Spectral Differentiation of Hydrogen in Weak Magnetic Fields

Cardell, Lux[1]

[1]CS Program, DigiPen Institute of Technology, Redmond, WA, 98052.

December 8, 2019

Abstract

This paper details the spectral analysis of an atom in a magnetic field, with the goal of writing a Linux program in C++ to solve the equations. The calculations are based on the methods outlined in the paper "Pseudospectral methods for atoms in strong magnetic fields" [2].

1 Introduction

A particle's binding energy is the amount of energy needed to free that particle from its bound system. In the presence of a magnetic field, the binding energy changes as described by the Zeeman effect [1]. The Zeeman effect describes the behavior of the perturbation of the fine structure of an atom in a magnetic field. In order to find the binding energy, we must operate on the particle's wave function, with an operator known as the Hamiltonian. The Hamiltonian gives us the binding energy of a particle based on its wave function by solving for the kinetic and potential energies.

The physical scenario we are modeling is a simple single-electron atom in a constant magnetic field. Our model simplifies the problem to a 2D slice on the range $[0, \infty)$ based on the atom's natural symmetries. The algorithm implemented in the appendix uses spherical coordinates, and as such isn't

well suited to handling magnetic fields stronger than $\beta = 10$. To accurately calculate a numerical solution to field strengths stronger than $\beta = 10$, the calculations should be performed in cylindrical coordinates [2].

2 Derivations

The derivation of the modified Hamiltonian used for the calculation begins with the time-independent Schrödinger equation:

$$\hat{H}\psi = E\psi \tag{2.1}$$

The Hamiltonian, $\hat{H}$, gives the total energy of the particle, which is reflected in the eigenvalue E.

$$\hat{H} = \frac{\mathrm{p}^2}{2m_e} - \mathrm{V} \tag{2.2}$$

In this equation $\frac{\mathrm{p}^2}{2m_e}$ represents the kinetic energy of the particle, while V represents its potential energy. We'll start with the kinetic term, defining $\vec{\mathrm{p}}$ as the momentum operator.

$$\vec{\mathrm{p}} = -i\hbar\vec{\nabla} \tag{2.3}$$

The grad operator $\vec{\nabla}$ and its square, the Laplace operator, are defined as:

$$\vec{\nabla} = \big(\frac{\partial}{\partial x}\hat{\mathbf{i}} + \frac{\partial}{\partial y}\hat{\mathbf{j}} + \frac{\partial}{\partial z}\hat{\mathbf{k}}\big), \vec{\nabla}^2 = \big(\frac{\partial^2}{\partial x^2}\hat{\mathbf{i}} + \frac{\partial^2}{\partial y^2}\hat{\mathbf{j}} + \frac{\partial^2}{\partial z^2}\hat{\mathbf{k}}\big)$$

The Laplace operator is used to describe how the wave's surface changes through space. This is a useful tool to "visualize" the wave as a 3D object. The waves we use to describe particles are best represented in spherical coordinates. As such, we will define ψ as a function of r, θ, and ϕ, which allows us to separate out the radial variable:

$$\psi(r, \theta, \phi) = \mathrm{R}_{nl}(r)\mathrm{Y}_{lm}(\theta, \phi) \tag{2.4}$$

Our initial potential function $\mathrm{V}(r)$ is defined:

$$\mathrm{V}(r) = -\frac{\mathrm{k} \cdot \mathrm{q}_1 \cdot \mathrm{q}_2}{r} \tag{2.5}$$
$$\mathrm{q}_1 = -\mathrm{q}_2 = e,$$

where e is the charge of the electron and proton. The constant k represents the capacitance of the electric field formed between the electron and the proton, and

$$\mathrm{k} = \frac{1}{4\pi\epsilon_0},$$

where ϵ_0 is defined as the permittivity of free space.

$$\mathrm{V}(r) = \frac{e^2}{4\pi\epsilon_0}\frac{1}{r} \tag{2.6}$$

Rewriting the Hamiltonian, we get:

$$\hat{H} = \frac{-\hbar^2}{2m_e}\nabla^2 - \frac{e^2}{4\pi\epsilon_0}\frac{1}{r} \tag{2.7}$$

Expanding the Laplacian in spherical coordinates gives us:

$$\left[\frac{-\hbar^2}{2m_e}\Big[\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2}{\partial\phi^2}\Big]\right.$$

$$\left. - \frac{e^2}{4\pi\epsilon_0}\frac{1}{r}\right]\mathrm{R}_{nl}(r)\mathrm{Y}_{lm}(\theta,\phi) = E\mathrm{R}_{nl}(r)\mathrm{Y}_{lm}(\theta,\phi) \tag{2.8}$$

$$E\mathrm{R}_{nl}(r)\mathrm{Y}_{lm}(\theta,\phi) \Longrightarrow (E_r + E_{\theta\phi})\mathrm{R}_{nl}(r)\mathrm{Y}_{lm}(\theta,\phi)$$

R is a function solely of the radial variable, and depends on the principal and azimuthal quantum numbers, n and l. We use R to isolate the radial term in the divergence operator so that we can expand it:

$$\frac{-\hbar^2}{2m_e}\left[\frac{d^2\mathrm{R}_{nl}(r)}{dr^2} + \frac{2}{r}\frac{d\mathrm{R}_{nl}(r)}{dr}\right] - \frac{e^2}{4\pi\epsilon_0}\frac{1}{r}\mathrm{R}_{nl}(r) = E_r\mathrm{R}_{nl}(r) \tag{2.9}$$

$$\mathrm{R}_{nl}(r) = r\mathrm{U}(r)e^{-\lambda r} \tag{2.10}$$

Hydrogen only has one electron in its 1s orbital, where $l = 0$. Normally, the first r in equation 2.10 would be raised to the power of l+1, but for Hydrogen in particular we can omit it. Now we can express R_{nl} as a summation over an infinite grid. Conveniently, this is the perfect starting point for our problem, since we model the slice of the wave in the range [0, ∞).

$$\mathrm{R}_{nl}(r) = \sum_{k=0}^{\infty}\mathrm{a}_k r^l \tag{2.11}$$

where a_k is a constant given by the wave function. Taking the second derivative, we get an expression for the radial term over an infinite grid, dependent on the quantum number l:

$$\frac{d^2}{dr^2}\sum_{k=0}^{\infty} a_k r^l = \sum_{k=0}^{\infty} a_k (l(l-1)r^{l-2}) \tag{2.12}$$

Next we turn to the angular term, $Y_{lm}(\theta, \phi)$. We can now separate the variables ϕ and θ:

$$Y(\theta, \phi) = \Theta(\theta)\Phi(\phi)$$

This term allows us to expand the second and third terms in the expanded gradient operator in equation 2.8. The second expands to:

$$\frac{1}{\sin\theta}\frac{\partial}{\partial\theta}\big(\sin\theta\frac{\partial Y_{lm}}{\partial\theta}\big) = \cot\frac{\partial Y_{lm}}{\partial\theta} + \frac{\partial^2 Y_{lm}}{\partial\theta^2} \tag{2.13}$$

and the last, which gives us a value for Φ:

$$\frac{d^2\Phi}{d\phi^2} = -m^2\Phi \Rightarrow \Phi(\phi) = e^{im\phi} \tag{2.14}$$

Rewriting equation 2.7, we get:

$$\hat{H} = \frac{-\hbar^2}{2m_e}\vec{\nabla}^2 - \frac{Ze^2}{4\pi\epsilon_0}\frac{1}{r} \tag{2.15}$$

where Z is the number of protons in the nucleus.

Next we have a chance to look back at equation 2.2 and begin to redefine the potential function. Equation 2.6 models the Coulomb interaction between the proton and the electron. In order to mathematically model the effect of a magnetic field, we must first have a model of the field itself. Associated with every magnetic field is a magnetic vector potential, a vector field defined:

$$\vec{B} = B_0\hat{\mathbf{z}} \tag{2.16}$$

The magnetic field can be found by taking the curl of the field's vector potential:

$$\vec{\nabla} \times \vec{A} = \vec{B} \tag{2.17}$$

The magnetic field doesn't remove the Coulombic term, since the interaction between the proton and the electron still occurs, but does change it.

$$\mathrm{F} = -\vec{\nabla}\mathrm{V} \tag{2.18}$$

$$U_F = -W_F = -\int_{\infty}^{r} \mathrm{F} \cdot d\vec{r} = -\frac{e^2}{4\pi\epsilon_0}\frac{1}{r} \tag{2.19}$$

The potential here is the work required to move the particle from the origin in the nucleus to its position in the radial direction. $\vec{\mathrm{A}}$ is the real value of interest in equation 2.17, since it describes the effect of the magnetic field. If we want to extract it, we'll need to invert the curl operator. While inverting the curl operator is an interesting problem in and of itself, it is far outside the purview of this project. We'll hand it off to our mathematician friends and move on with their answer:

$$\begin{aligned} \vec{\mathrm{A}} &= \frac{1}{2}\vec{\mathrm{B}} \times \vec{r} \\ \vec{\mathrm{A}} &= -\frac{1}{2}\mathrm{B}_0 r \sin\theta \hat{\phi} \\ \mathrm{A}_r &= \mathrm{A}_\theta = 0 \\ \mathrm{A}_\phi &= -\frac{1}{2}\mathrm{B} r \sin\theta \end{aligned} \tag{2.20}$$

Note that equation 2.20 is only dependent on $\hat{\phi}$.

We're now ready to start modifying the Hamiltonian directly by remapping the momentum operator to apply the magnetic field:

$$\vec{\mathrm{p}} \mapsto \vec{\mathrm{p}} - e\vec{\mathrm{A}}$$

$$\begin{aligned} \hat{H}' &= \frac{(\vec{p} - e\vec{\mathrm{A}})(\vec{p} - e\vec{\mathrm{A}})}{2m_e} - \frac{e^2}{4\pi\epsilon_0}\frac{1}{r} \\ &= \frac{1}{2m_e}\big[-\hbar^2\nabla^2 + \frac{1}{4}e^2\mathrm{B}^2\sin^2(\theta) + \frac{i}{2}e\mathrm{B}\hbar\frac{\partial}{\partial\phi}\big] - \frac{e^2}{4\pi\epsilon_0}\frac{1}{r} \end{aligned} \tag{2.21}$$

Here we can begin to take advantage of some of the atom's natural symmetry. We know the magnetic field is aligned vertically, and is constant throughout

all space, which means that we can eliminate the azimuthal term entirely. To begin, we redefine the wave:

$$\begin{aligned}\psi(r,\theta,\phi) &= \xi(r,\theta)e^{im\phi} \\ &= \frac{\mathrm{u}(r,\theta)}{r}e^{im\phi}\end{aligned} \tag{2.22}$$

This gives us our Hamiltonian in all its glory:

$$\begin{aligned}\hat{H}' = \frac{1}{2m_e}\Bigg[&- \hbar^2\Big(\frac{1}{r^2}\frac{\partial}{\partial r}\big(r^2\frac{\partial\frac{\mathrm{u}(r,\theta)}{r}e^{im\phi}}{\partial r}\big) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\big(\sin\theta\frac{\partial\frac{\mathrm{u}(r,\theta)}{r}e^{im\phi}}{\partial\theta}\big) \\ &+ \frac{1}{r^2\sin^2\theta}\frac{\partial^2\frac{\mathrm{u}(r,\theta)}{r}e^{im\phi}}{\partial\phi^2}\Big) + \frac{1}{4}e^2\mathrm{B}^2r^2\sin^2\theta\frac{\mathrm{u}(r,\theta)}{r}e^{im\phi} \\ &+ \frac{i}{2}e\mathrm{B}\hbar\frac{\partial\frac{\mathrm{u}(r,\theta)}{r}e^{im\phi}}{\partial\phi}\Bigg] - \frac{e^2}{4\pi\epsilon_0}\frac{1}{r}\frac{\mathrm{u}(r,\theta)}{r}e^{im\phi}\end{aligned} \tag{2.23}$$

There are some obvious simplifications we can make to equation 2.23:

$$\begin{aligned}\Big[\frac{-\hbar^2}{2m_e}\big[\frac{\partial^2}{\partial r^2} + \frac{1}{r^2}\frac{\partial^2}{\partial\theta^2} + \frac{1}{r^2}\cot\theta\frac{\partial}{\partial\theta} + \frac{m^2}{r^2\sin^2\theta}\big] + \frac{1}{8m_e}e^2B^2r^2\sin^2\theta \\ + \frac{e\mathrm{B}\hbar m}{4m_e} - \frac{e^2}{4\pi\epsilon_0}\frac{1}{r}\Big]\mathrm{u}(r,\theta) = E\mathrm{u}(r,\theta).\end{aligned} \tag{2.24}$$

Next we make the substitution $\mu = \cos\theta$:

$$\begin{aligned}\hat{H}' = \Bigg[&\frac{-\hbar^2}{2m_e}\big[\frac{\partial^2}{\partial\tilde{r}^2} + \frac{1-\mu^2}{\tilde{r}^2}\frac{\partial^2}{\partial\mu^2} - 2\frac{\mu}{\tilde{r}^2}\frac{\partial}{\partial\mu} + \frac{m^2}{\tilde{r}^2(1-\mu^2)}\big] \\ &+ \frac{1}{8m_e}e^2\mathrm{B}^2\tilde{r}^2(1-\mu^2) + \frac{e\mathrm{B}\hbar m}{4m_e} - \frac{e^2}{4\pi\epsilon_0}\frac{1}{\tilde{r}}\Bigg]\end{aligned} \tag{2.25}$$

Note the change from r to $\tilde{r}$. Up to this point, r has been in S.I. units, but in order to differentiate we need it to be unitless. To this end, we will put $\tilde{r}$ in units of the Bohr radius, which is the typical radius of an atom:

$$\mathrm{a}_0 = \frac{\hbar}{\alpha m_e c} \tag{2.26}$$

where α is the fine structure constant:

$$\alpha = \frac{e^2}{4\pi\epsilon_0\hbar c}$$

$$\frac{\partial}{\partial r} = \frac{\partial}{\partial \tilde{r}}\frac{\partial \tilde{r}}{\partial r} \Rightarrow \frac{1}{\mathrm{a}_0}\frac{\partial}{\partial r} = \frac{\partial}{\partial \tilde{r}} \tag{2.27}$$

Applying this change and dividing throughout by $\frac{\hbar^2}{2m_e\mathrm{a}_0^2}$, we get:

$$\Bigg[-\Big[\frac{\partial^2}{\partial r^2} + \frac{1-\mu^2}{r^2}\frac{\partial^2}{\partial \mu^2} - 2\frac{\mu}{r^2}\frac{\partial}{\partial \mu} + \frac{m^2}{r^2(1-\mu^2)}\Big] + \frac{1}{8m_e}e^2\mathrm{B}^2r^2\mathrm{a}_0^2(1-\mu^2)\frac{2m_e\mathrm{a}_0^2}{\hbar^2}$$
$$+ \frac{e\mathrm{B}\hbar m}{4m_e}\frac{2m_e\mathrm{a}_0^2}{\hbar^2} - \frac{e^2}{4\pi\epsilon_0}\frac{1}{\mathrm{a}_0 r}\frac{2m_e\mathrm{a}_0^2}{\hbar^2}\Bigg] u(r,\theta) = E\frac{2m_e\mathrm{a}_0^2}{\hbar^2}u(r,\theta) \tag{2.28}$$

Now we can accumulate and define the constants that have filtered out of the equation:

$$\Bigg[-\Big[\frac{\partial^2}{\partial r^2} + \frac{1-\mu^2}{r^2}\frac{\partial^2}{\partial \mu^2} - 2\frac{\mu}{r^2}\frac{\partial}{\partial \mu} + \frac{m^2}{r^2(1-\mu^2)}\Big] + \beta_z^2r^2(1-\mu^2)$$
$$+2\beta_z(m-1) - \frac{2}{r}\Bigg]\mathrm{u}(r,\theta) = \epsilon\mathrm{u}(r,\theta) \tag{2.29}$$

where β_z is the magnetic field strength, $\epsilon = \frac{E}{E_\infty}$, E_∞ is the Rydberg energy, and $\frac{2}{r}$ is the Coulomb potential.

There's one last transformation we need to make before we can apply the differentiation matrix, which is the discretization of our variables. Our variables are currently $\mu \in [-1, 1]$ and $r \in [0, \infty)$. The method of differentiation we use to get a numerical solution, discussed later in §3, only acts on the range [-1, 1], so we need to transform our radial variable appropriately. To this end we set $x = 2\tanh(r) - 1$:

$$\frac{\partial^2}{\partial r^2} = 4\big(1 - \frac{1}{4}(1+x)^2\big)^2\frac{\partial^2}{\partial x^2} + 2\big(1 + x + \frac{1}{4}(1+x)^3\big)\frac{\partial}{\partial x} \tag{2.30}$$

$$\Bigg[-\Big[4\big(1-\frac{1}{4}(1+x)^2\big)^2\frac{\partial^2}{\partial x^2}+2\big(1+x+\frac{1}{4}(1+x)^3\big)\frac{\partial}{\partial x}+\frac{1-\mu^2}{r^2}\frac{\partial^2}{\partial \mu^2}$$
$$-2\frac{\mu}{r^2}\frac{\partial}{\partial \mu}+\frac{m^2}{r^2(1-\mu^2)}\Big]+\beta_z^2 r^2(1-\mu^2)+2\beta_z(m-1)-\frac{2}{r}\Bigg]\mathrm{u}(r,\theta)$$
$$=\epsilon \mathrm{u}(r,\theta) \tag{2.31}$$

3 Spectral Methods

In order to get a numerical value for the binding energies of hydrogen, we need a way to solve the partial differential equation we get from the Hamiltonian. To this end we turn to a class of numerical methods called spectral methods. Spectral methods were developed in the 1970's as a method for the numerical solution of partial differential equations like the ones we discussed in §2. The driving theory behind spectral methods is the representation of a function by a vector of values calculated at a given set of points. By representing the function numerically like this, we can perform matrix operations over the entire domain.

In order to apply spectral methods to our system, we must first apply a conformal mapping to our unbounded wave function. As mentioned in §1, the wave we generate by solving the Schrödinger equation is unbounded in the radial dimension. There are a number of applicable mapping functions to reduce an infinite domain to a finite space, but for the purposes of these calculations we use

$$x = 2\tanh(r) - 1$$

as mentioned in §2. We only need to apply this mapping to the radial dimension. Since the angular dimension is already bounded on the range [0, 2π], we can use the transformation

$$\mu = \cos(\theta),$$

which was substituted in equation 2.25, to map it onto the desired range.

In these calculations, we are using Chebyshev differentiation, which acts on a non-periodic set of points known as Chebyshev points. Chebyshev points are distributed with a nonuniform density over the range [-1, 1] based on a given mesh refinement, *N*. The exact values for Chebyshev points are given by the formula

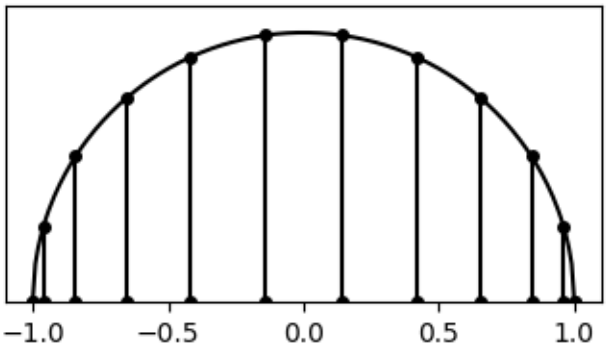

Figure 1: The Chebyshev points for N=11 are shown as the x-axis projections of equispaced points on the unit circle.

$$x_j = \cos(j\pi/N), \qquad j = 0, 1, ..., N$$

Non-periodic grid points help eliminate issues like the Runge phenomenon, which make spectral methods vastly less accurate.

4 Software

The two main parts to the program in the appendix are the solution code (Solve.cpp) which solves the eigenvalue problem for the conditions provided, and the setup of the Chebyshev differentiation matrix and grid points (Cheb.cpp). When called from the command line, the program takes 4 arguments:

- beta: The magnetic field strength in Tesla (0-10)
- M: The mesh refinement for the mu term (approx. 10-11, max. 15)
- N: The mesh refinement for the radial term (max. 30)
- m: The magnetic quantum number for the electron

The program attempts to reduce the number of spurious eigenvalues based on a tolerance, then outputs the remaining values to the command line. The eigenvectors for these values are output to a CSV file named "Eigenvector_[num].csv", where [num] is replaced with an arbitrary number for the index of the eigenvalue in the calculated matrix.

The accompanying Python script is used to visualize the data. On execution, the script prompts the user for a number. This number corresponds to the CSV file storing the eigenvector to visualize.

5 Results and Discussion

The following matrices are the output from calling the Cheb function defined in "Cheb.cpp" in the appendix with values for N ranging from 1 to 5. They have been verified against the output from Trefethen's "cheb.m" [4].

$$\begin{pmatrix} 0.5000 & -0.5000 \\ 0.5000 & -0.5000 \end{pmatrix} \begin{pmatrix} 1.5000 & -2.0000 & 0.5000 \\ 0.5000 & 0 & -0.5000 \\ -0.5000 & 2.0000 & -1.5000 \end{pmatrix}$$

$$\begin{pmatrix} 3.1667 & -4.0000 & 1.3333 & -0.5000 \\ 1.0000 & -0.3333 & -1.0000 & 0.3333 \\ -0.3333 & 1.0000 & 0.3333 & -1.0000 \\ 0.5000 & -1.3333 & 4.0000 & -3.1667 \end{pmatrix}$$

$$\begin{pmatrix} 5.5000 & -6.8284 & 2.0000 & -1.1716 & 0.5000 \\ 1.7071 & -0.7071 & -1.4142 & 0.7071 & -0.2929 \\ -0.5000 & 1.4142 & 0 & -1.4142 & 0.5000 \\ 0.2929 & -0.7071 & 1.4142 & 0.7071 & -1.7071 \\ -0.5000 & 1.1716 & -2.0000 & 6.8284 & -5.5000 \end{pmatrix}$$

$$\begin{pmatrix} 8.5000 & -10.4721 & 2.8944 & -1.5279 & 1.1056 & -0.5000 \\ 2.6180 & -1.1708 & -2.0000 & 0.8944 & -0.6180 & 0.2764 \\ -0.7236 & 2.0000 & -0.1708 & -1.6180 & 0.8944 & -0.3820 \\ 0.3820 & -0.8944 & 1.6180 & 0.1708 & -2.0000 & 0.7236 \\ -0.2764 & 0.6180 & -0.8944 & 2.0000 & 1.1708 & -2.6180 \\ 0.5000 & -1.1056 & 1.5279 & -2.8944 & 10.4721 & -8.5000 \end{pmatrix}$$

The output from the solution has also been verified against the output from the Octave solution for magnetized hydrogen [2], but since the matrices generated by the calculations are far larger than we have space for they have been omitted from this paper.

The calculations implemented are designed specifically for hydrogen, and as such only require analysis of a single electron. If we were working with multiple-electron systems, the program would benefit from multithreading the loops over each electron, but the overhead from parallelization would

only serve as a hindrance here. In addition, a dense matrix is currently used to store the Hamiltonian. Since a new Hamiltonian matrix is required for each particle we calculate, calculations for a parallelized multiple-electron system could be limited by memory usage. This could easily be avoided with the use of a sparse matrix, but for the purposes of these calculations the dense matrix was entirely sufficient.

6 Data

The figures 2 through 6 display the error between the calculated output for various energy states and the data from *Atoms in Strong Magnetic Fields* [3]. The minimum error is marked in each of the figures. The numerical output for the various states is compared with reference [3] in tables 1 through 5. All calculations were done for a mesh refinement of M = 11 and N = 30

Table 1: Comparison of output values for the state $1s_0$

β_Z	here	Ref. [3]	β_Z	here	Ref. [3]
1×10^{-4}	1.00056	1.000200	1×10^{-1}	1.18078	1.180763
1.5×10^{-4}	1.00064	1.000300	1.5×10^{-1}	1.25839	1.258373
2×10^{-4}	1.00072	1.000400	2×10^{-1}	1.32923	1.329211
3×10^{-4}	1.0009	1.000600	3×10^{-1}	1.45495	1.454925
5×10^{-4}	1.00127	1.000999	5×10^{-1}	1.66238	1.662338
7×10^{-4}	1.00164	1.001399	7×10^{-1}	1.83239	1.832332
1×10^{-3}	1.00222	1.001998	1	2.04451	2.044428
1.5×10^{-3}	1.00318	1.002996	1.5	2.32916	2.329066
2×10^{-3}	1.00415	1.003992	2	2.5617	2.561596
3×10^{-3}	1.00611	1.005982	3	2.9366	2.936492
5×10^{-3}	1.01004	1.009950	5	3.49448	3.495594
7×10^{-3}	1.01397	1.013902	7	3.92506	3.922425
1×10^{-2}	1.01985	1.019800	1×10^{1}	4.41984	4.430797
1.5×10^{-2}	1.02958	1.029550	1.5×10^{1}	5.08765	5.084843
2×10^{-2}	1.03923	1.039201	2×10^{1}	5.70019	5.602058
3×10^{-2}	1.05822	1.058207	3×10^{1}	6.07375	6.4103
5×10^{-2}	1.09507	1.095053	5×10^{1}	7.15103	7.5781
7×10^{-2}	1.13041	1.130396	7×10^{1}	7.84973	8.4443

Table 2: Comparison of output values for state $2p_0$

β_Z	here	Ref. [3]	β_Z	here	Ref. [3]
1×10^{-4}	0.250208	0.2501999	1×10^{-1}	0.370403	0.3703681
1.5×10^{-4}	0.250309	0.2502997	1.5×10^{-1}	0.403048	0.4030083
2×10^{-4}	0.250407	0.2503995	2×10^{-1}	0.428568	0.428531
3×10^{-4}	0.250607	0.2505989	3×10^{-1}	0.467367	0.4673569
5×10^{-4}	0.251003	0.250997	5×10^{-1}	0.520706	0.5200132
7×10^{-4}	0.251397	0.2513941	7×10^{-1}	0.555277	0.556267
1×10^{-3}	0.251983	0.251988	1	0.591899	0.5954219
1.5×10^{-3}	0.252982	0.252973	1.5	0.61488	0.6400802
2×10^{-3}	0.253962	0.253952	2	0.677538	0.6713901
3×10^{-3}	0.255903	0.2558921	3	0.585068	0.71432
5×10^{-3}	0.259713	0.2597008	5	0.701634	0.7652975
7×10^{-3}	0.262655	0.2634152	7	0.795126	0.7963735
1×10^{-2}	0.267265	0.2688129	1×10^{1}	1.57171	0.8267545
1.5×10^{-2}	0.277379	0.2773627	1.5×10^{1}	3.00624	0.8577958
2×10^{-2}	0.285405	0.2853874			
3×10^{-2}	0.300053	0.3000325			
5×10^{-2}	0.324846	0.3248202			
7×10^{-2}	0.345267	0.3452365			

Table 3: Comparison of output values for state $2p_{-1}$

β_Z	here	Ref. [3]	β_Z	here	Ref. [3]
1×10^{-4}	-0.250407	0.2503998	1×10^{-1}	-0.50111	0.5010782
1.5×10^{-4}	-0.250607	0.2505995	1.5×10^{-1}	-0.578224	0.578185
2×10^{-4}	-0.250807	0.250799	2×10^{-1}	-0.642757	0.6427096
3×10^{-4}	-0.251206	0.2511978	3×10^{-1}	-0.749318	0.7492475
5×10^{-4}	-0.252002	0.251994	5×10^{-1}	-0.913341	0.9131941
7×10^{-4}	-0.252797	0.2527882	7×10^{-1}	-1.04228	1.042015
1×10^{-3}	-0.253985	0.253976	1	-1.19978	1.199226
1.5×10^{-3}	-0.255956	0.255946	1.5	-1.40845	1.407093
2×10^{-3}	-0.257914	0.2579041	2	-1.57818	1.575651
3×10^{-3}	-0.261795	0.2617843	3	-1.85223	1.846584
5×10^{-3}	-0.269414	0.2694023	5	-2.26415	2.250845
7×10^{-3}	-0.276845	0.2768327	7	-2.58173	2.56045
1×10^{-2}	-0.287649	0.2876352	1×10^{1}	-2.96272	2.93095
1.5×10^{-2}	-0.304784	0.3047682	1.5×10^{1}	-3.45441	3.41052
2×10^{-2}	-0.320913	0.3208951	2×10^{1}	-3.84117	3.79211
3×10^{-2}	-0.350549	0.3505288	3×10^{1}	-4.43488	4.39389
5×10^{-2}	-0.401716	0.4016913	5×10^{1}	-5.23777	5.26948
7×10^{-2}	-0.445252	0.4452239	7×10^{1}	-5.76654	5.924947

Table 4: Comparison of output values for state $3d_{-1}$

β_Z	here	Ref. [3]	β_Z	here	Ref. [3]
1×10^{-4}	-0.111526	0.1115104	1×10^{-1}	-0.263647	0.26357
1.5×10^{-4}	-0.111725	0.1117095	1.5×10^{-1}	-0.2966	0.2964437
2×10^{-4}	-0.111923	0.1119082	2×10^{-1}	-0.322113	0.3218464
3×10^{-4}	-0.112317	0.1123046	3×10^{-1}	-0.360903	0.3604109
5×10^{-4}	-0.113094	0.1130931	5×10^{-1}	-0.413582	0.4131347
7×10^{-4}	-0.11386	0.1138759	7×10^{-1}	-0.449226	0.4499882
1×10^{-3}	-0.115057	0.1150392	1	-0.4854	0.4904806
1.5×10^{-3}	-0.116968	0.1169495	1.5	-0.519291	0.5377152
2×10^{-3}	-0.118844	0.1188244	2	-0.533479	0.5716053
3×10^{-3}	-0.12249	0.1224693	3	-0.52744	0.6192621
5×10^{-3}	-0.129379	0.1293563	5	-0.440522	0.6779122
7×10^{-3}	-0.135769	0.135745			
1×10^{-2}	-0.144533	0.1445071			
1.5×10^{-2}	-0.157353	0.157324			
2×10^{-2}	-0.168457	0.1684264			
3×10^{-2}	-0.187089	0.187055			
5×10^{-2}	-0.215665	0.2156242			
7×10^{-2}	-0.237633	0.2375816			

Table 5: Comparison of output values for state $3d_{-2}$

β_Z	here	Ref. [3]	β_Z	here	Ref. [3]
1×10^{-4}	-0.111725	0.11171	1×10^{-1}	-0.362686	0.3626412
1.5×10^{-4}	-0.112024	0.1120087	1.5×10^{-1}	-0.428009	0.4279525
2×10^{-4}	-0.112322	0.1123068	2×10^{-1}	-0.48204	0.4819653
3×10^{-4}	-0.112917	0.1129014	3×10^{-1}	-0.57068	0.5705063
5×10^{-4}	-0.114101	0.1140841	5×10^{-1}	-0.706959	0.7060961
7×10^{-4}	-0.115275	0.1152582	7×10^{-1}	-0.81495	0.8124766
1×10^{-3}	-0.117021	0.1170033	1	-0.949635	0.9423439
1.5×10^{-3}	-0.119887	0.1198689	1.5	-1.13929	1.114303
2×10^{-3}	-0.122701	0.1226814	2	-1.31357	1.254019
3×10^{-3}	-0.128171	0.1281506	3	-1.36393	1.479163
5×10^{-3}	-0.138517	0.1384944	5	-2.00708	1.816423
7×10^{-3}	-0.148137	0.1481133	7	-1.56472	2.075819
1×10^{-2}	-0.161398	0.1613717			
1.5×10^{-2}	-0.181009	0.1809811			
2×10^{-2}	-0.198279	0.1982491			
3×10^{-2}	-0.227995	0.2279625			
5×10^{-2}	-0.275715	0.275679			
7×10^{-2}	-0.314401	0.3143615			

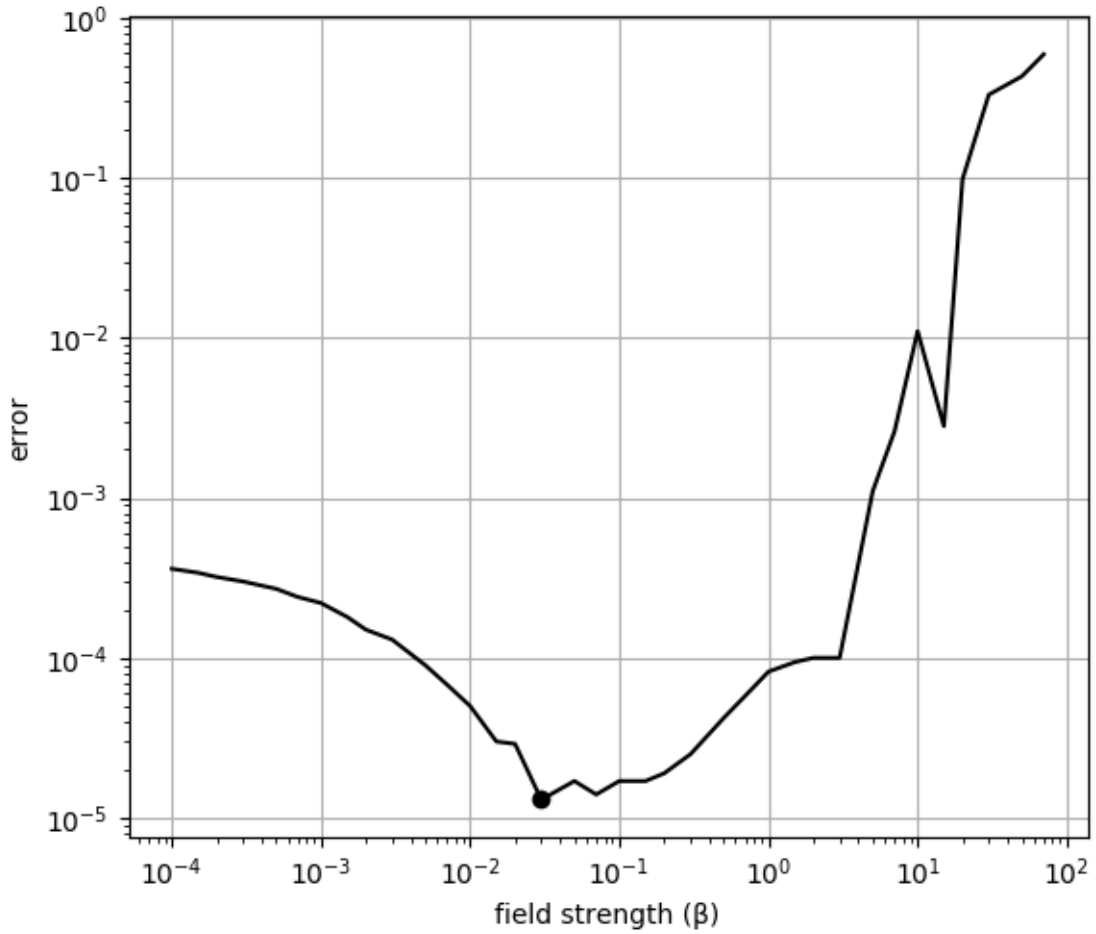

Figure 2: The error between the calculated output for $1s_0$ (table 1) and the values from [3].

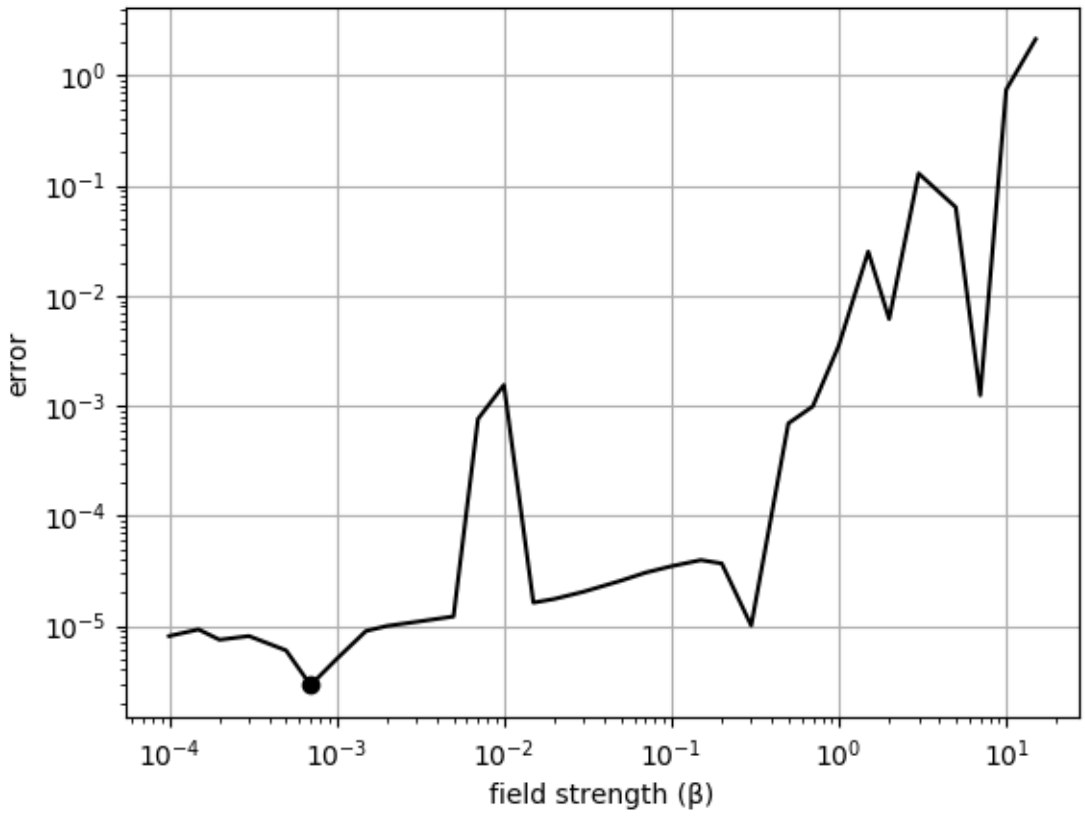

Figure 3: The error between the calculated output for $2p_0$ (table 2) and the values from [3].

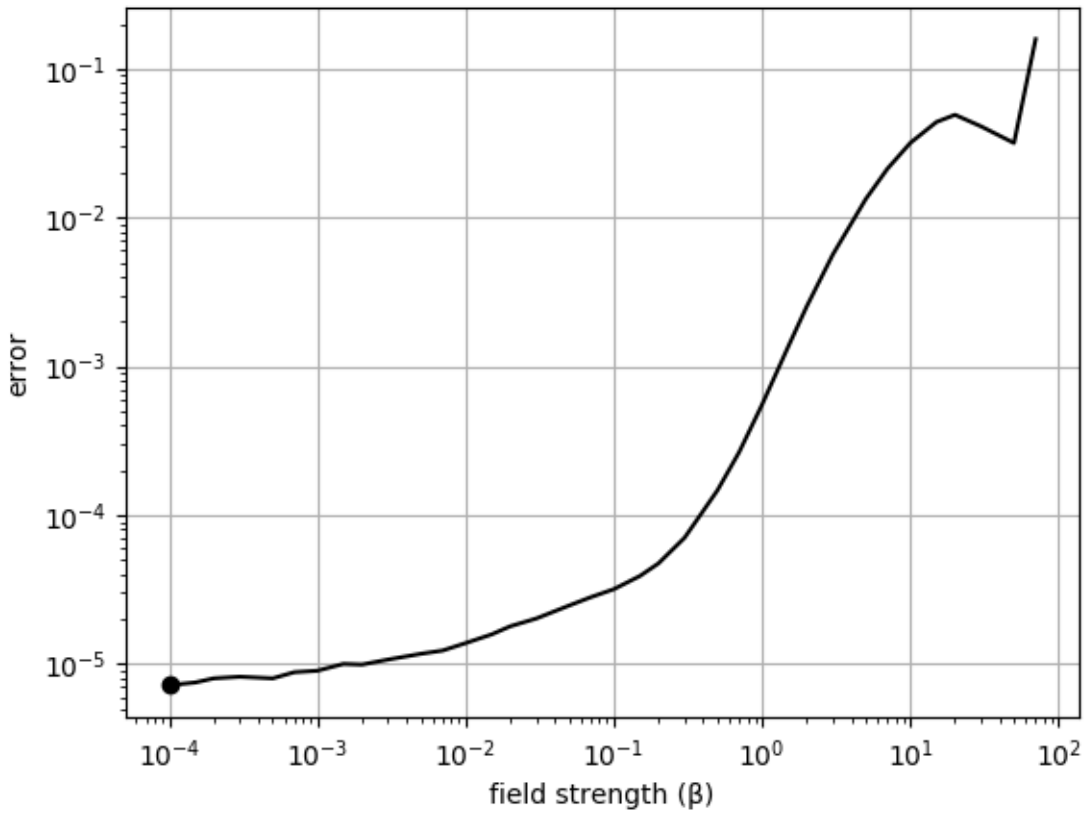

Figure 4: The error between the calculated output for $2p_{-1}$ (table 3) and the values from [3].

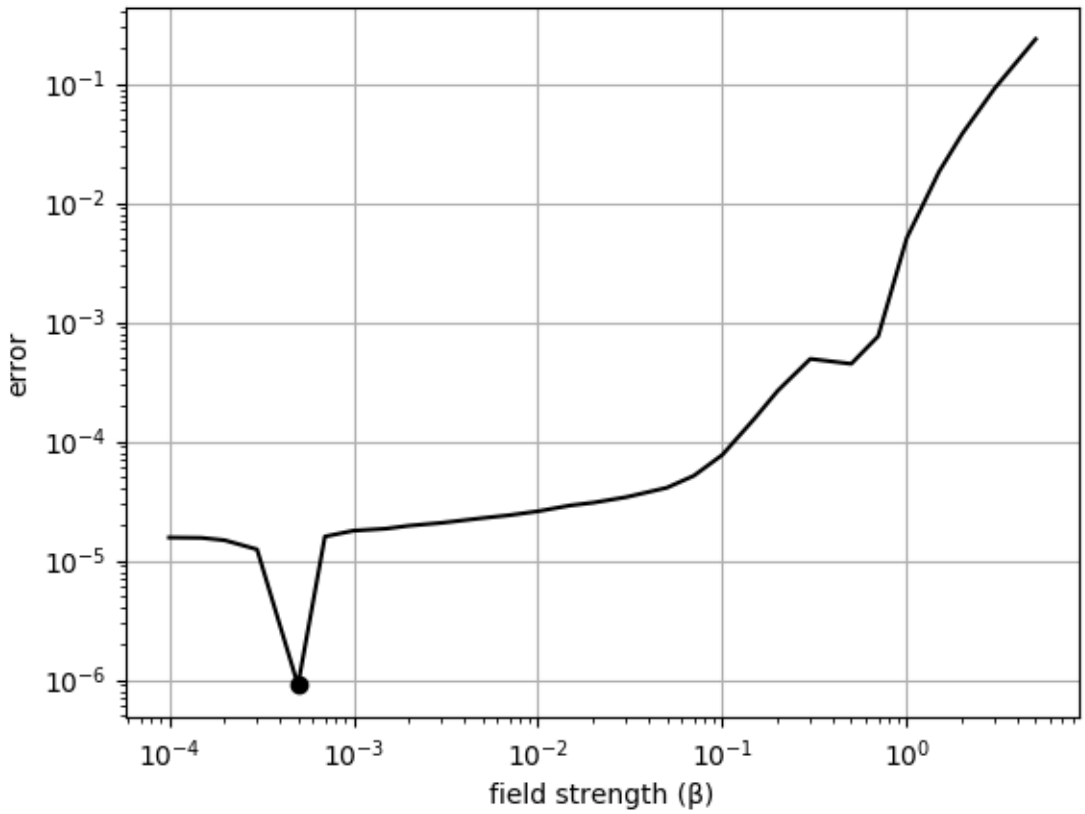

Figure 5: The error between the calculated output for $3d_{-1}$ (table 4) and the values from [3].

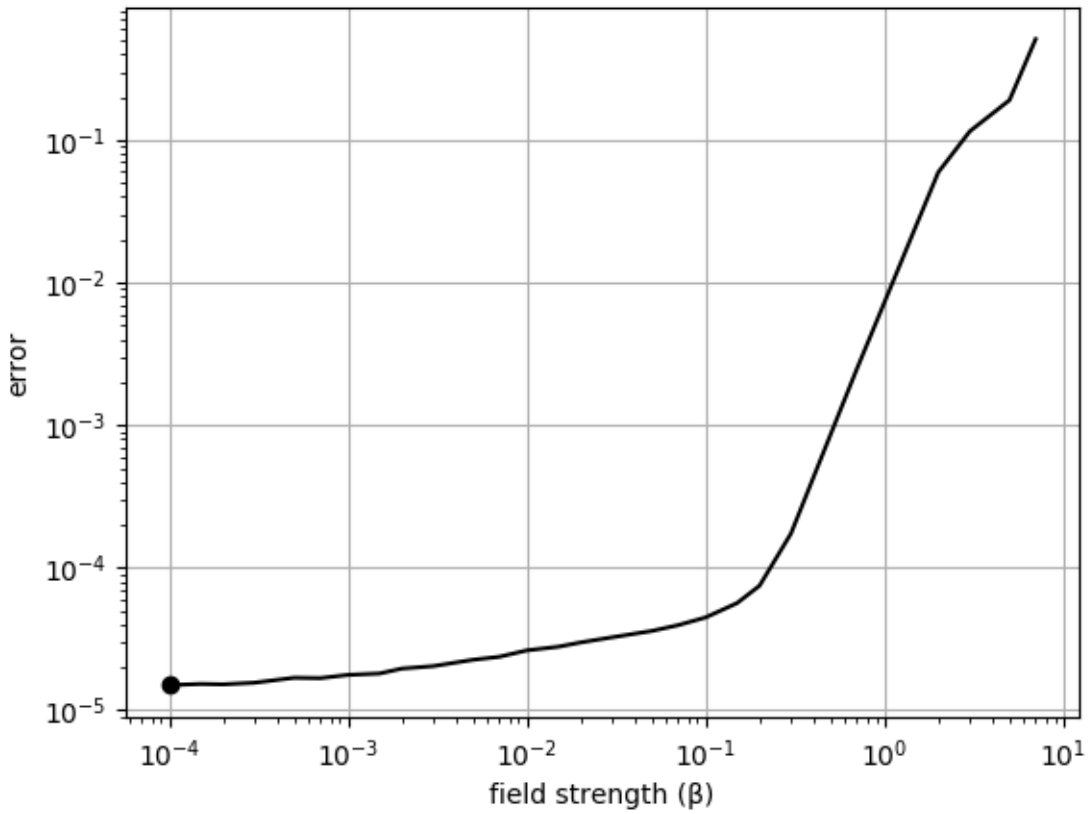

Figure 6: The error between the calculated output for $3d_{-2}$ (table 5) and the values from [3].

References

[1] David J. Griffiths and Darrell F. Schroeter. *Introduction to Quantum Mechanics*. Third. Cambridge University Press, 2018.

[2] Jeremy S. Heyl and Anand Thirumalai. "Pseudo-spectral methods for atoms in strong magnetic fields". In: *MNRAS* 407.1 (Sept. 2010), pp. 590–598. arXiv: `0903.0020 [physics.atom-ph]`.

[3] H. Ruder et al. *Atoms in Strong Magnetic Fields: Quantum Mechanical Treatment and Applications in Astrophysics and Quantum Chaos*. Springer-Verlag, 1994.

[4] L. N. Trefethen. *Spectral Methods in Matlab*. SIAM, 2000, p. 54.

Appendix

Compatibility Note

The following source code was written for C++ 11, and compiled using the GNU Compiler Collection C++ compiler version 7.4.0. This software requires version 9.200.8 of the Armadillo C++ Linear Algebra Library, which can be found at `http://arma.sourceforge.net/`.

Types.h

```
/*! \file     Types.h
**  \author   Lux Cardell
**  \project  CSP 450 - Computational Atomic Structures
**  \brief
**    Type definitions for the program
*/
#ifndef TYPES_H
#define TYPES_H

#include <armadillo> // Linear algebra

//! Floating point precision
typedef double Scalar;

//! Matrix
typedef arma::Mat<Scalar> Matrix;

//! Column vector
typedef arma::Col<Scalar> Column;

//! Row vector
typedef arma::Row<Scalar> Row;

#endif // !TYPES_H
```

Globals.h

```
/*! \file     Globals.h
**  \author   Lux Cardell
**  \project  CSP 450 - Computational Atomic Structures
**  \brief
**    Declarations of global variables used in the
**    program.
*/
#ifndef GLOBALS_H
#define GLOBALS_H

#include <vector>

#include "Types.h"

//! Namespace for global variables
namespace _G
{
    //! Command line arguments
    extern std::vector<std::string> _cl;
}

#endif // !GLOBALS_H
```

Meshgrid.h

This header and its associated source file were written because Armadillo doesn't have support for the meshgrid function in MATLAB and Octave.

```
/*! \file     Meshgrid.h
**  \author   Lux Cardell
**  \project  CSP 450 - Computational Atomic Structures
**  \brief
**    Meshgrid function
*/
#ifndef MESHGRID_H
#define MESHGRID_H
```

```
#include "Types.h"

/*! \brief
**    Creates a pair of matrices from the elements of the
**    two column vectors passed in, similar to the MATLAB
**    function by the same name.
**  \param x
**    The vector used to populate the x-grid
**  \param y
**    The vector used to populate the y-grid
**  \return
**    Returns a tuple containing the x-grid as the first
**    element and the y-grid as the second element.
*/
std::tuple<Matrix, Matrix> Meshgrid(
    Column const& x,
    Column const& y
);

#endif // !MESHGRID_H
```

Meshgrid.cpp

```
/*! \file     Meshgrid.cpp
**  \author   Lux Cardell
**  \project  CSP 450 - Computational Atomic Structures
*/
#include "Meshgrid.h"

/*! \brief
**    Functor used to fill the first return value of the
**    Meshgrid function.
*/
struct ImbueXX
{
    ImbueXX(
```

```
        Column const& _x,
        Column const& _y
    )
        : x(_x), y(_y), elem(0)
    {}

    Scalar operator()()
    {
        int i = elem / y.n_elem;
        ++elem;
        return x[i];
    }

    Column const& x;
    Column const& y;
    int elem;
};

/*! \brief
**    Functor used to fill the second return value of the
**    Meshgrid function.
*/
struct ImbueYY
{
    ImbueYY(Column const& _y)
        : y(_y), elem(0)
    {}

    Scalar operator()()
    {
        int i = elem % y.n_elem;
        ++elem;
        return y[i];
    }

    Column const& y;
    int elem;
};
```

```
std::tuple<Matrix, Matrix> Meshgrid(
    Column const& x,
    Column const& y
)
{
    Matrix xx(y.n_elem, x.n_elem);
    Matrix yy = xx;

    // Assign values to xx
    ImbueXX xImb(x, y);
    xx.imbue(xImb);

    // Assign values to yy
    ImbueYY yImb(y);
    yy.imbue(yImb);

    return std::make_tuple(xx, yy);
}
```

MakePlottable.h

This header is used to pick out the real part of a matrix of complex numbers. This is used to format the data generated before it is read into the Python script used to visualize it.

```
/*! \file     MakePlottable.h
**  \author   Luke Cardell
**  \project  CSP 450 - Computational Atomic Structures
**  \brief
**    Converts a complex matrix to a scalar matrix
*/
#ifndef MAKEPLOTTABLE_H
#define MAKEPLOTTABLE_H

#include "Types.h"
```

```
arma::Mat<Scalar> MakePlottable(
        size_t N,
        arma::Col<std::complex<Scalar>> vec
    )
{
    size_t M = vec.n_elem / N;
arma::Mat<Scalar> result(N, M);

size_t n = 0;

for (size_t i = 0; i < M; ++i)
{
for (size_t j = 0; j < N; ++j)
{
            result(j, i) = vec(n++).real();
        }
}

return result;
}

#endif // !MAKEPLOTTABLE_H
```

Solve.h

This header and its associated source file perform the calculations from the function "hydrogen_s"[2].

```
/*! \file     Solve.h
**  \author   Lux Cardell
**  \project  CSP 450 - Computational Atomic Structures
**  \brief
**    Functions to solve the hydrogenic problem
*/
#ifndef SOLVE_H
#define SOLVE_H
```

```
#include "Types.h"

/*! \brief
**    Implements the spectral calculations
**  \param beta
**    Magnetic field strength
**  \param M
**    Mesh refinement for the mu term
**  \param N
**    Mesh refinement for the radial term
**  \param m
**    Magnetic quantum number of the electron
**  \return
**    Returns 0
*/
int Solve(
    Scalar beta,
    size_t M,
    size_t N,
    int m
);

#endif // !SOLVE_H
```

Solve.cpp

```
/*! \file     Solve.cpp
**  \author   Lux Cardell
**  \project  CSP 450 - Computational Atomic Structures
**  \brief
**    Functions to solve the hydrogenic problem
*/
#include "Solve.h"
#include "Cheb.h"
#include "Meshgrid.h"

//function [V,Lam,w,rs,igood,zoom] = hydrogen_s(beta,M,N,mphi)
```

```
int Solve(
    Scalar beta,
    size_t M,
    size_t N,
    int m
    )
{
    // Chebyshev matrix and points
    // w coordinate, ranging from -1 to 1
    std::tuple<Matrix, Column> cheb = chebM;

    // First derivative differentiation matrix
    Matrix D = std::get<0>(cheb);
    // Chebyshev points for mu
    Column w = std::get<1>(cheb);
    // Second derivative differentiation matrix
    Matrix D2 = D * D;

    if (m != 0)
    {
        w = w.subvec(1, M-1);
        D = D.submat(1, 1, M-1, M-1);
        D2 = D2.submat(1, 1, M-1, M-1);
    }

    Matrix Hw = - arma::diagmat(1 - w % w) * D2
        + arma::diagmat(2 * w) * D;

    // r coordinate, ranging from 1 to -1, rp from 1 to 0
    cheb = chebN;
    // Reusing differentiation matrix memory
    D = std::get<0>(cheb);
    // Chebyshev points for the radial term
    Column r = std::get<1>(cheb);
    Column rp = 0.5 * (r + 1);
    D = 2 * D;
    // Reusing second derivative diff matrix memory
    D2 = D * D;
```

```
Matrix hh = arma::diagmat(1 - rp % rp);
D2 = hh * (hh * D2 + arma::diagmat(-2 * r) * D);
D = hh * D;
D2 = D2.submat(1, 1, N-1, N-1);
D = D.submat(1, 1, N-1, N-1);
rp = rp.subvec(1, N-1);

// zoom factor
Scalar zoom = 1.0 / ((1.0 / 110) + (sqrt(beta) / 41));
Column rs = zoom * arma::atanh(rp);

Matrix R = arma::diagmat(1. / rs);
Matrix R2 = arma::diagmat(1. / (rs % rs));
Matrix Hr = -1. / (zoom * zoom) * D2 - 2*R;

std::tuple<Matrix, Matrix> meshgrid = Meshgrid(rs, w);
Column rr = arma::vectorise(std::get<0>(meshgrid));
Column ww = arma::vectorise(std::get<1>(meshgrid));

Column rperp2 = rr % rr % (1 - ww % ww);


// Hamiltonian operator
Matrix H;

if (m == 0)
{
    H = arma::kron(Hr, arma::eye(M+1, M+1))
        + arma::kron(R2, Hw)
        + arma::diagmat(beta * beta * rperp2);
}
else
{
    H = arma::kron(Hr, arma::eye(M-1, M-1))
        + arma::kron(R2, Hw)
        + arma::diagmat(beta * beta * rperp2)
        + arma::diagmat(m*m / rperp2);
```

```
}

CxMatrix V;
CxColumn Lam;
arma::eig_gen(Lam, V, H);

Lam = Lam + (2 * beta * (m - 1));

std::vector<double> cLam;
std::vector<size_t> igood;

double tolerance = 1.0 / (((M-1)*(N+1)) * ((M-1)*(N+1))) * 0.0001;
CxColumn processed_evs = (V.row(0) % V.row(0)).t();

// remove spurious eigenvalues
for (size_t i = 0; i < Lam.n_elem; ++i)
{
    if (Lam[i].imag() == 0 && Lam[i].real() < 0)
    {
        if (processed_evs[i].real() < tolerance)
        {
            cLam.push_back(Lam[i].real());
            igood.push_back(i);
        }
    }
}

// Process and save the eigenvector
for (size_t i = 0; i < igood.size(); ++i)
{
    CxColumn c = V.row(i).t();
    CxColumn c2 = c % c;
    c2 = c2 / c;

    auto normal = arma::norm(c2);

    c2 = c2 / std::sqrt(normal);
```

```
        std::string filename("Eigenvectors/Eigenvector_");
        filename += std::to_string(igood[i]);
        filename += ".csv";

        Matrix formatted = MakePlottable(N-1, c2);
        formatted.save(filename.c_str(), arma::csv_ascii);
    }

    for (size_t i = 0; i < cLam.size(); ++i)
    {
        std::cout << " Eig#: " << igood[i] << ": "
                  << cLam[i] << std::endl;
    }

    std::sort(cLam.begin(), cLam.end());

    return 0;
}
```

Cheb.h

This header and its associated .cpp file are modeled on the file "cheb.m" published previously by Trefethen [4].

```
/*! \file     Cheb.h
**  \author   Lux Cardell
**  \project  CSP 450 - Computational Atomic Structures
**  \brief
**    Constructs a chebyshev differentiation matrix
**    Based on the file "cheb.m"
*/
#ifndef CHEB_H
#define CHEB_H

#include <tuple>

#include "Types.h"
```

```
extern std::tuple<Matrix, Column> chebM;
extern std::tuple<Matrix, Column> chebN;

/*! \brief
**    Computes a Chebyshev differentiation matrix.
**    Based on "cheb.m"
**  \param N
**    The dimension of the differentiation matrix
**    to construct.
**  \return
**    Returns a tuple containing:
**    + D: the differentiation matrix
**    + x: the Chebyshev grid
**    .
*/
std::tuple<Matrix, Column> Cheb(size_t N);

#endif // !CHEB_H
```

Cheb.cpp

```
/*! \file     Cheb.cpp
**  \author   Lux Cardell
**  \project  CSP 450 - Computational Atomic Structures
*/
#include "Cheb.h"
#include "Types.h"

std::tuple<Matrix, Column> chebM;
std::tuple<Matrix, Column> chebN;

struct ImbueCheb {
    Scalar operator()()
    {
        Scalar r = (M_PI * elem) / N;
        ++elem;
```

```
        return cos(r);
    }

    size_t N = 0;
    int elem = 0;
};

// function [D,x] = cheb(N)
std::tuple<Matrix, Column> Cheb(size_t N)
{
    Matrix D;
    Column x(N+1);

    // if N==0, D=0; x=1; return, end
    if (N == 0)
    {
        D = Matrix{0};
        x = Column{1};
        return std::make_tuple(D, x);
    }

    // x = cos(pi*(0:N)/N)';
    ImbueCheb v;
    v.N = N;
    x.imbue(v); // x is a column vector

    // c = [2; ones(N-1,1); 2].*(-1).^(0:N)';
    Column c(N+1);
    c.ones();
    c[0] = 2;
    c[N] = 2;
    for (size_t i = 1; i < N+1; i += 2)
        c[i] *= -1;

    // X = repmat(x,1,N+1);
    Matrix X = arma::repmat(x, 1, N+1);

    // dX = X-X';
```

```
    Matrix dX = X - X.t();

    // D  = (c*(1./c)')./(dX+(eye(N+1)));   % off-diagonal entries
    Matrix t = c;
    t.transform(
        [](Scalar v) {
            return 1.0 / v;
        }
    );
    t = c * t.t();
    Matrix t2;
    t2.eye(N+1, N+1);
    t2 = dX + t2;
    D = t / t2;

    // D  = D - diag(sum(D'));
    t = arma::diagmat(arma::sum(D.t()));
    D = D - t;

    return std::make_tuple(D, x);
}
```

Main.cpp

```
/*! \file     Main.cpp
**  \author   Lux Cardell
**  \project  CSP 450 - Computational Atomic Structures
**  \brief
**    Solution to the hydrogenic problem.
**    Based on "hydrogen_s.m"
*/

#include <chrono>     // system clock
#include <utility>    // utilities
#include <vector>     // vector
#include <tuple>      // tuple
#include <string>     // string
```

```
#include <iostream>   // cout
#include <limits>     // numeric limits
#include <armadillo>  // Armadillo C++

#define _USE_MATH_DEFINES // M_PI
#include <math.h>     // math functions

#include "Globals.h"  // Globals
#include "Solve.h"    // Calculation code

namespace _G
{
    std::vector<std::string> _cl;
}

/*! \brief
**    Program entry point
**  \param count
**    Number of command line arguments
**  \param args
**    Vector of command line arguments
**  \return
**    Returns 0 on success
*/
int main(
    int count,
    char ** args
)
{
    if (count > 1)
    {
        // Read command line
        for (size_t i = 1;
             i < static_cast<size_t>(count);
             ++i)
        {
        std::string arg(args[i]);
        _G::_cl.push_back(arg);
```

```
    }
}
if (count < 5)
{
    std::cout << "Missing required command line arguments\n";
    std::cout << "Expected arguments are: beta, M, N, m\n";
    return -1;
}

// Start timer
auto tic = std::chrono::system_clock::now();

// Run the calculations
Scalar beta = stof(_G::_cl[0]);
size_t M = stoi(_G::_cl[1]);
size_t N = stoi(_G::_cl[2]);
int m = stoi(_G::_cl[3]);


chebM = Cheb(M);
chebN = Cheb(N);

std::string c_filename = "Cheb/Cheb";
c_filename += std::to_string(M);
c_filename += ".csv";
std::get<1>(chebM).save(c_filename.c_str(), arma::csv_ascii);

c_filename = "Cheb/Cheb";
c_filename += std::to_string(N);
c_filename += ".csv";
std::get<1>(chebN).save(c_filename.c_str(), arma::csv_ascii);

// Solve the problem
Solve(beta, M, N, m);

// End timer
auto toc = std::chrono::system_clock::now();
std::chrono::duration<double> runTime = toc - tic;
```

```
    // Output calculation time
    std::cout << "Calculation finished in "
              << runTime.count() << " seconds.\n";

    return 0;
}
```

visualize.py

```
# file      visualize.py
# author    Lux Cardell
# project   CSP 450 - Computational Atomic Structures
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import csv

data = []
minimums = []
maximums = []

eig_num = input("Eigenvector to visualize: ")

f_name = f'Eigenvectors/Eigenvector_{eig_num}.csv'

with open(f_name) as dataFile:
    csvReader = csv.reader(dataFile)
    for row in csvReader:
        n_row = []

        for e in row:
            n_row.append(float(e))

        data.append(n_row)
        minimums.append(min(n_row))
        maximums.append(max(n_row))
```

```
minimum = min(minimums)
maximum = max(maximums)

# process data

a = []

for row in data:
    n_row = []
    for elem in row:
        norm = (elem - minimum) / (maximum - minimum)
        n_row.append(norm)

    a.append(n_row)

N = len(a)
M = len(a[0])

chebPoints = []

f_name = f'Cheb/Cheb{N+1}.csv'

with open(f_name) as dataFile:
    csvReader = csv.reader(dataFile)
    for row in csvReader:
        for e in row:
            chebPoints.append(float(e) + 1)

fig = plt.figure(figsize=(4,4))
ax = Axes3D(fig)

mg = []
for r in a:
    mg.append(r + r)
    mg[-1].append(r[0])

r = [chebPoints[1:-1] for x in range(2*(M)+1)]
```

```
r = np.transpose(np.array(r))
th = [x * (np.pi / M)  for x in range(2*(M)+1)]
th = np.array([th for x in range(N)])
z = np.array(mg)

plt.subplot(projection="polar")

plt.pcolormesh(th, r, z, cmap=plt.get_cmap("Greys"))

ax = plt.gca()
ax.axes.xaxis.set_ticklabels([])
ax.axes.yaxis.set_ticklabels([])

#plt.legend(labels=[f"M: {M}", f"N: {N}"], loc="upper left")
plt.title(f"M: {M-1}, N: {N+1}")
plt.show()
```

Makefile

```
# author: Lux Cardell
# project: CSP 450 - Computational Atomic Structures
PRG=Solution
OUTDIR=./

GCC=g++
GCCFLAGS=-g -O2 -Wall -Wextra -std=c++11 -pedantic
ARMADILLO=-larmadillo

OBJECTS=$(OUTDIR)Cheb.o $(OUTDIR)Meshgrid.o $(OUTDIR)Solve.o
DRIVER=Main.cpp

gcc: $(OBJECTS)
$(GCC) $(DRIVER) $(OBJECTS) $(GCCFLAGS) $(ARMADILLO) -o $(OUTDIR)$(PRG)

$(OUTDIR)Electron.o: Electron.cpp
$(GCC) -c Electron.cpp $(GCCFLAGS) -pthread
```

```
$(OUTDIR)Solve.o: Solve.cpp
$(GCC) -c Solve.cpp $(GCCFLAGS) -pthread

$(OUTDIR)Cheb.o: Cheb.cpp
$(GCC) -c Cheb.cpp $(GCCFLAGS) -pthread

$(OUTDIR)Meshgrid.o: Meshgrid.cpp
$(GCC) -c Meshgrid.cpp $(GCCFLAGS) -pthread

clean:
rm -f $(OUTDIR)$(PRG) $(OUTDIR)*.o
```