the answer.
484   long double val;
485   // As we want the last calculated value,
486   // simply check to see what 'i' is
487     if(i%2!=0)
488       val = h2;
489     else
```

```
490       val = h1;
491
492   return val;
493 }
494
495 // Calculates G(n,n-1,0,n)
496 long double G_n_n_1_0_n(int n)
497 {
498   long double numerator;
499   long double denominator;
500
501   numerator = sqrtl(PI/2)*8*n*powl(4*n,n)*expl(-2*n);
502   denominator = factorial_minus_lower(2*n-1,1);
503
504   return numerator/denominator;
505 }
506
507 // Calculates G(n,n-1,K,n)
508 long double G_n_n_1_K_n(int n, double K)
509 {
510   if(K==0.0)
511   {
512     return G_n_n_1_0_n(n);
513   }
514
515   long double numerator;
516   long double denominator;
517
518   numerator = expl(2*n-2/K*atanl(n*K));
519   denominator = sqrtl(1-exp(-2*PI/K))*powl(1+n*n*K*K,n+2);
520
521   return numerator/denominator*G_n_n_1_0_n(n);
522 }
523
524 // Calculates G(n,n-2,K,n-1)
525 long double G_n_n_2_K_n_1(int n, double K)
526 {
527   return (2*n-1)*(1+n*n*K*K)*n*G_n_n_1_K_n(n,K);
```

```
528 }
529
530 // Calculates G(n,n-1,K,n-2)
531 long double G_n_n_1_K_n_2(int n, double K)
532 {
533   return (1+n*n*K*K)/(2*n)*G_n_n_1_K_n(n,K);
534 }
535
536 // Calculates G(n,n-2,K,n-3)
537 long double G_n_n_2_K_n_3(int n, double K)
538 {
539   return (2*n-1)*(4+(n-1)*(1+n*n*K*K))*G_n_n_1_K_n_2(n,K);
540 }
541
542 // Calculates the number: n!/lower_limit! - hence if
543 // lower_limit = 1, factorial_minus_lower(n,1) = n!
544 long double factorial_minus_lower(int n, int lower_limit)
545 {
546   long double fact = 1.0;
547
548   int i;
549   for(i=n;i>lower_limit;i--)
550   {
551     fact *= i;
552   }
553
554   return fact;
555 }
556
557 int max(int n1, int n2)
558 {
559   if(n1>n2)
560     return n1;
561   else
562     return n2;
563 }
564
565 int input_accepter()
```

```
566 {
567   char input[256];
568   gets(input);
569   int n = atoi(input);
570 
571   while(!(n+1))
572   {
573         printf("You did not enter a number correctly, please enter
                 again:\n");
574         gets(input);
575         n = atoi(input);
576   }
577   return n;
578 }
579 
580 int open_file(const char* FILENAME, FILE** fp)
581 {
582   int write = -2;
583 
584   printf("Trying to open '%s'\n",FILENAME);
585   if((*fp = fopen(FILENAME,"r")))
586   {
587     //File exists - set for reading
588     write = 0;
589     printf("File is ready for reading\n");
590   }
591   else if((*fp = fopen(FILENAME,"w")))
592   {
593     //File does not exist - set for writing
594     write = 1;
595     printf("File is ready for writing\n");
596   }
597   else
598   {
599     printf("Could not open file for ");
600     if(write)
601     {
602       printf("writing.\n");
```

```
603     }
604     else
605     {
606       printf("reading.\n");
607     }
608     printf("Will calculate function without saving\n");
609     write = -1;
610   }
611
612   return write;
613 }
614
615 // Single threaded alpha_nl calc
616 double calc_alpha_nl(int n, int l, int iter, double y, double h_factor,
      double alpha, double* theta)
617 {
618   double alpha_nl;
619   double sum_I =
        calc_I_y(n,l,l-1,y,iter,h_factor,theta)+calc_I_y(n,l,l+1,y,iter,
620                h_factor,theta);
621
622   // 'c' is speed of light
623   // 'a_0' is the Bohr Radius
624   alpha_nl =
        (2*pow(PI,0.5)*pow(alpha,4)*pow(a_0,2)*c)/3*2*sqrt(y)/(n*n)*sum_I;
625
626   return alpha_nl;
627 }
628
629 double calc_I_y(int n, int l, int l_p, double y, int iter, double
      h_factor, double* theta)
630 {
631   if(!(l_p==-1 || l_p==n))
632   {
633     int l_max            = max(l,l_p);
634     double const const_fact = l_max*y;
635
636     // As l_p is no longer used for any other purpose than indexing
```

```
637     // theta, we assign 1 if larger than l, or 0 if smaller.
638     if(l_p > l)
639           l_p = 1;
640     else
641           l_p = 0;
642     double I_integral    =
          calc_I_integral_y(n,l,l_p,iter,y,h_factor,theta);
643
644     return const_fact*I_integral;
645   }
646   else
647     return 0.0;
648 }
649
650 double calc_I_integral_y(int n, int l, int l_p, int iter, double y,
      double h_factor, double* theta)
651 {
652   int num_iter = iter;
653   long double integral_val = 0;
654   int i;
655   double h_m = h_factor/n;
656   double x1 = 0.0;
657   double x5 = 4*h_m; // x5 = x1+4h
658   int theta_index = 0;
659
660   // theta_index = ((n-1)*(n-2)+2*l-(l_p-1))*num_iter
661   // This is the base offset in the theta_array. num_Iter values of
662   // theta will be needed.
663
664
665   // 4 is the amount of iterations in the five point integration.
666   // '+1' to account for increment of theta_index in outer of loop.
667   theta_index = ((n-1)*(n-2)+2*l+(l_p-1))*(num_iter*4+1);
668
669   for(i=0;i<num_iter;i++)
670   {
671         // Note: x1 = K^2 NOT K
672         integral_val += boole_integral_y(n,x1,x5,h_m,&integrand_y,y,
```

```
673                             theta,theta_index);
674         h_m         = 2*h_m;
675         x1          = x5;
676         x5          = x1+4*h_m;
677
678         theta_index += 4; // Add 4 to offset
679   }
680
681   return integral_val;
682 }
683
684 // boole's integral as given on:
      http://en.wikipedia.org/wiki/Boole%27s_rule
685 double boole_integral_y(int n, double x1, double x5, double h, double
      (*i_f)(int n, double K, double y, double* theta, int theta_index),
      double y, double* theta, int theta_index)
686 {
687   // Int_{x_1}^{x_5}f(x)dx =
        frac{2h}{45}(7f(x_1)+32f(x_1+h)+12f(x_1+2h)+32f(x_4)+7f(x_5))
688   // +error_term, where h is the step size, hence x_1+4h = x_5
689   // Note: Although i_f takes several parameters, only the third
690   // (given by the x1 and h term) is a variable. Hence we can use
691   // Boole's law for functions of 1 variable.
692
693   return (2*h)/(45)*(7*i_f(n,x1,y,theta,theta_index)+32*i_f(n,x1+h,y,
694         theta,theta_index+1)+12*i_f(n,x1+2*h,y,theta,theta_index+2)+
695         32*i_f(n,x1+3*h,y,theta,theta_index+3)+7*i_f(n,x1+4*h,y,theta,
696         theta_index+4));
697 }
698
699 double integrand_y(int n, double K2, double y, double* theta, int
      theta_index)
700 {
701   // NOTE: K2 = K^2
702   return pow(1+n*n*K2,2)*theta[theta_index]*exp(-K2*y);
703 }
```

radiative_recombination_long_double.c

A.4 Iterative Computation

```
/*********************************
 * file: 'b_n_calc_iterative.c' *
 *********************************/

#include <sys/types.h>
#include <time.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "constants.h"
#include "oscillator_strength_gaunt_final.c"
#include "expint.c"

// CASE = 1 for case A, CASE = 2 for case B
const int CASE = 2;

int compute_b_b_coll = 1;

int input_accepter();
void b_n_calculate(int n, int m, int num_iter, double N_e, double T_e);
double b_n_calc_level_n(double b[], int n, int m, double T_n_const,
    double beta, double N_e, int num_of_levels, double T_e, double P_n,
    double P_cn, double sum_S_n_infty, double** S_mn, double T_n);
double sum_of_A_nm(int n);
double p_cn_term(int n, double x_n, double N_e, double T_e);
double spontaneous_radiation(int n, int m);
double x_n_term(double beta, int n);
double collisional_ionization(int n, double T_e, double N_e);
double p_n_term(int n, int m, double T_e, double N_e, double beta);
double sum_of_b_m_S_mn(double b[], double P_n, int n, int num_of_b_n,
    double beta, int infty_int, double T_e, double sum_S_n_infty, double
    N_e, double** S_mn);
double collisional_transition(int n, int m, double beta, double T_e,
    double N_e);
double sum_of_C_nm(int n, int m, double beta, double T_e, double N_e);
```

```
32 double partial_S_mn_sum(double P_n, int n, int num_of_b_n, int
       infty_level, double beta, double T_e, double N_e);
33 double input_accepter_f();
34 double s_mn_term_m_greater(double P_n, int n, int m, double beta, double
       T_e, double N_e);
35 double s_mn_term_n_greater(double P_n, int n, int m, double beta, double
       T_e, double N_e);
36 double p_mn_term_m_greater(int m, int n, double beta, double T_e, double
       N_e);
37 double p_mn_term_n_greater(int m, int n, double beta, double T_e, double
       N_e);
38 void calc_S_mn_term(double P_n, int n, int num_of_b_n, double beta,
       double T_e, double N_e, double** S_n);
39
40 int main()
41 {
42   // Input variables:
43   int n;
44   int num_iter;
45   int m; // number of levels from CASE to continuum
46   double T_e;
47   double N_e;
48
49   // Default values of T_e and N_e if non entered:
50   T_e = 1e4;
51   N_e = 1e4;
52
53     printf("Please enter n (number of b_n coefficients to calulate):\n");
54     n = input_accepter();
55
56     printf("Please enter number of iterations for the iterative
           method:\n");
57     num_iter = input_accepter();
58
59   printf("Please enter number of levels from 1 to continuum:\n");
60   m = input_accepter();
61
62   // Setup timing
```

```
63   time_t t0, t1; /* time_t is defined in <time.h> and <sys/types.h> as
         long */
64   clock_t c0, c1; /* clock_t is defined in <time.h> and <sys/types.h> as
         int */
65
66   long count;
67
68   // Clear Screen
69   system("clear");
70   printf("Calculating b_n ...\n");
71   printf("\n");
72   printf("N_e = %e\n",N_e);
73   printf("n_max = %d\n",n);
74   printf("# of iterations    time_taken\n");
75
76   N_e = 10000.0;
77   int i;
78   for(i=0;i<=100;)
79   {
80     // Start timing
81     t0 = time(NULL);
82     c0 = clock();
83
84     // Perform b_n calculation
85     b_n_calculate(n, m, num_iter, N_e, T_e);
86
87     // Measure elapsed time
88     t1 = time(NULL);
89     c1 = clock();
90     printf("%d                %f\n",num_iter,(float) (c1 -
           c0)/CLOCKS_PER_SEC);
91     num_iter +=50;
92   }
93 }
94
95 int input_accepter()
96 {
97   char input[256];
```

```
98   gets(input);
99   int n = atoi(input);
100
101  while(!n)
102  {
103         printf("You did not enter a number correctly, please enter
                 again:\n");
104         gets(input);
105         n = atoi(input);
106  }
107  return n;
108 }
109
110 double input_accepter_f()
111 {
112  char input[256];
113  gets(input);
114  double number = atof(input);
115
116  scanf("%*c");
117  int result = scanf("%lf", &number);
118
119  while(result != 1)
120  {
121    printf("You did not enter a number correctly, please enter again:\n");
122    gets(input);
123    number = atof(input);
124    result = scanf("%lf", &number);
125  }
126
127  return number;
128 }
129
130
131 void write_output(double b[], int size, int n_lower, double T_e, double
       N_e, int num_iter)
132 {
133  FILE* pFile;
```

```
134  FILE* pFile_b_n;
135  char filename[256];
136  char filename_b_n[256];
137  sprintf(filename,
138    "./b_n_output/b_n_%d_case_%d_T_e_%.0E_N_e_%.0E_num_iter_%d.dat",
139    size,CASE,T_e,N_e,num_iter);
140  sprintf(filename_b_n,
141    "./b_n_output/b_n_%d_case_%d_T_e_%.0E_N_e_%.0E_num_iter_%d.b_n",
142    size,CASE,T_e,N_e,num_iter);
143
144  pFile = fopen(filename, "w");
145  pFile_b_n = fopen(filename_b_n, "w");
146
147  // Specify columns
148  fprintf(pFile, "# n b_n                db/b                beta \n");
149
150  double db_b = 0.0;
151  double beta = 0.0;
152  double v   = 0.0;
153
154  int j;
155  for(j=0; j<=size-CASE; j++)
156  {
157    // (d/dn)lnB(n) = db/b
158    db_b = (b[j+1]-b[j])/b[j];
159
160    v    = R_H*c*pow(Z,2)*(1.0/pow(j,2)-1.0/pow(j+1,2));
161
162    beta = 1-(k*T_e)/(h*v)*db_b; // delta_n not included as this is for
         the alpha transition
163
164    fprintf(pFile,"%d  %.10f                %.10f                %.10f
         \n", j+n_lower, b[j], log10(db_b), beta);
165    fprintf(pFile_b_n,"%.10f\n", b[j]);
166  }
167
168  printf("Saving to file: %s\n", filename);
169
```

```
170   fclose(pFile);
171   fclose(pFile_b_n);
172 }
173
174 void b_n_calculate(int n, int m, int num_iter, double N_e, double T_e)
175 {
176   // Constants
177   double T_n_const = pow(2*PI*m_e*k*T_e, 3.0/2)/(N_e*pow(h,3));
178   double beta    = 1.5789e5/T_e;
179
180   int i, j;
181
182   // NOTE size of b_n = n-1 for CASE B
183   int array_size = n-CASE+1;
184   double b[array_size];
185
186   // Initialise all bn's to 1 for initial iteration
187   for(i=0;i<n-CASE+1;i++)
188   {
189     b[i] = 1.0;
190   }
191
192   // Declare P_n outside to avoid recalculation
193   double P_n[array_size ];
194   double P_cn[array_size];
195   double x_n = 0.0;
196
197   // Holds the sum of S_mn but only from n+1 to infty,
198   // as b_n is = 1.0 at level n+1
199   double sum_S_n_infty[array_size];
200   double** S_mn;
201   double T_n[array_size];
202
203   S_mn = malloc(array_size*sizeof(*S_mn));
204   for(i=0;i<array_size;i++)
205   {
206     S_mn[i] = malloc(array_size*sizeof(*S_mn[i]));
207   }
```

```
208
209   // Initialise all values before entering loop to avoid if-else
210   // statements
211   for(j=CASE; j<=n;j++)
212   {
213     x_n                = x_n_term(beta,j);
214     P_n[j-CASE]        = p_n_term(j,m,T_e,N_e,beta);
215     P_cn[j-CASE]       = p_cn_term(j,x_n,N_e,T_e);
216     T_n[j-CASE]     =
          T_n_const*(P_cn[j-CASE]/(pow(j,2)*P_n[j-CASE]*exp(x_n)));
217     calc_S_mn_term(P_n[j-CASE],j,n,beta,T_e,N_e,S_mn);
218     sum_S_n_infty[j-CASE] =
          partial_S_mn_sum(P_n[j-CASE],j,n,m,beta,T_e,N_e);
219   }
220
221   for(i=1; i<=num_iter; i++)
222   {
223     for(j=CASE; j<=n; j++)
224     {
225       b[j-CASE] = b_n_calc_level_n(b,j, m, T_n_const, beta, N_e, n, T_e,
            P_n[j-CASE], P_cn[j-CASE], sum_S_n_infty[j-CASE], S_mn,
            T_n[j-CASE]);
226     }
227
228   }
229
230   // Deallocate S_mn
231   for (i=0; i<array_size; i++)
232   {
233     free(S_mn[i]);
234   }
235   free(S_mn);
236
237   write_output(b,n,CASE,T_e,N_e,num_iter);
238 }
239
240 void calc_S_mn_term(double P_n, int n, int num_of_b_n, double beta,
      double T_e, double N_e, double** S_mn)
```

```
241 {
242   int i;
243   for(i=CASE;i<n;i++)
244   {
245     S_mn[n-CASE][i-CASE] = s_mn_term_n_greater(P_n, n, i, beta, T_e, N_e);
246   }
247   for(i=n+1;i<=num_of_b_n;i++)
248   {
249     S_mn[n-CASE][i-CASE] = s_mn_term_m_greater(P_n, n, i, beta, T_e, N_e);
250   }
251 }
252
253 double partial_S_mn_sum(double P_n, int n, int num_of_b_n, int
      infty_level, double beta, double T_e, double N_e)
254 {
255   double partial_sum = 0.0;
256
257   int i;
258   for(i=num_of_b_n+1;i<=infty_level;i++)
259   {
260     partial_sum += s_mn_term_m_greater(P_n, n, i, beta, T_e, N_e);
261   }
262
263   return partial_sum;
264 }
265
266 double b_n_calc_level_n(double b[], int n, int m, double T_n_const,
      double beta, double N_e, int num_of_levels, double T_e, double P_n,
      double P_cn, double sum_S_n_infty, double** S_mn, double T_n)
267 {
268   double b_n       = 0.0;
269   double sum_of_terms = 0.0;
270   double x_n    = 0.0;
271
272   x_n           = x_n_term(beta,n);
273   sum_of_terms    = sum_of_b_m_S_mn(b, P_n, n, num_of_levels, beta, m,
        T_e, sum_S_n_infty, N_e, S_mn);
274   b_n       = T_n+sum_of_terms;
```

```
275
276   return b_n;
277 }
278
279 double x_n_term(double beta, int n)
280 {
281   return (beta/n)/n;
282 }
283
284 double sum_of_b_m_S_mn(double b[], double P_n, int n, int num_of_b_n,
      double beta, int infty_int, double T_e, double sum_S_n_infty, double
      N_e, double** S_mn)
285 {
286   double sum = 0.0;
287
288   // Create two loops, one for m>n and one for m<n
289   int i;
290   for(i=CASE;i<n;i++)
291   {
292     double temp = 0.0;
293     temp = b[i-CASE]*S_mn[n-CASE][i-CASE];
294     if(isnan(temp))
295     {
296       temp = 0.0;
297     }
298     sum += temp;
299   }
300   // SKIP level i==n
301   for(i=n+1;i<=num_of_b_n;i++)
302   {
303     double temp = 0.0;
304     temp = b[i-CASE]*S_mn[n-CASE][i-CASE];
305     if(isnan(temp))
306     {
307       temp = 0.0;
308     }
309     sum += temp;
310   }
```

```
311
312   return sum+sum_S_n_infty;
313 }
314
315 double s_mn_term_m_greater(double P_n, int n, int m, double beta, double
        T_e, double N_e)
316 {
317   double P_mn = 0.0;
318   double x_n = x_n_term(beta, n);
319   double x_m = x_n_term(beta, m);
320
321   P_mn = p_mn_term_m_greater(m,n,beta,T_e,N_e);
322
323   return P_mn/P_n*pow(m,2)/pow(n,2)*exp(x_m-x_n);
324 }
325
326
327 double s_mn_term_n_greater(double P_n, int n, int m, double beta, double
        T_e, double N_e)
328 {
329   double P_mn = 0.0;
330   double x_n = x_n_term(beta, n);
331   double x_m = x_n_term(beta, m);
332
333   P_mn = p_mn_term_n_greater(m,n,beta,T_e,N_e);
334
335   return P_mn/P_n*pow(m,2)/pow(n,2)*exp(x_m-x_n);
336 }
337
338 double p_cn_term(int n, double x_n, double N_e, double T_e)
339 {
340   // Collection of terms from free-bound
341   double P_cn = 0.0;
342
343   // Collisional 3-Body Recombination
344   double C_cn = 0.0;
345
346   // Collisional Ionization
```

```
347   double C_nc = 0.0;
348   double N_TE = 0.0; // Population at Thermodynamic Equilibrium
349
350   // Radiative Recombination
351   double alpha_n = 0.0;
352   double S_0 = 0.0;
353
354   S_0  = exp(x_n)*calc_expint(x_n);
355   alpha_n  = 5.197e-14*pow(x_n,3.0/2)*S_0;
356
357   N_TE   = pow(N_e,2)*pow(h,3)*pow(n,2)*exp(x_n)/pow(2*PI*m_e*k*T_e,3.0/2);
358
359   C_nc   = collisional_ionization(n,T_e,N_e);
360
361   // Use detailed balancing relation to compute
362   // C_cn = \frac{h^2}{2\pi*m*k*T}^(3/2)\frac{\omega_n}{2}e^{x_n}C_nc
363
364   // Taken from Shaver 1975,p.8 - unit is: cm^3*s^-1
365   C_cn   = C_nc*N_TE/(N_e*N_e);
366
367   P_cn   = N_e*(alpha_n+C_cn);
368
369   return P_cn;
370 }
371
372 double p_n_term(int n, int m, double T_e, double N_e, double beta)
373 {
374       /*******************************
375        * Transitions OUT of level n *
376        *******************************/
377
378   // n-> down
379   double A_n    = 0.0; // Sum of spontaneous radiation down from level n
380
381   // n-> down AND n-> up
382   double C_nm_sum = 0.0; // Sum of stimulated radiation down AND up from
          level n
383
```

```
384       // n-> continuum
385       double C_nc   = 0.0; // Collisional ionization
386
387       // Sum of all transition probabilities
388       double P_n = 0.0;
389
390   A_n          = sum_of_A_nm(n);
391   C_nc         = collisional_ionization(n,T_e,N_e);
392   C_nm_sum   = sum_of_C_nm(n,m,beta,T_e,N_e);
393   P_n   = A_n + C_nc + C_nm_sum;
394
395   return P_n;
396 }
397
398 double sum_of_A_nm(int n)
399 {
400       /*****************************************
401        * Transitions OUT of level n TO level m *
402        *****************************************/
403
404   double A_nm_sum = 0.0;
405
406   int i;
407   for(i=CASE;i<n;i++)
408   {
409     A_nm_sum += spontaneous_radiation(n,i)*gaunt_approximation(i,n);
410   }
411
412   return A_nm_sum;
413 }
414
415 double sum_of_C_nm(int n, int m, double beta, double T_e, double N_e)
416 {
417   double C_nm_sum = 0.0;
418   int i;
419
420   // Turn if-else into two loops to avoid branching (bad for GPUs)
421   for(i=CASE;i<n;i++)
```

```
422   {
423     C_nm_sum += collisional_transition(i,n,beta,T_e,N_e)*
424                     pow((i+0.0)/n,2)*
425                     exp(x_n_term(beta,i)-x_n_term(beta,n));
426   }
427   for(i=n+1;i<=m;i++)
428   {
429     C_nm_sum += collisional_transition(n,i,beta,T_e,N_e);
430   }
431
432   return C_nm_sum;
433 }
434
435 // (Shaver,1975)
436 double p_mn_term_m_greater(int m, int n, double beta, double T_e, double
      N_e)
437 {
438   // Collection of terms
439   double P_mn = 0.0;
440
441   // Spontaneous Radiation
442   double A_mn = 0.0;
443
444   // Collisional Transition
445   double C_mn = 0.0;
446
447   // We know m>n and hence must use detailed balancing for C_mn and must
        also calculate A_mn
448   A_mn = spontaneous_radiation(m,n)*gaunt_approximation(n,m);
449
450   C_mn = collisional_transition(n,m,beta,T_e,N_e)*pow((n+0.0)/m,2)*
451               exp(x_n_term(beta,m)-x_n_term(beta,n));
452   P_mn = A_mn+C_mn;
453
454   return P_mn;
455 }
456
```

```
457 double p_mn_term_n_greater(int m, int n, double beta, double T_e, double
      N_e)
458 {
459   // Collection of terms
460   double P_mn = 0.0;
461
462   // Collisional Transition
463   double C_mn = 0.0;
464
465   // We know m<n and hence need only consider collisional transition
466   C_mn = collisional_transition(m,n,beta,T_e,N_e);
467   P_mn = C_mn;
468
469   return P_mn;
470 }
471
472 double spontaneous_radiation(int n, int m)
473 {
474   return 1.574e10*pow(n,-5)*pow(m,-3)/(pow(m,-2)-pow(n,-2));
475 }
476
477 double collisional_ionization(int n, double T_e, double N_e)
478 {
479   // Class II cross section - (Sejnowski & Hjellming, 1969)
480   double beta = 1.5789e5/T_e;
481   double C_n_i = 7.8*10e-11*sqrt(T_e)*pow(n,3)*exp(-x_n_term(beta,n))*N_e;
482
483   return C_n_i;
484 }
485
486 // Based on (Gee, Percival, Lodge & Richards, 1976) and Gulyaev, S.
487 // m>n
488 // Collisional transitions between bound states
489 double collisional_transition(int n, int m, double beta, double T_e,
      double N_e)
490 {
491   // Class II cross section - (Sejnowski & Hjellming, 1969)
492   double osc_proper = f_kramer_func(n,m)*gaunt_approximation(n,m);
```

```
493   double I_1 = -2.17989724e-11;
494   double I_n = I_1/pow(n,2);
495   double I_m = I_1/pow(m,2);
496   double pow_exp = ((I_m-I_n)/I_1);
497   double pow_fin = 0.0;
498   double x_nm = (I_m-I_n)/(k*T_e);
499
500   // Ensure pow_fin is real i.e. not complex
501   if(pow_exp < 0)
502   {
503     pow_fin = pow(-pow_exp,-1.1856);
504   }
505   else
506   {
507     pow_fin = pow(pow_exp,-1.1856);
508   }
509
510   double alpha_n_m = 1.2e-7*osc_proper*exp(-x_nm)*pow_fin*N_e;
511
512   return alpha_n_m;
513 }
```

b_n_calc_iterative.c

A.5 Matrix Computation

```
1  /*********************************
2   * file: 'b_n_calc_matrix.c' *
3   *********************************/
4
5  #include <CL/cl.h>
6  #include <errno.h>
7  #include <string.h>
8  #include <sys/types.h>
9  #include <sys/resource.h>
10 #include <time.h>
11 #include <unistd.h>
12 #include <stdio.h>
```

```
13 #include <stdlib.h>
14 #include <math.h>
15 #include "constants.h"
16 #include "oscillator_strength_gaunt_final.c"
17 #include "expint.c"
18 #include <meschach/matrix2.h>
19 #include "kernelReader.h"
20
21 //Input parameters:
22
23 const int N0=2; // Case B
24 int N=500;
25 const double T_e=1e4; // in K
26 const double stim_ion_const = 7.889356e9;
27 const double stim_rec_const = 4.134682e-16;
28
29 // Functions
30 double collisional_ionization(double beta, int n, double T_e, double N_e);
31 double spontaneous_radiation(int n, int m);
32 double gaunt_approximation(int n, int m);
33 double collisional_transition(int n, int m, double beta, double T_e,
       double N_e);
34 void calc_b_n(double N_e, double T_e);
35 void write_output(double b[], int size, int n_lower, double T_e, double
       N_e);
36 double stimulated_radiative_ionization(int n, double T_r, double W);
37 double stimulated_radiative_recombination(int n, double T_r, double T_e,
       double W);
38 double nu(int n, int m);
39 void execute_open_cl_kernel(double** C, int N, int N0, double T_e, double
       N_e, double beta, double W, double T_r);
40
41 int input_accepter()
42 {
43         char input[256];
44         gets(input);
45         int n = atoi(input);
46
```

```
47        while(!(n+1))
48        {
49
50                printf("You did not enter a number correctly, please enter
                      again:\n");
51                gets(input);
52                n = atoi(input);
53        }
54        return n;
55 }
56
57 int main()
58 {
59
60   // Increase stack size
61   struct rlimit old_lim;
62   getrlimit(RLIMIT_STACK,&old_lim);
63
64   printf("old_lim_cur = %lld\n",(long long)old_lim.rlim_cur);
65   printf("old_lim_max = %lld\n",(long long)old_lim.rlim_max);
66   long long newLim = 10000*old_lim.rlim_cur;
67   struct rlimit new_lim = {newLim,newLim};
68
69   // Set new limit in BYTES (4 times the original stack size)
70   setrlimit(RLIMIT_STACK,&new_lim);
71   printf("new_lim_max = %lld\n",(long long)new_lim.rlim_max);
72
73   printf("Please enter N:\n");
74   N = input_accepter();
75
76        // Setup timing
77        time_t t0, t1; /* time_t is defined in <time.h> and <sys/types.h>
                 as long */
78        clock_t c0, c1; /* clock_t is defined in <time.h> and
                 <sys/types.h> as int */
79
80        // Clear Screen
81        system("clear");
```

```
82        printf("Num of b_n's     : %d\n", N);
83        //printf("N_e              : %Ef\n", N_e);
84        printf("T_e              : %Ef\n", T_e);
85        printf("\n");
86
87        printf("Calculating b_n ...\n");
88
89        // Start timing
90        t0 = time(NULL);
91        c0 = clock();
92
93        // Perform b_n calculation
94
95   int iter;
96   double N_e = 0.0;
97   for(iter = 1; iter <= 4; iter++)
98   {
99     N_e = pow(10,iter);
100    calc_b_n(N_e,T_e);
101  }
102
103       // Measure elapsed time
104       t1 = time(NULL);
105       c1 = clock();
106
107       printf("Done calculating b_n's.\n");
108       printf("Total time was:\n");
109       printf ("\telapsed wall clock time: %ld\n", (long) (t1 - t0));
110       printf ("\telapsed CPU time:   %f\n", (float) (c1 -
              c0)/CLOCKS_PER_SEC);
111 }
112
113 void calc_b_n(double N_e, double T_e)
114 {
115   // 'Strength' of star - used for stimulated emission/absorption etc.
116   double W = 0.0;
117   double T_r = 30000;
118   double beta=1.5789e5/T_e;
```

```
119 //double b_n[N-1];
120 double b_n[N-N0+1];
121 double N_TE[N]; // Size should be (N-N0+1)+(1) = N <-- +1 must be
122                 // added as the ionisation level is N+1
123
124 // LTE populations N_TE
125 int j;
126 for(j=1;j<=N;j++)
127 {
128   N_TE[j-1] =
        pow(N_e,2)/pow(T_e,3.0/2)*pow(j,2)*4.1396e-16*exp(pow(j,-2)/
129         T_e*1.579e5);
130 }
131
132 double C[N][N]; // C must have size 1 larger than Col and Spon as it
    also includes the level N+1
133
134 int n;
135 int m;
136
137     // Setup timing
138     time_t t0, t1; /* time_t is defined in <time.h> and <sys/types.h>
            as long */
139     clock_t c0, c1; /* clock_t is defined in <time.h> and
            <sys/types.h> as int */
140
141
142     // Start timing
143     t0 = time(NULL);
144     c0 = clock();
145     // Compute array on GPU:
146
147     execute_open_cl_kernel(NULL, N, N0, T_e, N_e, beta, W, T_r);
148
149     // Measure elapsed time
150     t1 = time(NULL);
151     c1 = clock();
152
```

```
153 printf("\n");
154 printf("\n");
155       printf("Done calculating C from:\n");
156       printf(" GPU\n");
157       printf("Total time was:\n");
158       printf ("\telapsed wall clock time: %ld\n", (long) (t1 - t0));
159       printf ("\telapsed CPU time:   %f\n", (float) (c1 -
              c0)/CLOCKS_PER_SEC);
160
161
162       // Start timing
163       t0 = time(NULL);
164       c0 = clock();
165
166       double col_trans = 0.0;
167       double col_trans_db = 0.0;
168       double spon_rad  = 0.0;
169       double ind_rad   = 0.0;
170       double ind_rad_db = 0.0;
171       // Calculate the lower triangular matrix, then use detailed
              balancing
172       // to reflect around the diagonal
173       for(n=N0;n<=N;n++)
174       {
175       for(m=N0;m<n;m++)
176         {
177           // m<n
178           spon_rad    =
                  spontaneous_radiation(n,m)*gaunt_approximation(m,n);
179           col_trans   = collisional_transition(m,n,beta,T_e,N_e);
180           col_trans_db =
                  col_trans*pow((m+0.0)/n,2)*exp(-beta*(1/pow(n,2)-1/
181                       pow(m,2)));
182           ind_rad     = W*spon_rad/(exp(h*nu(n,m)/k/T_r)+1);
183           ind_rad_db  = ind_rad*pow((n+0.0)/m,2);
184           C[n-N0][m-N0] = -(spon_rad+col_trans_db+ind_rad);
185           C[m-N0][n-N0] = -(col_trans+ind_rad_db);
186           C[n-N0][n-N0] = 0.0;
```

```
187           C[m-N0][m-N0] = 0.0;
188         }
189       }
190       // Measure elapsed time
191       t1 = time(NULL);
192       c1 = clock();
193
194  printf("\n");
195  printf("\n");
196       printf("Done calculating C from:\n");
197  printf(" CPU\n");
198       printf("Total time was:\n");
199       printf ("\telapsed wall clock time: %ld\n", (long) (t1 - t0));
200       printf ("\telapsed CPU time:   %f\n", (float) (c1 -
              c0)/CLOCKS_PER_SEC);
201
202  C[N-N0][N-N0] = 0.0; // Set probability to transition from continuum to
203                       // continuum to zero
204
205  // Radiative recombination from continuum to bound states
206  // based on Seaton 1959. I took the first term S_0(xn) --
207  // which is expressed through the exponential integral "expint"
208  // -- and ignored S_1 and S_2. They may be important (but
209  // not dominate) for the lowest levels n=1, 2, 3.
210  // Good idea is just to take alpha_rad for the low levels
211  // calculated as the function of Temperature.
212
213  for(n=N0;n<=N;n++)
214  {
215    // Radiative Recombination
216    double x_n    = (beta/n)/n;
217    double alpha_n = 0.0;
218    double S_0 = 0.0;
219    double stim_ion = 0.0;
220    double stim_rec = 0.0;
221
222    S_0  = exp(x_n)*calc_expint(x_n);
223    alpha_n    = 5.197e-14*pow(x_n,3.0/2)*S_0;
```

```
224    // Here, N-1 is the ionisation level as levels go from 0->(N-N0) when
           indexing
225    // and ionisation level is at index sizeOf(C)
226          C[N-1][n-N0]  = N_e*N_e*alpha_n;
227
228       // Collisional ionization
229    C[n-N0][N-1]  = collisional_ionization(beta,n,T_e,N_e);
230
231       // Collisional 3-body recombination
232       // C_i,n = N_TE(n)*C(n,N+1)
233       // Add Collisional 3-body recombination to radiative recombination
234         C[N-1][n-N0] = C[N-1][n-N0]+C[n-N0][N-1]*N_TE[n-N0+1];
235
236       // N_TE=Ne^2/Te^(3/2)*nn.^2*4.1396e-16.*exp(nn.^-2/Te*1.579e5);
237       // here I used the principle of DB for Coll ioniz and 3-body rec.:
238       // Ne^3*K_3 = Ne*S(n)*N_TE(n), where Ne*S(n) is C(n,N+1) and
239       // Ne^3*K_3=C(N+1,n) is the rate of 3-body recombination on level
              n.
240       // See e.g. van der Mullen, 1990, p.165 (with misprint - Ne is
              omitted
241       // in the right-hand side.
242
243    stim_ion = stimulated_radiative_ionization(n,T_r,W);
244    C[n-N0][N-1] = -(C[n-N0][N-1]+stim_ion);
245
246    stim_rec = stimulated_radiative_recombination(n,T_r,T_e,W);
247    C[N-1][n-N0] = -(C[N-1][n-N0]+N_e*N_e*stim_rec);
248
249  }
250
251     C[N-1][N-1]=0;
252
253  double D[N];
254  int row;
255  int col;
256  for(row=0;row<N;row++)
257  {
258    double tempSum = 0.0;
```

```
259     for(col=0;col<N;col++)
260     {
261       tempSum += C[row][col];
262     }
263     D[row] = -tempSum;
264   }
265
266   double E[N][N];
267   int i;
268   for(i=0;i<N;i++)
269   {
270     for(j=0;j<N;j++)
271     {
272       E[i][j] = 0.0;
273     }
274     // Make the diagonal equal to D[i]
275     E[i][i] = D[i];
276   }
277
278   double CCC[N][N];
279
280       for(i=0;i<N;i++)
281   {
282
283     for(j=0;j<N;j++)
284     {
285       CCC[i][j] = C[i][j]+E[i][j];
286     }
287   }
288
289   VEC *Rec;
290   Rec = v_get(N-1);
291
292   for(i=0;i<N-1;i++)
293   {
294     Rec->ve[i] = -CCC[N-1][i];
295   }
296
```

```
297       //Left=Left';
298   // Matrix of l.h.s. of eqution is transposed matrix C+E
299   MAT *Left_T;
300   Left_T = m_get(N-1,N-1);
301
302   for(row=0;row<N-1;row++)
303   {
304     for(col=0;col<N-1;col++)
305     {
306       Left_T->me[col][row] = CCC[row][col];
307     }
308   }
309
310   // Use Mechachs
311   // http://www.math.uiowa.edu/~dstewart/meschach/html_manual/
312   // tutorial.html
313   VEC    *N_n;
314   MAT *LU;
315   PERM *pivot;
316   N_n = v_get(N-1);
317   LU = m_get(Left_T->m,Left_T->n);
318   LU = m_copy(Left_T,LU);
319   pivot = px_get(Left_T->m);
320   LUfactor(LU,pivot);
321
322   N_n = LUsolve(LU,pivot,Rec,VNULL);
323
324   for(i=0;i<N-1;i++)
325   {
326        b_n[i] = N_n->ve[i]/N_TE[i+1];
327   }
328
329   write_output(b_n,N-N0,N0,T_e,N_e);
330 }
331
332 double collisional_transition(int n, int m, double beta, double T_e,
      double N_e)
333 {
```

```
334     double s     = m-n;
335     double power   = 1+s+s;
336     double en2     = n*n;
337     double ennp    = n*m;
338     double beta1   = 1.4*sqrt(ennp);
339     double betrt   = beta1/beta;
340     double betsum  = beta1+beta;
341     double f0    = s/ennp;
342     double f1    = pow(1-0.2*f0,power);
343     double f2    = pow(1-0.3*f0,power);
344     double s23trm  = 0.184-0.04/pow(s,(2.0/3));
345     double a     = 8.0/3/s*pow(m/s/n,3)*s23trm*f1;
346     double L     = 0.85/beta;
347         L      = log((1+0.53*L*L*ennp)/(1+0.4*L));
348     double j1    = 4.0/3*a*L*(1/beta-1/betsum);
349     double drt     = sqrt(2-pow((n+0.0)/m,2));
350     double y1    = 1-log(18*s)/4.0/s;
351     double j2    =
            16.0/9*f2*y1*pow(m*(drt+1)/(n+m)/s,3)*exp(-1/betrt)/beta;
352     double xi    = 2.0/pow(n,2)/(drt-1);
353     double z     = 0.75*xi*betsum;
354     double j4    = 2.0/z/(2+z*(1+exp(-z)));
355     double j3    = 0.25*pow(en2*xi/m,3)*j4/betsum*log(1+0.5*beta*xi);
356     double alpha_n_m = N_e*en2*en2*(j1+j2+j3)/sqrt(pow(T_e,3));
357
358   return alpha_n_m;
359 }
360
361 double spontaneous_radiation(int n, int m)
362 {
363   return 1.574e10*pow(n,-5)*pow(m,-3)/(pow(m,-2)-pow(n,-2));
364 }
365
366 double collisional_ionization(double beta, int n, double T_e, double N_e)
367 {
368   double x_n = (beta/n)/n;
369   double S_n = 3.45e-5*pow(n,2)/sqrt(T_e)*exp(-x_n);
370   double C_n_i = N_e*S_n;
```

```
371
372   return C_n_i;
373 }
374
375 void write_output(double b[], int size, int n_lower, double T_e, double
      N_e)
376 {
377         FILE* pFile;
378         FILE* pFile_b_n;
379         char filename[256];
380         char filename_b_n[256];
381         sprintf(filename,
382             "./b_n_output/b_n_%d_T_e_%.0E_N_e_%.0E.dat",size,T_e,N_e);
383         sprintf(filename_b_n,
384             "./b_n_output/b_n_%d_T_e_%.0E_N_e_%.0E.b_n",size,T_e,N_e);
385
386         pFile = fopen(filename, "w");
387         pFile_b_n = fopen(filename_b_n, "w");
388
389         // Specify columns
390         fprintf(pFile, "# n b_n                db/b                beta
              \n");
391
392         double db_b = 0.0;
393         double beta = 0.0;
394         double v   = 0.0;
395
396         int j;
397         for(j=0; j<=size; j++)
398         {
399                 // (d/dn)lnB(n) = db/b
400                 db_b = (b[j+1]-b[j])/b[j];
401
402                 v    = R_H*c*pow(Z,2)*(1.0/pow(j,2)-1.0/pow(j+1,2));
403
404                 beta = 1-(k*T_e)/(h*v)*db_b; // delta_n not included as
405                                              // this is for the alpha
406                                              // transition
```

```
407
408                 fprintf(pFile,"%d  %.10f                %.10f
                        %.10f \n", j+n_lower, b[j], log10(db_b), beta);
409                 fprintf(pFile_b_n,"%.10f\n", b[j]);
410         }
411         printf("size = %d\n", size);
412
413         printf("Saving to file: %s\n", filename);
414
415         fclose(pFile);
416         fclose(pFile_b_n);
417 }
418
419 double stimulated_radiative_ionization(int n, double T_r, double W)
420 {
421         double I_n        = (2.179e-11/n)/n;
422         double LB_n       = (I_n/k)/T_r;
423   //printf("LB_n = %e\n",LB_n);
424   double int1   = calc_ff1(LB_n);
425
426   double stim_rad_ion = stim_ion_const*W/pow(n,5)*int1;
427
428   return stim_rad_ion;
429 }
430
431 double stimulated_radiative_recombination(int n, double T_r, double T_e,
      double W)
432 {
433         double I_n      = (2.179e-11/n)/n;
434         double LB_n     = I_n/k/T_r;
435   double int2   = calc_ff2(LB_n,T_r,T_e);
436
437   double stim_rad_rec =
        stim_rec_const/pow(T_e,3.0/2)*stim_ion_const*W/pow(n,3)*int2;
438
439   return stim_rad_rec;
440 }
441
```

```
442 double nu(int n, int m)
443 {
444   // m<n
445   // Take negative to make positive
446   return -3.29e15*(1/pow(n,2)-1/pow(m,2));
447 }
448
449 void checkErr(int err, int funcNumber)
450 {
451   if(err != CL_SUCCESS)
452   {
453     printf("Err: '%d' from func_num: %d\n",err,funcNumber);
454   }
455 }
456
457 void initmat(int Mdim, int Ndim, int Pdim, float* A, float* B, float* C)
458 {
459         int numEntries = Mdim*Ndim;
460         printf("numEntries = %d\n", numEntries);
461         int i;
462         // populate array
463         for(i = 0; i<numEntries; i++)
464         {
465                 A[i] = i;
466                 B[i] = i;
467                 C[i] = 0;
468         }
469 }
470
471 void print_matrix(double* MAT, int Mdim, int Ndim, char name)
472 {
473         int i, j;
474         int k = 0;
475         printf("%c=",name);
476         puts("[");
477         for(i=0; i<Mdim; i++)
478         {
479                 for(j=0; j<Ndim; j++)
```

```
480                 {
481                         //printf("%i+%i: ", i, j);
482                         printf("%e, ", MAT[k]);
483                         k++;
484                 }
485                 puts("\n");
486         }
487         puts("]");
488 }
489
490 void index_array_calc(int index_array[], int N)
491 {
492         int h,j,k,n,m;
493         h = 0;
494         n = 0;
495         int size = N*(N-1);
496         printf("size = %d\n",size);
497
498         for(j=0;j<N-1;N--)
499         {
500                 m = n+1;
501                 for(k=0;k<N-1;k++)
502                 {
503                         index_array[h] = m;
504                         h++;
505                         index_array[h] = n;
506                         h++;
507                         m++;
508                 }
509                 n++;
510         }
511 }
512
513 void execute_open_cl_kernel(double** C_t, int N, int N0, double T_e,
      double N_e, double beta, double W, double T_r)
514 {
515         // DATA INIT
516   int DIM = N-1;
```

```
517 //int DIM = N;
518       int err;
519       size_t global[1];
520       cl_device_id device_id[100];
521       cl_context context;
522       cl_command_queue commands;
523       cl_program program;
524       cl_kernel kernel;
525       cl_uint nd;
526       cl_mem index_array_in, c_out;
527 int index_array_size = DIM*(DIM-1);
528 int szC = DIM*DIM;
529 int* index_array = (int *)malloc(index_array_size*sizeof(int));
530       double* C = (double *)malloc(szC*sizeof(double));
531
532       // PLATFORM SETUP
533       cl_platform_id platforms[100];
534       cl_uint platforms_n = 0;
535       cl_uint devices_n = 0;
536       clGetPlatformIDs(100, platforms, &platforms_n);
537       if(platforms_n == 0)
538             puts("no devices found");
539       err = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_GPU, 100,
              device_id, &devices_n);
540
541       //context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
542       context = clCreateContext(NULL, 1, device_id, NULL, NULL, &err);
543       commands = clCreateCommandQueue(context, device_id[0], 0, &err);
544
545       // SETUP buffers and writes "index_array" matrix to the device
              memory
546 index_array_calc(index_array,DIM);
547
548 index_array_in = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(int) *
      index_array_size, NULL, NULL);
549       c_out = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(double)
              * szC, NULL, NULL);
```

```
550   err  = clEnqueueWriteBuffer(commands, index_array_in, CL_TRUE, 0,
          sizeof(int) * index_array_size, index_array, 0, NULL, NULL);
551
552       // BUILD the program, define the kernel and setup arguments
553       Program_kernel* pgmKernel = loadKernel("kernel.cl");
554       const char** program_source = (const char**)pgmKernel->kernel;
555       int pgmSize = pgmKernel->size;
556       program = clCreateProgramWithSource(context,
              pgmSize/sizeof(*program_source), program_source, NULL, &err);
557       err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
558
559   // SETUP Kernel
560       kernel = clCreateKernel(program, "matrix_population", &err);
561       err  = clSetKernelArg(kernel, 0, sizeof(int) , &DIM);
562       err |= clSetKernelArg(kernel, 1, sizeof(int) , &N0);
563       err |= clSetKernelArg(kernel, 2, sizeof(double), &T_e);
564       err |= clSetKernelArg(kernel, 3, sizeof(double), &beta);
565       err |= clSetKernelArg(kernel, 4, sizeof(double), &N_e);
566       err |= clSetKernelArg(kernel, 5, sizeof(double), &W);
567       err |= clSetKernelArg(kernel, 6, sizeof(double), &T_r);
568       err |= clSetKernelArg(kernel, 7, sizeof(double), &h);
569       err |= clSetKernelArg(kernel, 8, sizeof(double), &k);
570       err |= clSetKernelArg(kernel, 9, sizeof(cl_mem), &c_out);
571       err |= clSetKernelArg(kernel, 10, sizeof(cl_mem),
572                 &index_array_in);
573
574       // RUN the kernel and collect results
575       global[0] = (size_t)index_array_size/2;
576       nd = 1;
577       err = clEnqueueNDRangeKernel(commands, kernel, nd, NULL, global,
              NULL, 0, NULL, NULL);
578       clFinish(commands);
579       err = clEnqueueReadBuffer(commands, c_out, CL_TRUE, 0,
              sizeof(double) * szC, C, 0, NULL, NULL);
580 }
```

b_n_calc_matrix.c

Appendix B

Test Machine Specification

All tests were carried out on AUT University's test machine, *Pohutukawa*. The specifications are as follows:

CPU	
Manufacturer:	Intel
Model:	X5660
Clock-speed:	2.8 GHz
# of cores:	6
L3 cache:	12 MB
Instruction set:	64-bit

GPU	
Manufacturer:	Nvidia
Model:	Tesla C2050/C2070
Clock-speed:	1.15 GHz
# of CUDA cores:	448
Memory:	6 GB GDDR5
Memory bandwidth:	144 GB/s
Single precision floating point performance:	1.03 TFlops
Double precision floating point performance:	515 GFlops
Memory interface:	384-bit

RAM	
Manufacturer:	Hyundai
Memory type:	DDR3 SDRAM
Bus type:	PC-10600
Data transfer rate:	1333 MHz
Memory clock:	166 MHz
CAS:	CL9
Error correction:	ECC
Size:	3×8192 MB

HDD	
Manufacturer:	Western Digital
Model number:	WDC WD1003FBYX-01Y7B0
Interface type:	SATA-II (3Gb/s)
Data transfer rate (measured):	133 MB/s
Cache:	32 MB
RPM:	7200
Capacity:	1 TB