Numerical Evaluation of the Eigenvalues and Eigenfunctions of Spin-Weighted Spheroidal Harmonics via a Relaxation Method

by

Conor Finn

A thesis submitted to University College Dublin
in partial fulfilment of the requirements
of the degree of Master of Science
in Simulation Science

Complex & Adaptive Systems Laboratory
School of Mathematical Sciences
School of Computer Science & Informatics
University College Dublin
National University of Ireland

August 29th, 2013

Project Supervisor:	Professor Adrian Ottewill
Co-director of Programme:	Doctor Edward Cox
Co-director of Programme:	Doctor Neil Hurley

Abstract

This thesis presents an efficient relaxation method for calculating the eigenvalues and eigenfunction of the angular Teukolsky equation or - through a trivial change of variables - the spin-weighted spheroidal differential equation. These solutions can then be used to calculate the full spin-weighted spheroidal harmonics. The aim is to relax from the analytically know spin-weighted spherical harmonics to the spin-weighted spheroidal harmonics.

The spin-weighted spheroidal differential equation has two regular singularities at $x = \pm 1$. The Frobenious method is used to construct a valid infinite series solutions at each of the two regular singular points. These are then combined together to give a representation where the singular behaviour at $x = \pm 1$ has been factored out. The relaxation method is then used in this representation since it being without any singular behaviour it is far more numerically tractable.

Boundary conditions at $x = \pm 1$ were also constructed by again using the Frobenious method and imposing the demand of regularity at these points. The relaxation method can then be applied to solve this two-point boundary value problem in this representation. This relaxation method reduces the problem to that of solving a system of linear equations. However, these equations have a very particular structure that will allow the total computation effort required to solve the system to be greatly reduced by using a special type of matrix solver. An example of one of these matrix solvers is presented as a proof of concept.

This relaxation method is then used to successfully calculate the solutions to the spin-weighted spheroidal differential equation and its special cases. These solutions are then used to calculate the full harmonics by including the solution to the azimuthal equation.

Contents

1 Introduction

1.1 The angular Teukolsky equation

Teukolsky obtained a partial differential equation (PDE) describing the dynamic gravitational, electromagnetic, scalar and neutrino field perturbations of a Kerr black hole [1]. When separating the PDE (in terms of Fourier modes) in Boyer-Linquist coordinates (the $s = 0$ also arises in the separation of the wave equation in flat space). The angular ordinary differential equation from this separation is

$$\left[\frac{1}{\sin\theta}\frac{\mathrm{d}}{\mathrm{d}\theta}\left(\sin\theta\frac{\mathrm{d}}{\mathrm{d}\theta}\right)+c^2\cos^2\theta-2cs\cos\theta-\frac{(m+s\cos\theta)^2}{\sin^2\theta}+s+A\right]S(\theta)=0 \quad (1.1)$$

where $\theta\in[0,\pi]$. This second order ordinary differential equation (ODE) - which shall henceforth be referred to as the *angular Teukolsky equation* - has two regular singular point on the boundaries of this interval, one at the $\theta=0$ pole and the other at $\theta=\pi$ pole: these regular singular points constitute *boundary conditions.* The parameter s is the so called spin-weight (or helicity) of the wave field, which can take on integer and half integer values; the parameter m is the azimuthal separation constant which can take on integer values; and c is a *"spheroidicity"* parameter which can take on complex values, *i.e.* $c\in\mathbb{C}$. It is related to the frequency of the Fourier mode.

For a given choice of the parameter A, a solution $S(\theta)$ which satisfies equation 1.1 and its boundary conditions is called an *eigenfunction* corresponding to A. The constant A is then called the *eigenvalue.* There is no guarantee that a solution/eigenfunction $S(\theta)$ will exist for any arbitrary choice of the parameter A. Indeed, the requirement of regularity at the boundaries restricts the acceptable values of A into a discrete set which will be labelled by discrete parameter l. Specifically, l is an integer such that $|m|\leq l$, which corresponds to the standard spherical harmonic parameter when $c=0$. Furthermore, the values of A that can possibly satisfy equation 1.1 and its boundary conditions depend on the choice of the parameters s, m and c. As such, the eigenvalue and eigenfunction will be labelled to reflect this, *i.e.* ${}_sA_{lmc}$ and ${}_sS_{lmc}(\theta)$ respectively. Once the parameters s, m and c have been fixed, finding the values of ${}_sA_{lmc}$ for which there exists a non-trivial function ${}_sS_{lmc}(\theta)$ that satisfies equation 1.1 is a Sturm-Liouville problem. Specifically, a parametric two-point boundary value problem.

1.2 Spin-weighted spheroidal harmonics

With the change of variable $x=\cos\theta$ the angular Teukolsky equation becomes

$$\left[\frac{\mathrm{d}}{\mathrm{d}x}\left(\left(1-x^2\right)\frac{\mathrm{d}}{\mathrm{d}x}\right)+c^2x^2-2csx-\frac{(m+sx)^2}{1-x^2}+s+{}_sA_{lmc}\right]{}_sS_{lmc}(x)=0 \quad (1.2)$$

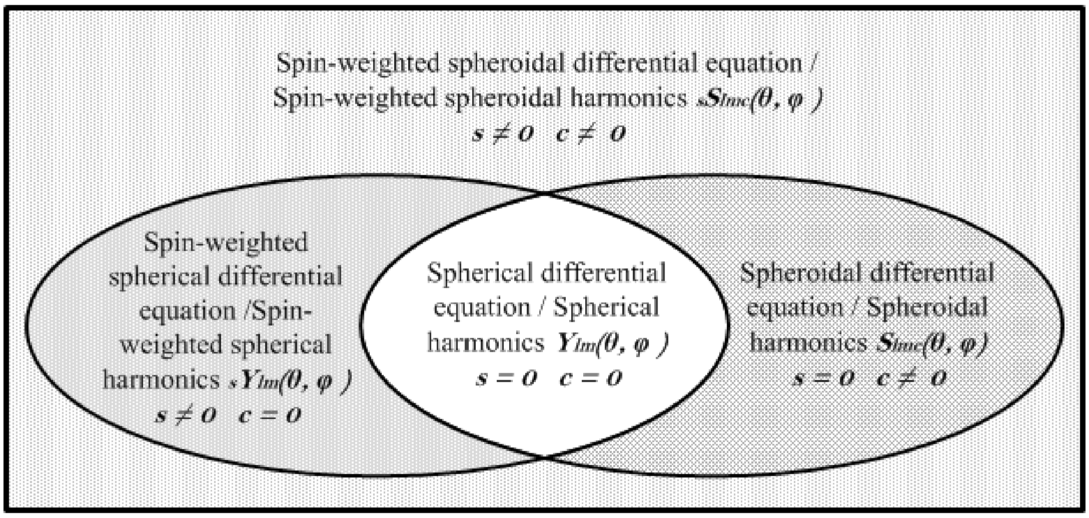

Spin-weighted spheroidal differential equation /
Spin-weighted spheroidal harmonics $_sS_{lmc}(\theta, \varphi)$
$s \neq 0 \quad c \neq 0$
Spin-weighted spherical differential equation /Spin-weighted spherical harmonics $_sY_{lm}(\theta, \varphi)$
$s \neq 0 \quad c = 0$
Spherical differential equation / Spherical harmonics $Y_{lm}(\theta, \varphi)$
$s = 0 \quad c = 0$
Spheroidal differential equation / Spheroidal harmonics $S_{lmc}(\theta, \varphi)$
$s = 0 \quad c \neq 0$

Figure 1.1: The relationship between spin-weighted spheroidal harmonics and its various special cases.

where the interval $\theta = [0, \pi]$ has been mapped onto the interval $x \in [-1, +1]$. Besides reducing the visual complexity, putting the angular Teukolsky equation in this form - which will be referred to as the *spin-weighed spheroidal differential equation* - removes the trigonometric functions which would be computationally costly to evaluate. When $c = 0$ the spin-weighted spheroidal differential equation reduces to the *spin-weighted spherical differential equation*; when $s = 0$ it reduces to the *spheroidal differential equation*; and when both $c = 0$ and $s = 0$ it reduces to the *spherical differential equation* which is perhaps more commonly known as the associated Legendre differential equation: when $m = 0$ this too reduces further to be the Legendre differential equation. When the solutions to the spin-weighted spheroidal; spin-weighted spherical; spheroidal; and spherical differential equations, are multiplied by $\mathrm{e}^{im\phi}$ - coming from the solution of azimuthal Teukolsky equation - the spin-weighted spheroidal harmonics $_sS_{lmc}(\theta, \phi)$; spin-weighted spherical harmonics $_sY_{lm}(\theta, \phi)$; spheroidal harmonics $S_{lmc}(\theta, \phi)$; and spherical harmonics $Y_{lm}(\theta, \phi)$, are obtained respectively (where the variable θ has been momentarily restored for sake of explicitly showing the dependence of harmonics on both the polar angle θ and the azimuthal angle ϕ). The relationship between the equations and their respective harmonics can clearly be seen in Figure 1.1.

It should be noted that throughout this thesis the solutions to the spin-weighted spheroidal, spin-weighted spherical and spheroidal differential equations may be referred to as spin-weighted spheroidal, spin-weighted spherical and spheroidal harmonics respectively - even though they have not been multiplied by $\mathrm{e}^{im\phi}$: this is an slight abuse of terminology and is technically incorrect, however, it is common throughout the literature on these equations. It is accepted since once the solution of the angular ODE has been obtained, it is trivial to obtain the harmonic via multiplication of that solution by $e^{im\phi}$ - the problem of finding the harmonics is essentially the same as the problem of finding the solutions to the angular equations.

1.3 Motivation

The spin-weighted spheroidal harmonics arise in many physical situations, *e.g.* nuclear modelling, signal processing, electromagnetic wave propagation, light scattering, quantum field theory in curved space-time and studies of D-branes in string theory. However, it is its applicability to black hole perturbation theory that is the main interest of this thesis. Hence, why it has been discussed in reference to the Teukolsky equation. With this in mind, the notation and reference to the physical values will be in the context of black hole perturbation theory. For a more detailed development and the Teukolsky equation and the general theory around the topic of black hole perturbation theory see [2].

Despite the fact that the angular Tuekolsky equation was derived in the early seventies, very little analytical or numerical work has been done on the spin-weighted spheroidal harmonics. The most substantial numerical work thus far has done in [3]. There is substantially more analytical (*e.g.* [4], [5], [6], [7] and [8]) and numerical work (*e.g.* [9] and [10]) for the spheroidal harmonics. There is also a substantial amount of analytical work covering spin-weighted spherical harmonics (*e.g.* [11]) although there is not so much numerical work although there are some (*e.g.* [12]).

2 Relaxation method

2.1 Numerical methods for two-point boundary value problems

Two-point boundary value problems are typically considerably more difficult to solve numerically than initial value problems. When dealing with an initial value problem, there is a sufficiently well specified solution on one boundary - the initial values - and numerical integration alone can be used to evolve the solution until the other boundary - the final boundary - is reached. That is, all of the necessary information needed to find the unique solution is contained on the initial boundary, hence one can numerically integrate from that boundary until a value at the final boundary to obtain a unique solution. However, in two-point boundary value problems the solution is not completely specified at either of the boundaries, but rather is partially specified on each of the the two boundaries - there is no sense of an initial or a final boundary. The information required to specify a unique solution is split between the two boundaries and as such numerical integration alone cannot be used to evolve a solution from one boundary to the other: trying to do so will result in non-unique solutions. Rather, some kind of iterative procedure is generally required. This is perhaps not surprising, since information required to specify the unique solution is spread out over two boundaries: there must be communication between the two boundaries in order find the unique solution. It is for this reason that two-point boundary value problems are typically more difficult than equivalent initial value problems: at least they generally require considerably more computational effort.

Shooting methods

The shooting method is perhaps best described as a somewhat naïve method. One boundary is selected to be treated as though it is an initial boundary. Then at this nominated initial boundary, an estimate (or guess) is made of the missing information necessary for a unique solution. Numerical integration is then used to evolve a trial solution to the other, nominated final, boundary. The values this trial solution specifies at this nominated final boundary is then compared to the known value of the partial solution at this boundary. Based on this comparison the estimate of the missing information at the nominated initial boundary is then in some way refined. This refinement is essentially a multidimensional optimisation problem, which is itself a multidimensional root finding problem. This process is iterated until the numerical values of the solution specified at the final boundary matches the known values at this boundary sufficiently well. The somewhat unusual name comes from the analogy between this algorithm and a marksman improving his or her accuracy with each shot when shooting at a target based on feedback from preceding shots. If the initial estimate of the missing information at the nominated initial boundary is a good estimate, the shooting method will converge quickly.

Potential refinements of this method that include multiple shooting methods, where multiple estimates or "shots" are made - not necessarily from just one nominated initial boundary, but rather from multiple points, typically in parallel. Mulitple shooting methods also include shooting to intermediate values. In the above argument one boundary condition was nominated to be treated as though it was an initial point, however, the choice of the boundary point to be nominated is in many ways arbitrary. Hence, one could shoot from both boundaries simultaneously and aim to meet at an intermediate fitting point where some continuity condition must be met. Another refinement could be based on the choice of numerical iteration scheme used in the shooting method. As such, a higher order numerical integration routine or perhaps a method that is better suited to the particular differential equation in hand - *e.g.* a variable step-size numerical integration method, say - would be obvious refinements.

Relaxation methods

As already stated, relaxation methods are generally more complicated then the shooting methods - at least, conceptually if not in practice. In a relaxation method, rather than just trying to improve on an estimate of some missing information at the nominated initial boundary and taking another "shot" iteratively - one starts with an entire trial solution and aims to update and improve all points of the numerical solution simultaneously. That is, it is not just missing information at a nominated initial boundary that is updated, but rather the entire numerical solution at every point is improved with each iteration, including the missing values at the boundaries. For this reason, relaxation will reward a good initial or trial solution with speedy convergence.

To start with the ODE is typically split up into a series of coupled ODEs. These coupled ODEs are then approximated with finite difference equations (FDEs) on a grid covering the range between the boundaries. The trial solution is then introduced - this gives a value for each of the dependant variables in the coupled equations at each point on the grid. These values, this trial solution, will in general not satisfy the FDEs, nor in general will it satisfy the boundary conditions. The relaxation method aims to refine all of the values at each of the grid points and boundaries simultaneously with each iteration. To this end, iteration consists of updating all of the values at each point slightly, in such a way that they all come closer to satisfying the relevant finite difference equations at each point.

An obvious way of improving the relaxation method would be to use higher order finite difference approximations. Another refinement is to assign grid points in a more intelligent manner: the exact choice of the location of the grid points is arbitrary, although it will affect the ability of the method to approximate the solution. As such, they could in some way be assigned dynamically to capture the most important sections of the numerical approximation most accurately.

Collocation method

Collocation methods will not be discussed in detail here, however, they are commonly used to solve two-point boundary value problems and so merit a mention. In collocation methods, a finite dimensional space of functions is chosen - typically the basis functions would be polynomials of some kind. A grid of points on the domain of the problem is defined. The constraint that a potential solution from the finite dimensional space of functions must satisfy the ODE at each of the grid points - the collocation points - and its two boundary conditions is then imposed. Now, provided the dimension of the solution space is equal to the number of grid points (including the boundary points) this will result in a system of linear equations for the coefficients of the solution vector in the solution space. If orthogonal polynomials are used and the collocation points are the roots to the orthogonal polynomials, then the method is known as the orthogonal collocation method.

Relationship between methods

As will be seen in section 2.2, the relaxation method will essentially reduce to the problem of finding the solution to system of linear algebraic equations, which can be represented succinctly as a matrix. It will turn out that this matrix will be of a particular form: an almost block diagonal (ABD) matrix. However, this form is not particular to relaxation methods.

In the relaxation method, a numerical grid over the domain of the problem is defined. Neighbouring grid points will be in some way coupled to each other. If this coupling is done to, say, first order using only the immediate neighbouring points, then it involves coupling pairs of (square) Jacobian matrices together at each point: the coupling of pairs of Jacobian matrices will result in the linear system having the ABD structure. This will be developed in detail in section 2.2.

Now consider the shooting method, in particular, multiple shooting methods. Consider defining a numerical grid on the domain of the problem and perform a multiple shooting method from each of the points to their neighbour on, say, their right hand side. Now, say the numerical integration scheme in the shooting method uses a step size that is equal to the distance between the neighbours. Multiple shooting requires that certain continuity conditions are then met at each end point, hence, effectively the points have coupled together in much the same way as in relaxation method. This will result in an ABD matrix structure. Vice versa, the relaxation method could be interpreted as a multiple shooting method where the shots are over one mesh interval using a one step numerical integration method. The point being, that despite their differences these methods are closely related. This is discussed in greater detail in [13].

Choice of method

As can be seen above, there are several techniques for solving for the eigensystem. Numerical Recipes [14] provides a cavalier, yet amusing, rule of thumb, that if one is unsure of the method to use then *"shoot first, and only then relax"*. However, this thesis presents a relaxation method for solving the parametric two-point boundary value problem that arises when solving the spin-weighted spheroidal differential equation 1.2. This choice of method will now be rationalised.

While individual solutions are of some practical interest, the far more interesting problem - from a mathematical perspective - is that of understanding the behaviour of these individual solutions as the underlying parameters of the spin-weighted spheroidal differential equations are varied. In particular, the dependence of the eigensystem on the *"spheroidicity"* parameter c, which is most generally complex. The main goal of this thesis is to present an efficient solver for the eigensystem as the parameter c is varied along a chosen path of discrete jumps in the complex plane - *i.e.* $c_{\text{new}} = c_{\text{old}} + \Delta c$ - as opposed to merely solving for one individual solution. Provided the discrete jumps Δc are kept reasonably small then the change in the eigensystem should not be substantial. As mentioned above, relaxation will reward a good trial solution with speedy convergence. Hence, relaxation is an ideal method since the old eigensystem for c_{old} provides an ideal trial solution for a relaxation to the new eigensystem for $c_{\text{new}} = c_{\text{old}} + \Delta c$. Furthermore, in the $c = 0$ case - *i.e.* spin-weighted spherical harmonics - the eigensystem can be easily obtained analytically. This provides a natural starting point for any chosen path for c in the complex plane. The ability to integrate around the Riemann surface for the value c is the end goal of this complex solver. This would provide a method of discerning the topology of the Riemann sheet. This knowledge could provide motivation, insight and intuition on the problem of calculating spin-weighted spheroidal harmonics.

2.2 Relaxation method for two-point boundary value problems

The necessary equations for the relaxation method are now developed. The aim of the relaxation method is to solve for N values at each of the M mesh points located at x_k for $k = 0, 1, \dots, M-1$ with x_0 being the left hand boundary point and x_{M-1} being the right hand boundary point: hence, in total there are $M \times N$ unknowns. On the right hand boundary at x_{M-1} there are n_1 equations coming from the directly from the boundary conditions on that point while on the left hand boundary at x_0 there are $n_2 = N - n_1$ equations: hence there are a total of N equations coming from the boundary conditions. In this implementation, only first order finite differences will be used to approximate the solution on the internal mesh points, as such points will be coupled in pairs by the FDEs. Higher order finite differences could be used; this would couple more points together and increase the accuracy of the numerical approximation, however it also increases the complexity of the implementation.

For a grid of size M there are a total of $M-1$ couplings between pairs of points; each coupling provides N differential equations. As such, there are a total of $N \times (M-1)$ equations coming from the finite differencing of the differential equation on the grid. Putting this together with the N equations from the boundary points gives a total of $N \times (M-1) + N = M \times N$ equations for $M \times N$ unknown values. The equations are now developed in detail.

Finite differencing scheme

Consider the set of differential equations, labelled by j, at each point x_k on the interior of the grid, *i.e.* excluding the boundary conditions,

$$\frac{\mathrm{d}y_{j,k}}{\mathrm{d}x} = f_j\left(x_k, y_j\right), \qquad \begin{array}{c} k = 1, 2, \dots, M-2 \\ j = 0, 1, \dots, N-1 \end{array}. \tag{2.1}$$

The one step trapezoidal method is then used to couple pairs of points on the grid

$$y_{j,k} - y_{j,k-1} = \frac{h_k}{2}\left[f_j\left(x_k, y_{j,k}\right) + f_j\left(x_{k-1}, y_{j,k-1}\right)\right], \qquad \begin{array}{c} k = 1, 2, \dots, M-1 \\ j = 0, 1, \dots, N-1 \end{array}, \tag{2.2}$$

where $h_k := x_k - x_{k-1}$. Rearranging yields

$$E_{j,k}\left(y_{j,k}, y_{j,k-1}\right) := y_{j,k} - y_{j,k-1} - \left(x_k - x_{x-1}\right) f_j\left(\frac{x_k - x_{k-1}}{2}, \frac{y_{j,k} - y_{j,k-1}}{2}\right) = 0, \qquad \begin{array}{c} k = 1, 2, \dots, M-1 \\ j = 0, 1, \dots, N-1 \end{array}. \tag{2.3}$$

These are the $(M-1)\,N$ equations coming from the finite differencing process.

Boundary conditions

At the left hand boundary there are $n_2 = N - n_1$ equations

$$E_{j,0}\left(y_{j,0}\right) := \mathrm{BC}_j^{\mathrm{left}}\left(x_0, y_{j,0}\right) = 0, \qquad j = n_2, n_2+1, ..., N-1, \tag{2.4}$$

while on the right hand boundary there are n_1 equations

$$E_{j,M}\left(y_{j,M-1}\right) := \mathrm{BC}_j^{\mathrm{left}}\left(x_{M-1}, y_{j,M-1}\right) = 0, \qquad j = 0, 1, ..., n_2-1. \tag{2.5}$$

Hence, in total there are $n_1 + n_2 = N$ equations coming from the boundaries. Combining these with the finite difference equations yields $N \times (M-1) + N = M \times N$ equations in total, as required. The labelling convention perhaps seems unusual. However, this labelling will produce the ABD matrix structure which is advantageous.

Multidimensional Newton method

These MN equations *should* be satisfied by a solution to the set of coupled ODEs and the boundary conditions. However, as discussed in section 1.1, there is no solution to the boundary value problem *a priori*. Hence, a guess solutions will be used as though it is a solution. The N values of this trial solution at each of the M grid points will in general not satisfy the set of equations that have been developed. However, the multidimensional Newton method will be used to improve the MN trial values in such a way that they should all satisfy the MN equations. The multidimensional Newton method is not the only algorithm that could be utilised for this process but it is the algorithm presented. This updating process will be applied in a gradual iterative manner so the the trial solution improves and converges on to the actual solution. A convergence rate control value will determine the size of the updates that are applied. This is an important feature: if too large of update was applied to the trial solution at each iteration then the algorithm might not converge smoothly or could even fail to converge. A tolerance value will be used to control when the iteration process should halt and present the approximate numerical solution.

Consider the derivatives of the MN equations in each of their MN unknown values. A natural way to present these partial derivatives would be in matrix form. The partial the derivatives of the MN equations in MN unknown values will yield a square matrix of size $MN \times MN$. The aim of this algorithm is to in some way utilise the knowledge of these partial derivatives to improve the solutions of each equation with each iteration.

First consider the multidimensional Taylor expansion of the FDEs 2.3 to first order in their dependent variables;

$$E_{j,k}\left(y_{j,k}+\Delta y_{j,k}, y_{j,k-1}+\Delta y_{j,k-1}\right) \approx E_{j,k}\left(y_{j,k}, y_{j,k-1}\right) \ldots$$
$$\ldots + \sum_{n=0}^{N-1} \frac{\partial E_{j,k}}{\partial y_{n,k}} \Delta y_{j,k} + \sum_{n=0}^{N-1} \frac{\partial E_{j,k}}{\partial y_{n,k-1}} \Delta y_{j,k-1} \qquad \begin{array}{l} k=1,2,\ldots,M-1 \\ j=0,1,\ldots,N-1 \end{array}. \tag{2.6}$$

The aim is to improve the current trial values $y_{j,k}$ with the addition $\Delta y_{j,k}$ so that the first order expansions of the equations are simultaneously satisfied. That is, $E_{j,k}\left(y_{j,k}+\Delta y_{j,k}, y_{j,k-1}+\Delta y_{j,k-1}\right)=0$ for the appropriate choice of the values $\Delta y_{j,k}$ for $k=1,2,\ldots,M-1$ and $j=0,1,\ldots,N-1$: the problem is to obtain the appropriate values $\Delta y_{j,k}$.

Knowing that $E_{j,k}\left(y_{j,k}+\Delta y_{j,k}, y_{j,k-1}+\Delta y_{j,k-1}\right)$ should equal zero, it is required that

$$E_{j,k}\left(y_{j,k}, y_{j,k-1}\right) = -\sum_{n=0}^{N-1} \frac{\partial E_{j,k}}{\partial y_{n,k}} \Delta y_{n,k} - \sum_{n=0}^{N-1} \frac{\partial E_{j,k}}{\partial y_{n,k-1}} \Delta y_{n,k-1} \qquad \begin{array}{l} k=1,2,\ldots,M-1 \\ j=0,1,\ldots,N-1 \end{array}. \tag{2.7}$$

Equivalently this can be rewritten as

$$\begin{aligned} E_{j,k}\left(y_{j,k}, y_{j,k-1}\right) &= -\sum_{n=N}^{2N-1} \frac{\partial E_{j,k}}{\partial y_{n-N,k}} \Delta y_{n-N,k} - \sum_{n=0}^{N-1} \frac{\partial E_{j,k}}{\partial y_{n,k-1}} \Delta y_{n,k-1} \\ &= -\sum_{n=N}^{2N-1} S_{j,n} \Delta y_{n-N,k} - \sum_{n=0}^{N-1} S_{j,n} \Delta y_{n,k-1} \qquad \begin{array}{l} k=1,2,\ldots,M-1 \\ j=0,1,\ldots,N-1 \end{array} \end{aligned} \tag{2.8}$$

where

$$S_{j,n+N} = \frac{\partial E_{j,k}}{\partial y_{n,k}} \qquad \text{and} \qquad S_{j,n} = \frac{\partial E_{j,k}}{\partial y_{n,k-1}} \qquad n=0,1,\ldots,N-1.$$

Applying the same argument to the boundary equations 2.4 - which only depend on the values $y_{j,0}$ for $j=n_2, n_2+1, \ldots, N-1$ - yields

$$E_{j,0}\left(y_{j,0}\right) = -\sum_{n=N}^{2N-1} S_{j,n} \Delta y_{n-N,0} \qquad j=n_2, n_2+1, \ldots, N-1 \tag{2.9}$$

where

$$S_{j,n+N} = \frac{\partial E_{j,0}}{\partial y_{n,0}} \qquad n=0,1,\ldots,N-1.$$

Similarly, applying the argument again to equation 2.5 - which only depend on the values $y_{j,M-1}$ for $j=0,1,\ldots,n_2-1$ - yields

$$E_{j,M}\left(y_{j,M-1}\right) = -\sum_{n=0}^{N-1} S_{j,n} \Delta y_{n,M-1} \qquad j=0,1,\ldots,n_2-1 \tag{2.10}$$

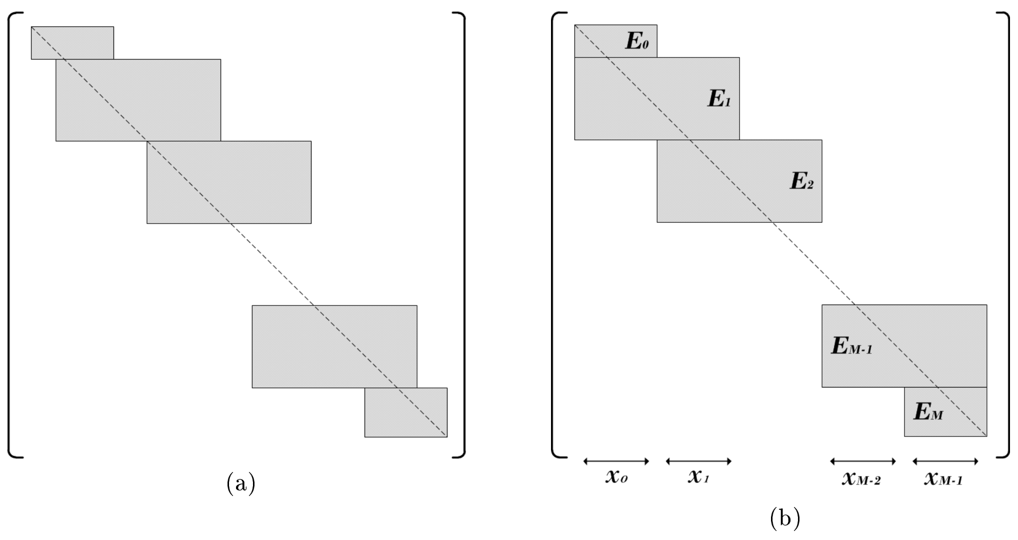


Figure 2.1: Almost block diagonal matrices (ABD): (a) shows the general form of ABD matrices, while (b) shows the special form of ABD matrices that typically arise from two-point boundary value problems.

where

$$S_{j,n} = \frac{\partial E_{j,M}}{\partial y_{n,M-1}} \qquad n = 0, 1, ..., N-1.$$

Equations 2.8-2.10, which are linear in $\Delta y_{j,k}$, must be solved simultaneously to find the values of $\Delta y_{j,k}$ - hence, a matrix must be inverted. For a large number of equations and/or grid points this task could be computational non-trivial, however, the structure of equations 2.8-2.10 yields a matrix of a very particular form. This structure can be exploited to greatly reduce the computational effort required to invert that matrix.

Almost block diagonal matrix structure

The matrix is an almost block diagonal (ABD) matrix - which are a subset of staircase matrices. In the most general ABD matrices there may be non-zero blocks of differing sizes; at most, each column intersects two successive blocks; the overlap between blocks is not necessarily constant; and each diagonal entry must lie in a block: see figure 2.1a. However, two-point boundary value problems produce an even more restricted ABD form in which the blocks $\boldsymbol{E}_1, \boldsymbol{E}_2 \ldots, \boldsymbol{E}_{M-1}$ are all of the same size - $N \times 2N$ in the problem at hand; the overlap between blocks is constant and is equal to the number of rows in $\boldsymbol{E}_0$ plus the number of rows in $\boldsymbol{E}_M$ - that is, N in total for this problem: see figure 2.1b. Blocks $\boldsymbol{E}_1, \boldsymbol{E}_2 \ldots, \boldsymbol{E}_{M-1}$ correspond to equation 2.8, while blocks $\boldsymbol{E}_0$ and $\boldsymbol{E}_M$ correspond to equations 2.9 and 2.10 respectively.

Although this structure is quite particular, it is hardly surprising that it has arisen in this two point boundary value problem. Equations 2.8-2.10 couple pairs of points x_k and x_{k-1} together. This splits the matrix into vertical bands of width N, each corresponding to a point on the grid, as is shown at the bottom of Figure 2.1b and the top of equation 2.11. As such, it is perhaps more natural to think of each of the blocks $\boldsymbol{E}_1, \boldsymbol{E}_2 \ldots, \boldsymbol{E}_{M-1}$ as pairs of blocks of size N^2 side by side, i.e. $\boldsymbol{E}_k = \boldsymbol{A}_{k,1} \star \boldsymbol{A}_{k,2}$ for $k = 1, 2, \ldots, M-1$ where $\star$ denotes that the matrices have been concatenated: see equation 2.11.

$$\begin{pmatrix} \overbrace{\boldsymbol{E}_0}^{x_0} & \overbrace{}^{x_1} & \cdots & \cdots & \overbrace{}^{x_{M-1}} \\ \boldsymbol{A}_{1,1} & \boldsymbol{A}_{1,2} & & & \\ & \boldsymbol{A}_{2,1} & \boldsymbol{A}_{2,2} & & \\ & & \ddots & \ddots & \\ & & & \boldsymbol{A}_{M-1,1} & \boldsymbol{A}_{M-1,2} \\ & & & & \boldsymbol{E}_M \end{pmatrix} \begin{pmatrix} \Delta \boldsymbol{y}_0 \\ \Delta \boldsymbol{y}_1 \\ \Delta \boldsymbol{y}_2 \\ \vdots \\ \Delta \boldsymbol{y}_{M-1} \\ \Delta \boldsymbol{y}_M \end{pmatrix} = \begin{pmatrix} \boldsymbol{C}_0 \\ \boldsymbol{C}_1 \\ \boldsymbol{C}_2 \\ \vdots \\ \boldsymbol{C}_{M-1} \\ \boldsymbol{C}_M \end{pmatrix}, \tag{2.11a}$$

With this change of viewpoint, the dependence of the structure of the matrix on the grid is now more explicit. Each of the square blocks $\boldsymbol{A}_{k,\{1,2\}}$ are Jacobian matrices, hence equation 2.11 corresponds to a system of coupled Jacobian matrices at each of the points x_k for $k = 1, 2, \ldots, \ldots, M-1$. The two boundary blocks, $\boldsymbol{E}_0$ and $\boldsymbol{E}_M$, correspond to partially complete Jacobians - since the boundary conditions are only partially specified at each boundary point. In detail,

$$\begin{aligned}
\boldsymbol{A}_{k,1} &= \begin{pmatrix} \frac{\partial E_{0,k}}{\partial y_{0,k-1}} & \cdots & \frac{\partial E_{0,k}}{\partial y_{N-1,k-1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E_{N-1,k}}{\partial y_{0,k-1}} & \cdots & \frac{\partial E_{N-1,k}}{\partial y_{N-1,k-1}} \end{pmatrix} = \begin{pmatrix} S_{0,0} & \cdots & S_{0,N-1} \\ \vdots & \ddots & \vdots \\ S_{N-1,0} & \cdots & S_{N-1,N-1} \end{pmatrix}, \\
\boldsymbol{A}_{k,2} &= \begin{pmatrix} \frac{\partial E_{0,k}}{\partial y_{0,k}} & \cdots & \frac{\partial E_{0,k}}{\partial y_{N-1,k}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E_{N-1,k}}{\partial y_{0,k}} & \cdots & \frac{\partial E_{N-1,k}}{\partial y_{N-1,k}} \end{pmatrix} = \begin{pmatrix} S_{0,N} & \cdots & S_{0,2N-1} \\ \vdots & \ddots & \vdots \\ S_{N-1,N} & \cdots & S_{N-1,2N-1} \end{pmatrix}, \\
\boldsymbol{E}_0 &= \begin{pmatrix} \frac{\partial E_{n_2,0}}{\partial y_{0,0}} & \cdots & \frac{\partial E_{n_2,0}}{\partial y_{N-1,0}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E_{N-1,0}}{\partial y_{0,0}} & \cdots & \frac{\partial E_{N-1,0}}{\partial y_{N-1,0}} \end{pmatrix} = \begin{pmatrix} S_{n_2,0} & \cdots & S_{n_2,N-1} \\ \vdots & \ddots & \vdots \\ S_{N-1,0} & \cdots & S_{N-1,N-1} \end{pmatrix}, \\
\boldsymbol{E}_M &= \begin{pmatrix} \frac{\partial E_{0,M}}{\partial y_{0,M-1}} & \cdots & \frac{\partial E_{0,M}}{\partial y_{N-1,M-1}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E_{n_2-1,M}}{\partial y_{0,M-1}} & \cdots & \frac{\partial E_{n_2-1,M}}{\partial y_{N-1,M-1}} \end{pmatrix} = \begin{pmatrix} S_{0,0} & \cdots & S_{0,N-1} \\ \vdots & \ddots & \vdots \\ S_{n_2-1,0} & \cdots & S_{n_2-1,N-1} \end{pmatrix},
\end{aligned} \tag{2.11b}$$

$$\begin{aligned}
\boldsymbol{C}_k &= -\Big(E_{0,k}\left(y_{0,k}, y_{0,k-1}\right)\cdots\ E_{N-1,k}\left(y_{N-1,k}, y_{N-1,k-1}\right)\Big)^{\mathrm{T}}, & \Delta\boldsymbol{y}_k &= \Big(\Delta y_{0,k}\cdots\ \Delta y_{N-1,k}\Big)^{\mathrm{T}},\\
\boldsymbol{C}_0 &= -\Big(E_{n_2,0}\left(y_{0,0}\right)\cdots\ E_{N-1,0}\left(y_{N-1,0}\right)\Big)^{\mathrm{T}}, & \Delta\boldsymbol{y}_0 &= \Big(\Delta y_{n_2,0}\cdots\ \Delta y_{N-1,0}\Big)^{\mathrm{T}},\\
\boldsymbol{C}_M &= \Big(E_{0,M}\left(y_{0,M-1}\right)\cdots\ E_{n_2-1,M}\left(y_{N-1,M-1}\right)\Big)^{\mathrm{T}}, & \Delta\boldsymbol{y}_M &= \Big(\Delta y_{0,M-1}\cdots\ \Delta y_{n_2-1,M-1}\Big)^{\mathrm{T}}.
\end{aligned} \tag{2.11c}$$

where $k = 1, 2, \ldots, M-1$. The ABD structure - which was introduced without justification at the start of this section - is explicitly clear now in these equations.

As previously mentioned, this ABD structure can be exploited to greatly reduce the computational effort required to invert this matrix and this is an important feature of the method presented in this thesis. Since the relaxation is an iterative procedure with an inversion of the above matrix required at each iteration, any speed-up in the matrix inversion will greatly speed-up the relaxation method as a whole. An efficient ABD matrix will be presented momentarily, but first the iterative process will be discussed in detail.

Iteration

Once the correction values $\Delta y_{j,k}$ have been found - that is, the matrix has been inverted - they must be utilised to improve the approximate solution $y_{j,k}$. A convergence rate control value r will determine the size of the updates that are applied, that is, $y_{j,k}^{\text{new}} = y_{j,k}^{\text{old}} + r\Delta y_{j,k}$. It would perhaps be tempting to apply the entire correction in one go (i.e. $r = 1$), however, this will in general not satisfy the solution to the equations to the desired accuracy since the correction values were calculated via linear approximations of the non-linear equations. Hence, an iterative procedure should be applied to approach the solution of the full non-linear equations. A tolerance value err_{conv} should be used to control when the iteration process should halt, that is, reaches the desired accuracy. Furthermore, the linear approximation at $y_{j,k}$ may not sufficiently capture the behaviour of the equations at $y_{j,k} + \Delta y_{j,k}$. As such, convergence could potentially become an issue since a linear approximation is assumed to be sufficient when it may not necessarily be sufficient at capturing the behaviour of the equations. The value r aims to somewhat mitigate these problems. It should be chosen in such a way that it controls the size of the updates so that they remain within an annulus of $y_{j,k}$ where the linear approximation remains reasonably valid.

In order to choose an appropriate value for r then there must be an estimate the validity of the linear approximation. Hence, there must be an estimate of the potential size of the error. There is no unique or special way of making this estimate, however in the implementation presented, the average of the absolute value of relative magnitude of the corrections $\Delta y_{j,k}$ is used as an error estimate, *i.e.*

$$err_{\text{norm}} = \frac{1}{MN}\sum_{j=0}^{N-1}\sum_{k=0}^{M-1}\frac{|\Delta y_{j,k}|}{\text{scale}_j}, \tag{2.12}$$

where scale_j is a measure of the typical size of the values $y_{j,k}$ for a particular family of values denoted by j. The convergence rate parameter c is then taken to be

$$r = \frac{r_{\text{ control}}}{\max\left(err_{\text{norm}}, r_{\text{ control}}\right)}, \tag{2.13}$$

where $r_{\text{ control}}$ is a further parameter supplied by the user. When $err_{\text{norm}} > r_{\text{ control}}$ then only a fraction of the corrections $\Delta y_{j,k}$ are used and so it is hoped the value $y^{\text{new}}_{j,k}$ has remained within an annulus of $y_{j,k}$ where the linear approximation is reasonably valid. If $err < r_{\text{control}}$ then the entire correction is applied. Once the value of err_{norm} falls below the tolerance parameter err_{conv} the relaxation process is complete: the algorithm should be halted before iterating again and the values of the eigensystem should be returned.

Efficient block-diagonal matrix solvers

Any algorithm for inverting the matrix is sufficient to invert an ABD matrix. However, as mentioned above, this ABD structure can be exploited to greatly reduce the computational effort required to invert that matrix. For example, regular Gaussian elimination and back-substitution could be used to invert this matrix, however, this would be a foolhardy endeavour as using Gaussian elimination would introduce matrix fill-in, which would ruin any potential computational speed-up that might be gained from the ABD structure - any ABD solver should aim to minimise if not completely avoid fill-in. For a substantial discussion of ABD matrix solvers (including parallel solvers), and in particular, the even more restricted subset of ABD matrices that arise in two-point boundary value problems, see [15].

The ABD matrix solver used in this thesis, as a proof of concept, is a modified form of Gaussian elimination which aims to minimise matrix fill-in. This method is based on an ABD solver found in *Numerical Recipes* [14, pp. 966-970]. However, it has been modified so that it utilises *GNU Scientific Library* (GSL) data types [16]. This solver aims to minimise the total number of operations required. In addition, by minimising the total number of matrix coefficients that must be stored at each iteration, the solver has relatively small memory footprint in comparison to other matrix solvers.

Rather than operating on the matrix as a single block, the modified Gaussian elimination runs along the stairwell of blocks and - along with some housekeeping -

performs Gaussian elimination on each of the blocks sequentially. This will reduce the matrix to a series of ones on the diagonal and a limited number of non-zero entries in the regions of overlap between the coupled Jacobian matrices - the housekeeping mentioned above is the storage of these few non-zero entries. Back-substitution can then be used to obtain the solution. The entire process consists of four main processes:

1. using results from the previous block to reduce certain elements in the current block to zero in the region of overlap between the two block (see the subroutine `red`);
2. Gaussian elimination of the current to reduce the remaining elements of the left-most square sub-block to an identity matrix (see subroutine `pinvs`);
3. storage of the limited number of remaining non-zero entries on the right of the current matrix for use in back-substitution (see subroutine `pinvs`);
4. and back-substitution (see subroutine `bksub`).

Starting in the top left corner of the ABD matrix, the solver starts on step 2 and then iterates through steps 1-3 until the entire ABD matrix has been reduced; step 4 is then used to yield the solution: this process is implemented by the routine `Solvde`. For full details about the algorithm used see [14, pp. 966-970]. To see the GSL modified algorithm used in this implementation, see the code found in the appendix.

Applying the relaxation method

Thus far, the relaxation method for general two-point boundary value problems has been described with particular attention paid to the structure of the matrix that must be solved upon each iteration of the algorithm. An ABD matrix of partial derivatives must be supplied to the algorithm by the user. These partial derivatives will come from the grid of finite difference equations on the domain of the problem and from the boundary conditions. The user must also supply an initial guess solution at the start of the relaxation process. The user will also specify some parameters to control the convergence of the algorithm as well as the halting of the algorithm.

It is now time to turn to the specific problem at hand, that is, the numerical evaluation of the spin-weighted spheroidal harmonics. In the next section, the specific details of utilising the algorithm outlined in this section, to the spin-weighted spheroidal differential equation and its boundary conditions will be developed.

3 Implementation

3.1 The Frobenius Method

The first step that shall be undertaken is to investigate the behaviour of the spin-weighted spheroidal differential equation 1.2 near its singular points at $x = \pm 1$. The Frobenius method provides a way of constructing infinite series solutions of second order ODEs in the region near regular singular points. This then allows the singular behaviour to be factored out. This will provide a representation that is numerically more tractable then the original equation.

First, consider the singular point at $x = -1$. The aim is to factor out the singular behaviour at this singular point by constructing a power series $z^{-}(x)$ that is valid near $x = -1$. As such, consider the series

$$\begin{aligned} {}_sS_{lmc}(x) &= (1+x)^{k_-}\ {}_sz^{-}_{lmc}(x) && (3.1a)\\ &= (1+x)^{k_-}\sum_{\lambda=0}^{\infty} a^{-}_{\lambda}(1+x)^{\lambda} && (3.1b)\\ &= \sum_{\lambda=0}^{\infty} a^{-}_{\lambda}(1+x)^{k_-+\lambda} && (3.1c)\end{aligned}$$

where the exponent k_- and the coefficients a^{-}_{λ} have yet to be determined. Here $a_0 \neq 0$ as it is chosen to be the coefficient of the the lowest non-vanishing term of the series. In this series the dependent variable is now ${}_sz^{-}_{lmc}(x)$ as opposed to ${}_sS_{lmc}(x)$.

This trial is then inserted into the spin-weighted spheroidal differential equation 1.2, i.e.

$$\left[\frac{\mathrm{d}}{\mathrm{d}x}\left(\left(1-x^2\right)\frac{\mathrm{d}}{\mathrm{d}x}\right) + c^2x^2 - 2csx - \frac{(m+sx)^2}{1-x^2} + s + {}_sA_{lmc}\right](1+x)^{k_-}\ {}_sz^{-}_{lmc}(x) = 0. \tag{3.2}$$

Every term in this result can be expanded trivially in terms of $(1+x)$. From the uniqueness of power series, the coefficients of each power of $(1+x)$ must individually be equal to zero. In particular, the term of the lowest power in the series must equal zero: it will be of order $\mathcal{O}\left((1+x)^{-1}\right)$ and its coefficient is

$$\left[a^{-}_0\frac{(m-s)^2}{2} - 2a^{-}_0k_-^2\right] = 0. \tag{3.3}$$

Since it has been chosen that $a^{-}_0 \neq 0$, the uniqueness of power series demands that the regular solution has

$$k_- = \frac{|m-s|}{2}. \tag{3.4}$$

This equation, which as was seen, comes from the coefficient of the lowest term in the power series expansion of equation 3.2 and the demand of uniqueness of power series representation, is known as the *indicial equation*. Hence, a value for k_- in equations 3.1 has been obtained.

Similarly, the Frobenius method can be applied to construct a series solution ${}_sz^+_{lm}(x)$, which is valid near the singular point at $x = +1$;

$$\begin{aligned} {}_sS_{lmc}(x) &= (1-x)^{k_+}\ {}_sz^+_{lmc}(x) &\quad (3.5a)\\ &= (1-x)^{k_+} \sum_{\lambda=0}^{\infty} a^+_\lambda (1-x)^\lambda &\quad (3.5b)\\ &= \sum_{\lambda=0}^{\infty} a^+_\lambda (1-x)^{k_+ + \lambda}. &\quad (3.5c)\end{aligned}$$

In this series the dependent variable is now ${}_sz^+_{lm}(x)$ as opposed to ${}_sS_{lm}(x)$. It is then inserted into the spin-weighted spheroidal differential equation 1.2, i.e.

$$\left[\frac{\mathrm{d}}{\mathrm{d}x}\left(\left(1-x^2\right)\frac{\mathrm{d}}{\mathrm{d}x}\right) + c^2x^2 - 2csx - \frac{(m+sx)^2}{1-x^2} + s + {}_sA_{lmc}\right](1-x)^{k_+}\ {}_sz^+_{lmc}(x) = 0. \tag{3.6}$$

This time, every term can be expanded trivially in terms of $(1-x)$. In particular, the term of the lowest power will be will be of order $\mathcal{O}\left((1-x)^{-1}\right)$ and its coefficient is

$$\left[a^+_0 \frac{(m+s)^2}{2} - 2a^+_0 k^2_+\right] = 0, \tag{3.7}$$

and so the indicial equation gives the value of k_+ to be

$$k_+ = \frac{|m+s|}{2}. \tag{3.8}$$

Thus far, for each series, only the indicial equation has been found - that is, the demand of the uniqueness of a power series representation being applied to the coefficient of the lowest order term in the power series expansions of equations 3.6 and 3.2. However, the uniqueness of power series actually demands that the coefficients of every term in each of the power series expansions must individually be equal to zero. Demanding that these coefficients each individually go to zero allows recursion relations between the coefficients a_λ to be developed for both series. Hence, with this in mind, consider the term of the second lowest order in the expansion of equation 3.2, which is the term of $\mathcal{O}(1)$; its coefficient is

$$\begin{aligned} a^-_1 &\left(16k_- + 8k^2_- - 2\left(-4 + m^2 - 2ms + s^2\right)\right)\dots \\ &\dots + a^-_0\left(4{}_sA_{lmc} + 4c^2 - 4k_- - 4k^2_- - m^2 + 4s + 8cs - 2ms + 3s^2\right) = 0. \end{aligned} \tag{3.9}$$

Similarly, the term of the second lowest order in the expansion of equation 3.6, which is the term of $\mathcal{O}(1)$; its coefficient is

$$a_1^+ \left(16k_+ + 8k_+^2 - 2\left(-4 + m^2 + 2ms + s^2\right)\right) \ldots$$
$$\ldots + a_0^+ \left(4_s A_{lmc} + 4c^2 - 4k_+ - 4k_+^2 - m^2 + 4s - 8cs + 2ms + 3s^2\right) = 0. \quad (3.10)$$

These relations can be developed into relationships between the solution/eigenfunction; its derivative; and its eigenvalue, at each of the boundaries - and they will be utilised to this end in the next section. Now, one could continue to develop the recursion relations for coefficients of even higher order terms, however, equations 3.9 and 3.10 will be sufficient for the present need.

3.2 Change of dependent variable

Combining the power series 3.1 and 3.5 into one new power series ${}_s z_{lm}(x)$ yields a power series in which the singular behaviour at $x = -1$ and $x = +1$ has been factored out:

$${}_s S_{lmc}(x) = (1+x)^{k_-} (1-x)^{k_+} \, {}_s z_{lmc}(x) \quad (3.11)$$

The idea is to use this function ${}_s z_{lmc}(x)$ to do the numerical ground work via a change of dependant variable from ${}_s S_{lmc}(x)$ to ${}_s z_{lmc}(x)$. It should be numerically more tractable than any regular power series representation of ${}_s S_{lmc}(x)$ since the singular points have now been factored out.

A new ODE

With this in mind, the series solution 3.11 is inserted into the spin-weighted spheroidal differential equation 1.2 which yields an ODE in the new dependent variable ${}_s z_{lmc}(x)$, that is

$$\left\{ \left(1-x^2\right) \frac{\mathrm{d}^2}{\mathrm{d}x^2} - 2\left[k_+ - k_- + x\left(1 + k_+ + k_-\right)\right] \frac{\mathrm{d}}{\mathrm{d}x} \right.$$
$$\left. + \left[{}_s A_{lm} + s(s+1) + c^2x^2 - 2csx - (k_+ + k_-)(k_+ + k_- + 1)\right] \right\} {}_s z_{lmc}(x) = 0 \quad (3.12)$$

where the insertion $0 = m^2 + s^2 - 2k_+^2 - 2k_-^2$ has been used to reduced the ODE to this succinct form[1]. It is this ODE that will be transformed into an FDE, hence this equation will lead on to equation 2.8 to be used in the relaxation process.

[1]For further details about the reduction to this succinct form see the appendix.

Boundary conditions on the new ODE

Having developed the ODE in this new dependent variable ${}_sz_{lmc}(x)$, a set of boundary conditions in this variable must also be developed. Consider the $x = -1$ boundary: from equations 3.11 and 3.1a it can be seen that

$${}_sz^-_{lmc}(x) = (1-x)^{k_+}\,{}_sz_{lmc}(x), \tag{3.13}$$

which can be series expanded about $(1+x)$, yielding

$$\begin{aligned}{}_sz^-_{lmc}(x) &= 2^{k_+}\,{}_sz_{lmc}|_{x=-1}\cdots\\ &\quad\ldots - 2^{k_+-1}\left(k_{+s}z_{lmc}|_{x=-1} - 2_sz'_{lmc}|_{x=-1}\right)(1+x) + \mathcal{O}\left((1+x)^2\right).\end{aligned} \tag{3.14}$$

By the uniqueness of power series and by comparing equation 3.14 to equation 3.1b, it is required that

$$a^-_0 = 2^{k_+}\,{}_sz_{lmc}|_{x=-1}, \qquad a^-_1 = k_{+s}z_{lmc}|_{x=-1} - 2_sz'_{lmc}|_{x=-1}, \tag{3.15}$$

hence

$$\frac{a^-_1}{a^-_0} = \frac{{}_sz'_{lmc}|_{x=-1}}{{}_sz_{lmc}|_{x=-1}} - \frac{k_+}{2}. \tag{3.16}$$

Finally, by making use of equation 3.9 and rearranging, a boundary condition at $x = -1$ is obtained

$$\frac{{}_sz'_{lmc}|_{x=-1}}{{}_sz_{lmc}|_{x=-1}} = \left(\frac{-4_sA_{lm} - 4c^2 + 4k_- + 4k_-^2 + m^2 - 4s - 8cs + 2ms - 3s^2}{16k_- + 8k_-^2 - 2\left(-4 + m^2 - 2ms + s^2\right)} + \frac{k_+}{2}\right). \tag{3.17}$$

Mutatis mutandis, a boundary condition at $x = -1$ is also obtained

$$\frac{{}_sz'_{lmc}|_{x=+1}}{{}_sz_{lmc}|_{x=+1}} = \left(\frac{4_sA_{lm} + 4c^2 - 4k_+ - 4k_+^2 - m^2 + 4s - 8cs + 2ms + 3s^2}{16k_+ + 8k_+^2 - 2\left(-4 + m^2 + 2ms + s^2\right)} - \frac{k_-}{2}\right). \tag{3.18}$$

Normalisation condition

At this point, a numerically tractable second order ODE (equation 3.12), along with two boundary conditions on this ODE (equations 3.17 and 3.18), have been developed. However, this is an eigenvalue problem; the parameter ${}_sA_{lm}$ is undetermined. As such, implicitly there will be another first order ODE, namely the ODE arising from the eigenvalue problem

$$\frac{\mathrm{d}_sA_{lm}}{\mathrm{d}x} = 0, \tag{3.19}$$

which allows the eigenvalue ${}_sA_{lm}$ to be determined. Now when the second order ODE is split into two coupled first order ODEs, there will be three first order ODEs and only two boundary conditions - which would be a problem. However, a third boundary condition arises from the fact that any constant multiple of a solution is itself a solutions. As such, there is freedom to choose a normalisation for the solution - this normalisation will act as the third boundary condition.

The particular choice of normalisation is not important, all that matters is that once chosen, the entire function adheres to the same chosen normalisation. As such, any *one point* on the domain could be chosen and any value of the function at that point could be specified, and this could be used as the normalisation condition - such is the freedom of this normalisation condition. However, the normalisation condition chosen here is that the function ${}_sS_{lmc}\left(x\right)$ - the spin-weighted spheroidal harmonics - has the same limiting behaviour as ${}_sY_{lm}\left(x\right)$ - the spin-weighted spherical harmonics - as $x \to +1$. This has two slight advantages over an arbitrary choice: firstly, it is similar to a common, recognisable normalisation convention of a closely related function; and secondly, the scale of the function ${}_sS_{lmc}\left(x\right)$ is predictable when using this normalisation condition. This is useful as the typical scale of the function ${}_sS_{lmc}\left(x\right)$ will be required *a priori* for the multidimensional Newton method (see equation 2.12). This choice of this normalisation condition leads to the following boundary condition at $x = +1$

$$\begin{aligned} \lim_{x\to 1}\left(1+x\right)^{-k_-}\left(1-x\right)^{-k_+}{}_sS_{lmc}\left(x\right) &= \lim_{x\to 1}\left(1+x\right)^{k_-}\left(1-x\right)^{k_+}{}_sY_{lm}\left(x\right) \\ &:= \gamma_r. \end{aligned} \tag{3.20}$$

The analytic value of γ_r will be calculated in a later section (see equation 3.44), but for now it is not required.

3.3 The numerical method

The second order ODE 3.12 is split into two first order ODEs and the eigenvalue problem - equation 3.19 - is posed, i.e.

$$y_0 := z\left(x\right), \qquad y_1 := \frac{\mathrm{d}z}{\mathrm{d}x}, \qquad y_2 := {}_sA_{lm},$$

with

$$y_0' = y_1, \tag{3.21a}$$

$$\begin{aligned} y_1' = \frac{1}{\left(1-x^2\right)}\Bigg\{&2\Big[k_+ - k_- + x\left(1+k_++k_-\right)\Big]y_1 \\ &+\Big[y_2 + s(s+1) + c^2x^2 - 2csx - (k_++k_-)(k_++k_-+1)\Big]y_0\Bigg\}, \end{aligned} \tag{3.21b}$$

$$y_3' = 0. \tag{3.21c}$$

The boundary condition, equation 3.17, at $x = -1$ in this notation is

$$\frac{y_1}{y_0} = \left(\frac{-4y_2 - 4c^2 + 4k_- + 4k_-^2 + m^2 - 4s - 8cs + 2ms - 3s^2}{16k_- + 8k_-^2 - 2\left(-4 + m^2 - 2ms + s^2\right)} + \frac{k_+}{2} \right), \tag{3.22a}$$

while at $x = 1$, equations 3.18 and 3.20, they are

$$\frac{y_1}{y_0} = \left(\frac{4y_2 + 4c^2 - 4k_+ - 4k_+^2 - m^2 + 4s - 8cs + 2ms + 3s^2}{16k_+ + 8k_+^2 - 2\left(-4 + m^2 + 2ms + s^2\right)} - \frac{k_-}{2} \right). \tag{3.23a}$$

$$y_0 = \lim_{x \to 1} (1 + x)^{k_-} (1 - x)^{k_+} {}_sY_{lm}(x) := \gamma_r. \tag{3.23b}$$

At this point, the numerically tractable equations for the boundary value problem have been specified. These equations must now be turned into finite difference equations so that the relaxation method can be applied.

Equations on the mesh

First, the numerical grid on the domain should be specified. For simplicity, a uniform grid of M mesh points on the interval $x = [-1, +1]$ will be used, hence

$$x_k = kh - 1, \qquad k = 0, 1, ..., M - 1, \tag{3.24}$$

where $h = 2/(M-1)$. On the interior points x_k for $k = 1, 2, \ldots, M-1$ equation 3.21a gives

$$E_{0,k} = y_{0,k} - y_{0,k-1} - \frac{h}{2}\left(y_{1,k} + y_{1,k-1}\right), \tag{3.25a}$$

where the one step trapezoidal method (equation 2.2) has been used to convert the ODE into an FDE. Equation 3.21b gives

$$\begin{aligned} E_{1,k} = y_{1,k} - y_{1,k-1} - \frac{h}{\left(1 - x_\kappa^2\right)} \Big\{ & 2\left[k_+ - k_- + x_\kappa\left(1 + k_+ + k_-\right)\right] y_{1,\kappa} \\ & + \left[y_{2,\kappa} + s(s+1) + c^2 x_\kappa^2 - 2csx_\kappa - (k_+ + k_-)(k_+ + k_- + 1)\right] y_{0,\kappa} \Big\}, \end{aligned} \tag{3.25b}$$

where

$$x_\kappa = \frac{x_k + x_{k-1}}{2}, \quad y_{0,\kappa} = \frac{y_{0,k} + y_{0,k-1}}{2}, \quad y_{1,\kappa} = \frac{y_{1,k} + y_{1,k-1}}{2}, \quad y_{2,\kappa} = \frac{y_{2,k} + y_{2,k-1}}{2}.$$

Finally, equation 3.21c yields

$$E_{2,k} = y_{2,k} - y_{2,k-1}. \tag{3.25c}$$

At $x = -1$ there is one boundary condition, equation 3.22a, which gives

$$E_{2,0} = y_{1,0} - y_{0,0}\left(\frac{-4y_{2,0} - 4c^2 + 4k_- + 4k_-^2 + m^2 - 4s - 8cs + 2ms - 3s^2}{16k_- + 8k_-^2 - 2\left(-4 + m^2 - 2ms + s^2\right)} + \frac{k_+}{2}\right), \tag{3.26a}$$

while at $x = +1$ there are two equations: equation 3.23a gives

$$E_{0,M} = y_{1,M} - y_{0,M}\left(\frac{4y_{2,M} + 4c^2 - 4k_+ - 4k_+^2 - m^2 + 4s - 8cs + 2ms + 3s^2}{16k_+ + 8k_+^2 - 2\left(-4 + m^2 + 2ms + s^2\right)} - \frac{k_-}{2}\right); \tag{3.27a}$$

and equation 3.23b gives

$$E_{1,M} = y_{0,M} - \gamma_r. \tag{3.27b}$$

S-matrix of partial derivatives

Now, recall the matrix of partial derivatives $S_{i,j}$ in equations 2.11. In particular, these partial derivatives arise from equations 2.8 - 2.10. In these equations $N = 3$, that is, there is a set of three coupled equations at each of the M points in the mesh - this can be explicitly seen in equations 3.21.

First, consider equation 2.8, which corresponds to the finite difference couplings on the interior of the grid points x_k for $k = 1, 2, \ldots, M - 1$. Here the label j can take the values $j = 0, 1, \ldots, N - 1$ while the label n can take on values $n = 0, 1, \ldots, 2N - 1$. In particular, since $N = 3$, the label j can take the values $j = 0, 1, 2$ while the label n can take on values $n = 0, 1, \ldots, 6$. This leads to the $N \times 2N$ matrices $\boldsymbol{E}_k = \boldsymbol{A}_{k,1} \star \boldsymbol{A}_{k,2}$ which have already been discussed in relation to equations 2.11. Equation 2.8 and equation 3.25a give

$$\begin{aligned} &S_{0,0} = -1, && S_{0,1} = -\frac{h}{2}, && S_{0,2} = 0, \\ &S_{0,3} = 1, && S_{0,4} = -\frac{h}{2}, && S_{0,5} = 0; \end{aligned} \tag{3.28a}$$

equation 2.8 and 3.25b give

$$\begin{aligned} S_{1,0} &= -\frac{h}{2\left(1 - x_\kappa^2\right)}\left[y_{2,\kappa} + s(s+1) + c^2x_\kappa^2 - 2csx_\kappa - (k_+ + k_-)(k_+ + k_- + 1)\right], \\ S_{1,1} &= -1 - \frac{h}{\left(1 - x_\kappa^2\right)}\left[k_+ - k_- + x_\kappa\left(1 + k_+ + k_-\right)\right], \qquad S_{1,2} = -\frac{h}{2\left(1 - x_\kappa^2\right)}y_{0,\kappa}, \\ S_{1,3} &= S_{1,0}, \qquad S_{1,4} = 2 + S_{1,1}, \qquad S_{1,5} = S_{1,2}; \end{aligned} \tag{3.28b}$$

while equation 2.8 and 3.25c give

$$\begin{aligned} &S_{0,0} = 0, && S_{0,1} = 0, && S_{0,2} = -1, \\ &S_{0,3} = 0, && S_{0,4} = 0, && S_{0,5} = 1. \end{aligned} \tag{3.28c}$$

Next, consider equation 2.9, which corresponds to the boundary conditions at $x = -1$. Here the label j can take the values $j = n_2, n_{2+1}, \ldots, N-1$ while the label n can take on values $n = 0, 1, \ldots, N-1$. In particular, since $N = 3$ and there is only one equation at this boundary, *i.e.* $n_1 = 1 \implies n_2 \equiv N - n_1 = 2$. Hence, $j = 2$ while $n = 0, 1, \ldots, 3$ and so equations 2.9 and 3.26a give

$$S_{2,3} = -\left(\frac{-4y_{2,0} - 4c^2 + 4k_- + 4k_-^2 + m^2 - 4s - 8cs + 2ms - 3s^2}{16k_- + 8k_-^2 - 2\left(-4 + m^2 - 2ms + s^2\right)} + \frac{k_+}{2}\right),$$

$$S_{2,4} = 1, \qquad S_{2,5} = \left(\frac{4y_{0,0}}{16k_- + 8k_-^2 - 2\left(-4 + m^2 - 2ms + s^2\right)}\right). \tag{3.29a}$$

Consider equation 2.10, which corresponds to the boundary conditions at $x = +1$. Here the label j can take the values $j = 0, 1, \ldots, n_2 - 1$ while the label n can take on values $n = 0, 1, \ldots, N-1$. In particular, since $N = 3$ and $n_2 = 2$, the possible values for the indices are $j = 0, 1$ and $n = 0, 1, \ldots, 3$ and so equations 2.10 and 3.27a give

$$S_{0,3} = -\left(\frac{4y_{2,M} + 4c^2 - 4k_+ - 4k_+^2 - m^2 + 4s - 8cs + 2ms + 3s^2}{16k_+ + 8k_+^2 - 2\left(-4 + m^2 + 2ms + s^2\right)} - \frac{k_-}{2}\right),$$

$$S_{0,4} = 1, \qquad S_{0,5} = \left(\frac{-4y_{0,M}}{16k_+ + 8k_+^2 - 2\left(-4 + m^2 + 2ms + s^2\right)}\right), \tag{3.30a}$$

while equations 2.10 and 3.27b give

$$S_{1,3} = 1, \qquad S_{1,4} = 0, \qquad S_{1,5} = 0. \tag{3.30b}$$

By now all of the necessary equations have been found and the S matrix of partial derivatives has been completely specified in this new dependent variable. However, there are still several details to be discussed. In particular, the initial guess solution for the relaxation process.

3.4 Initial trial solution

Analytic values for spin-weighted spherical harmonics

As previously mentioned in section 2, the spin-weighted spherical harmonics, that correspond to $c = 0$, will be used as the starting point for any path in the complex c plane. Both the eigenfunctions ${}_sY_{lm}(x)$ and corresponding eigenvalues ${}_s\lambda_{lm}$ are known

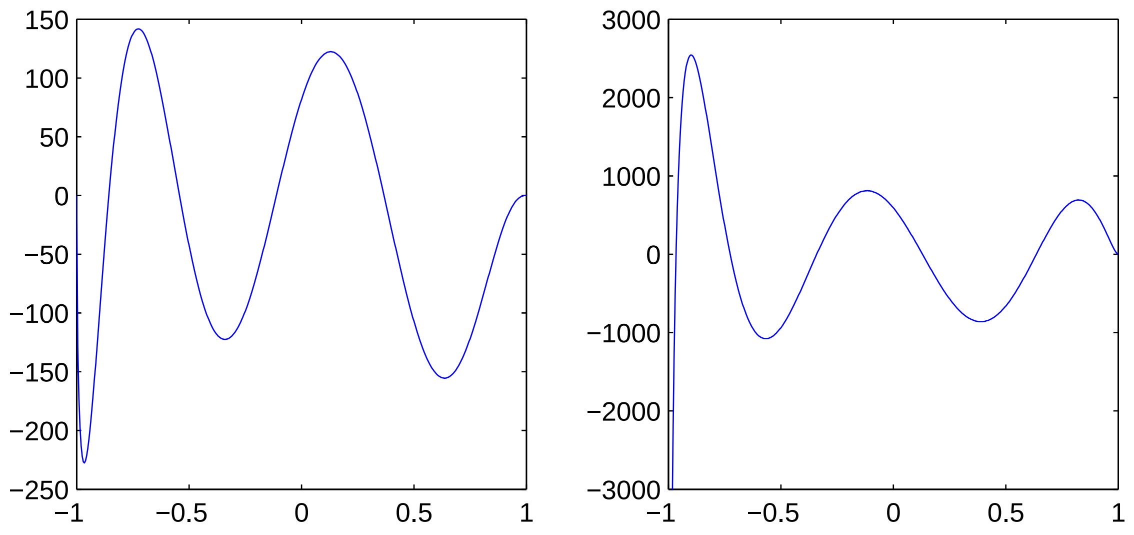


Figure 3.1: The solution to the spin-weighted spherical differential equation ${}_sY_{lm}\left(\theta,\phi\right)$ (left) and its derivative ${}_sY'_{lm}\left(\theta,\phi\right)$ (right) for $s=2$, $l=7$ and $m=3$.

analytically for $c=0$. Specifically the eigenfunctions are given by

$$
\begin{aligned}
{}_sY_{lm}\left(x\right) = \left(-1\right)^{m+l-s}\sqrt{\frac{\left(l+m\right)!\,\left(l-m\right)!\,\left(2l+1\right)}{\left(l+s\right)!\,\left(l-s\right)!\,4\pi}}\left[\sin\left(\frac{\cos^{-1}\left(x\right)}{2}\right)\right]\dots \\
\dots\sum_{r=0}^{l-s}\left\{\left(-1\right)^{-r}\binom{l-s}{r}\binom{l+s}{r+s-m}\left[\cot\left(\frac{\cos^{-1}\left(x\right)}{2}\right)\right]^{2r+s-m}\right\}
\end{aligned}
\tag{3.31}
$$

while the corresponding eigenvalue is given by

$$
{}_s\lambda_{lm} = l\left(l+1\right) - s\left(s+1\right). \tag{3.32}
$$

The derivative of the eigenfunction will also be need for the initial guess

$$
\begin{aligned}
\frac{\mathrm{d}{}_sY_{lm}}{\mathrm{d}x} = \left(-1\right)^{m+l-s}\sqrt{\frac{\left(l+m\right)!\,\left(l-m\right)!\,\left(2l+1\right)}{\left(l+s\right)!\,\left(l-s\right)!\,4\pi}}\dots \\
\dots\sum_{r=0}^{l-s}\left\{\left(-1\right)^{-r}\binom{l-s}{r}\binom{l+s}{r+s-m}\left[-\frac{\left(lx+l+m-2r-s\right)}{2\sqrt{1-x^2}}\dots\right.\right. \\
\left.\left.\left[\cot\left(\frac{\cos^{-1}\left(x\right)}{2}\right)\right]^{2r+s-m}\sec\left(\frac{\cos^{-1}\left(x\right)}{2}\right)\sin\left(\frac{\cos^{-1}\left(x\right)}{2}\right)^{2l-1}\right]\right\}.
\end{aligned}
\tag{3.33}
$$

Now, it should be noted that here the generalised definition of the binomial coefficient

$$
\binom{n}{k} = \begin{cases} \dfrac{n!}{k!\,\left(n-k\right)!} & \text{for } 0 \le k < n, \\ 0 & \text{otherwise,} \end{cases} \tag{3.34}
$$

is being used. An example of a spin-weighed spherical harmonic can be seen in Figure 3.1.

Change of dependent variable

The analytical results above give ${}_sS_{lmc}(x)$ for $c = 0$ - *i.e.* ${}_sY_{lm}(x)$. However, the numerical work will not be performed on the variable ${}_sS_{lmc}(x)$, but rather on the variable ${}_sz_{lmc}(x)$ from equation 3.11. Hence, a change of dependent variable must be performed on ${}_sY_{lm}(x)$ so that the initial guess in also in the correct dependent variable. Clearly, for ${}_sY_{lm}(x)$ the change is trivial

$$ {}_sy_{lm}(x) = (1+x)^{-k_-}(1-x)^{-k_+}\,{}_sY_{lm}(x), \tag{3.35}$$

where ${}_sy_{lm}(x)$ is to ${}_sY_{lm}(x)$ as ${}_sz_{lmc}(x)$ is to ${}_sS_{lmc}(x)$. For the derivative, the change is also trivial

$$\begin{aligned}\frac{\mathrm{d}_sy_{lm}}{\mathrm{d}x} &= (1-x)^{-1}(1+x)^{-1}\left[k_+(1+x) - k_-(1-x)\right]{}_sY_{lm}(x)\\ &\quad + (1-x)^{-1-k_+}(1+x)^{-1-k_-}\left(1-x^2\right)\frac{\mathrm{d}_sY_{lm}}{\mathrm{d}x}.\end{aligned} \tag{3.36}$$

However, equations 3.35 and 3.36 are not defined at $x = \pm 1$, hence the following limits must be calculated

$$\lim_{x\to -1} {}_sy_{lm}(x) = \lim_{x\to -1}(1+x)^{-k_-}(1-x)^{-k_+}\,{}_sY_{lm}(x), \tag{3.37a}$$

$$\lim_{x\to +1} {}_sy_{lm}(x) = \lim_{x\to +1}(1+x)^{-k_-}(1-x)^{-k_+}\,{}_sY_{lm}(x). \tag{3.37b}$$

If these limits are evaluated then it is trivial to find the value of the derivative at the boundary by using the boundary conditions, *i.e.* equations 3.17 and 3.18.

To calculate the limits, first consider the following half angle expressions

$$\cos\left(\frac{\theta}{2}\right) = \sqrt{\frac{1+\cos\theta}{2}} = \sqrt{\frac{1+x}{2}}, \tag{3.38}$$

$$\sin\left(\frac{\theta}{2}\right) = \sqrt{\frac{1-\cos\theta}{2}} = \sqrt{\frac{1-x}{2}}, \tag{3.39}$$

where the fact that $x = \cos(\theta)$ has been used. These half angle expressions allow equation 3.31 to be expressed in terms of x and hence equation 3.35 can be written as

$$\begin{aligned}{}_sy_{lm}(x) &= (1+x)^{-k_-}(1-x)^{-k_+}(-1)^{m+l-s}\sqrt{\frac{(l+m)!\,(l-m)!\,(2l+1)}{(l+s)!\,(l-s)!\,4\pi}}\\ &\left(\frac{1-x}{2}\right)^{l}\left(\frac{1+x}{1-x}\right)^{(s-m)/2}\sum_{r=0}^{l-s}\left\{(-1)^{-r}\binom{l-s}{r}\binom{l+s}{r+s-m}\left(\frac{1+x}{1-x}\right)^{r}\right\}.\end{aligned} \tag{3.40}$$

This form allows the limiting behaviour to be determined much more readily, however there is a caveat. For the sake of argument, consider the zeroth term in the series expansion of this expression about $x = -1$, which might be assumed to pertain to the $r = 0$ term in the sum - this zeroth order term would be equal to the limit at $x = -1$ - *i.e.*

$$\lim_{x\to -1} {}_sy_{lm}(x) = (-1)^l \, 2^{[-k_+ - (m-s)/2]} \sqrt{\frac{(l+m)!\,(l-s)!\,(2l+1)}{(l-m)!\,(l+s)!\,4\pi}} \frac{1}{(m-s)!}. \tag{3.41}$$

However, examining equation 3.40 reveals the assumption that $r = 0$ pertains to the zeroth order term, is not a good assumption since this would only be true if $m \not> s$. When $m > s$ the second binomial coefficient in equation 3.40 is equal to zero: hence, the $r = 0$ term does not pertain to the zeroth order term in the series expansion. In this case all terms up to the term $r = m - s$ will be zero due to this binomial, and so $r = m - s$ pertains to the zeroth order term, *i.e.*

$$\lim_{x\to -1} {}_sy_{lm}(x) = (-1)^{[l-s+m]} \, 2^{[-k_+ - (s-m)/2]} \sqrt{\frac{(l+s)!\,(l-m)!\,(2l+1)}{(l-s)!\,(l+m)!\,4\pi}} \frac{1}{(s-m)!}. \tag{3.42}$$

Hence, in full for the sake of clarity

$$\lim_{x\to -1} {}_sy_{lm}(x) = \lim_{x\to -1} (1+x)^{-k_-}(1-x)^{-k_+} {}_sY_{lm}(x)$$
$$= \begin{cases} (-1)^l \, 2^{[-k_+ - (m-s)/2]} \sqrt{\frac{(l+m)!\,(l-s)!\,(2l+1)}{(l-m)!\,(l+s)!\,4\pi}} \frac{1}{(m-s)!}; & \text{if } m > s, \\ (-1)^{[l-s+m]} \, 2^{[-k_+ - (s-m)/2]} \sqrt{\frac{(l+s)!\,(l-m)!\,(2l+1)}{(l-s)!\,(l+m)!\,4\pi}} \frac{1}{(s-m)!}; & \text{otherwise.} \end{cases} \tag{3.43}$$

For the $x = +1$ boundary there is a similar situation, that is, a similar branching into two separate cases - depending on the values of the parameters - which is induced by the presence of the binomial coefficients

$$\lim_{x\to +1} {}_sy_{lm}(x) = \lim_{x\to 1} (1+x)^{-k_-}(1-x)^{-k_+} {}_sY_{lm}(x)$$
$$= \begin{cases} (-1)^m \, 2^{[-k_- - (s+m)/2]} \sqrt{\frac{(l+s)!\,(l+m)!\,(2l+1)}{(l-s)!\,(l-m)!\,4\pi}} \frac{1}{(s+m)!}; & \text{if } m > -s, \\ (-1)^s \, 2^{[-k_- - (s+m)/2]} \sqrt{\frac{(l-s)!\,(l-m)!\,(2l+1)}{(l+s)!\,(l+m)!\,4\pi}} \frac{1}{(-s-m)!}; & \text{otherwise.} \end{cases} \tag{3.44}$$

The value of this limit will also be used for the value of γ_r which was introduced in equation 3.20 and also appears in equations 3.23b and 3.27b.

3.5 Return to the original dependent variable

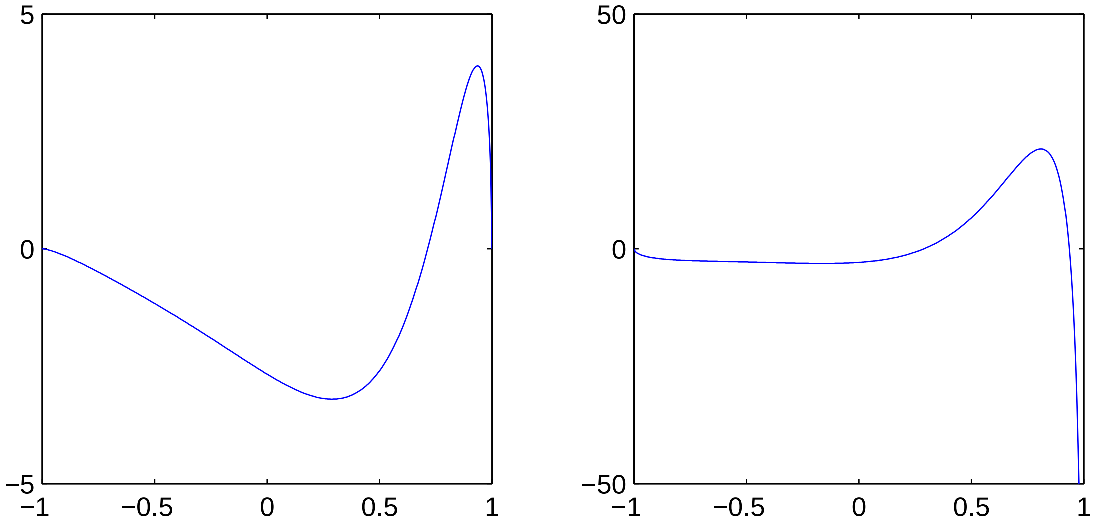
At this point, a matrix of partial derivatives has been developed and an initial guess has also been specified. Convergence parameters discussed in section 2.2 must also be chosen, however, these are relatively trivial. Hence, all of the necessary developments required to enable the application of the the relaxation method - as described in detail in section 2 - on the new dependent variable ${}_s z_{lmc}\left(x\right)$ have been made. This relaxation process should yield a solution in the dependent variable ${}_s z_{lmc}\left(x\right)$ as well as the derivative of this function ${}_s z'_{lmc}\left(x\right)$. As such, an inverse transformation to return to the original dependent variable ${}_s S_{lmc}\left(x\right)$ for both ${}_s z_{lmc}\left(x\right)$ and ${}_s z'_{lmc}\left(x\right)$. However, trivially

$$ {}_s S_{lmc}\left(x\right) = \left(1+x\right)^{k_-}\left(1-x\right)^{k_+}\,{}_s z_{lmc}\left(x\right), \tag{3.45}$$

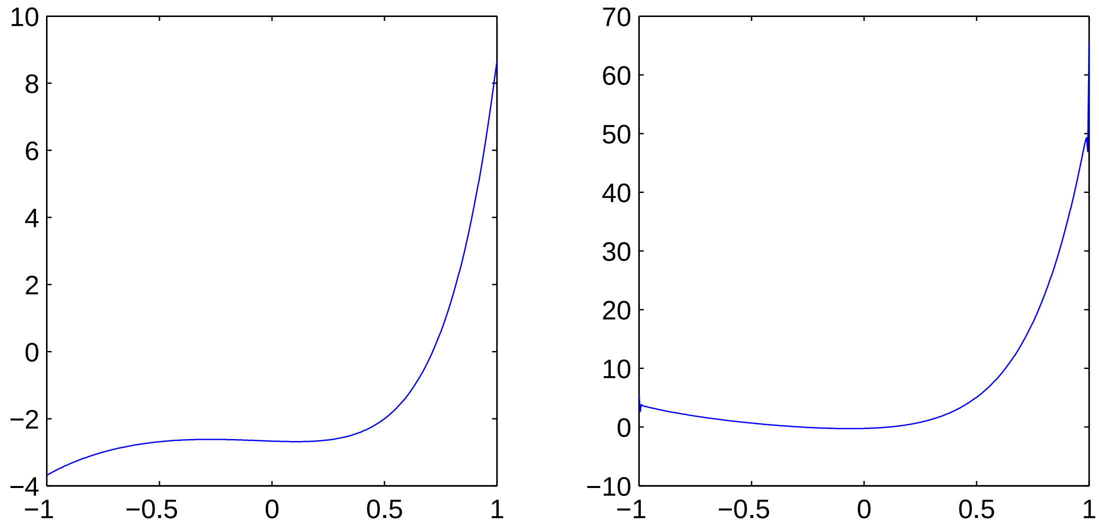
while

$$\frac{\mathrm{d}_s S_{lm}}{\mathrm{d}x} = \left(1-x\right)^{-1-k_+}\left(1+x\right)^{-1-k_-}\Bigg\{\left(1-x\right)^{k_+}\left(1+x\right)^{k_-} \\ \Big[k_+\left(1+x\right) - k_-\left(1-x\right)\Big]\,{}_s Y_{lm}\left(x\right) + \left(1-x^2\right)\Bigg\}\frac{\mathrm{d}_s z_{lm}}{\mathrm{d}x}. \tag{3.46}$$

And so the spin-weighed spheroidal harmonics ${}_s S_{lmc}\left(x\right)$ can be recovered from the end result of the relaxation process in the dependent variable ${}_s z_{lmc}\left(x\right)$. An example of the difference between the two dependent variables can seen in Figure 3.2.



(a)



(b)

Figure 3.2: A comparison between the solutions the spin-weighted spheroidal differential equation in the different dependent variables: ${}_sS_{lmc}\,(x)$ (left) and ${}_sS'_{lmc}\,(x)$ (right) in Figure 3.2a; and ${}_sz_{lmc}\,(x)$ (left) and ${}_sz'_{lmc}\,(x)$ (right) in Figure 3.2b, for $s = -1$,$l = 3$, $m = 2$ and $c = 4$.

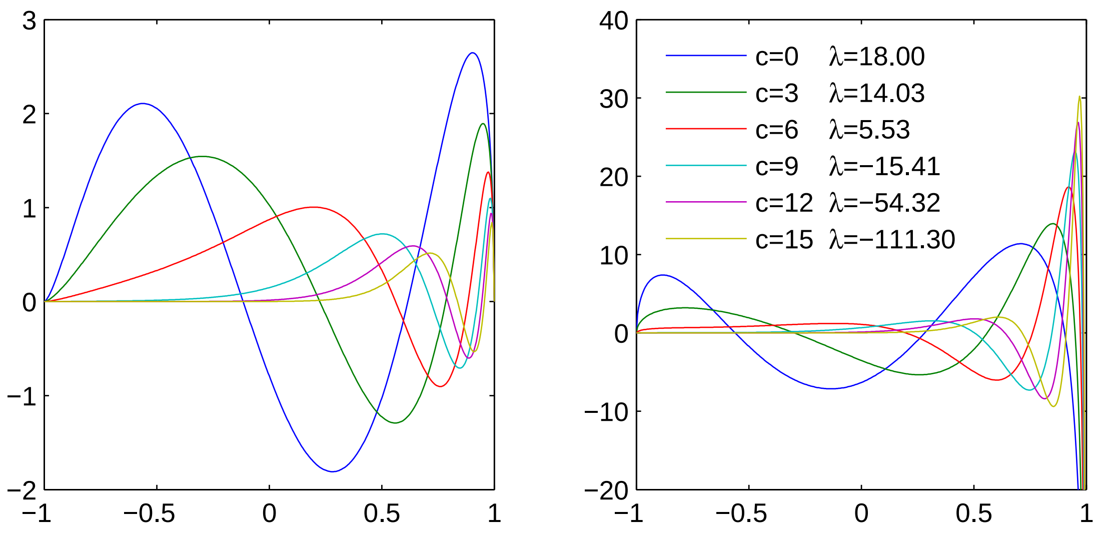


Figure 4.1: The solution the spin-weighted spheroidal differential equation ${}_sS_{lmc}\,(x)$ (left hand plot) and ${}_sS'_{lmc}\,(x)$ (right hand plot) for $s = -2$, $l = 4$ and $m = 1$ for various values of c. Both of these plots were generated from the output of the C-code presented in the appendix, *i.e.* gives the derivative for free.

4 Conclusions

4.1 Results

The method described above was coded up in C and can be found in the appendix. In its current form the code can cannot evaluate the eigensystem for general complex values of c but rather only pure real or, through a minor change, (flip four signs in the code) pure imaginary values. However, the way it has been coded means that altering it to take general complex c arguments should be trivial: by changing all of the relevant real *GSL* data types to their complex equivalent *should* allow the code to take complex c arguments. There is a caveat however: changing to *GSL* data types also entails changing all of the relevant basic C operators - *i.e.* "+", "−", "∗", "/" *etc.* - to the corresponding *GSL* operator for complex data types and this trivial change is, in practice, rather fiddly. This could be done by hand or perhaps with the assistance of a regular expression find and replace tool of some kind, *e.g.* [17].

The typical number of iterations required for the multi-dimensional Newton method to converge is highly dependent on the selected values of the parameters s, l, m and c. In particular non-zero s and large c can fail to converge in a reasonable amount of time. The code is fast: given most values small of c, say $c = 10$, the code will relax to the solution in approximately 1 second. Large values of c will take longer. The memory footprint of the program is negligible: *circa* 0.5 megabytes for a grid of about 500 points. Figure 4.1 shows an example of the output of code.

This output can then easily be combined with the azimuthal solution - *i.e.* multiplied by $e^{im\phi}$ - to form the harmonics. Figures 4.2-4.7 contain plots of the absolute value of the, generally complex, harmonics for various values of the parameters, *i.e.* $|{}_sS_{lmc}(\theta, \phi)|$. In particular, for each of the figures the value of l starts at the top where $l = 1$ and increases by 1 on each lower tier. The value of m starts at left hand side where $m = 0$ and increases by 1 in each column. This leads to the triangular layout of the plots since $|m| \leq l$ (only positive values of m are plotted). The value of s and c will be specified individually for each figure. The colour red corresponds to the positive values of the harmonics while the colour blue corresponds to the negative values of the harmonics.

4.2 Future work

The numerical method could be improved in several ways. First consider the allocation of the grid points on the domain, *i.e.* equation 3.24. This allocation could be improved by assigning the grid uniformly in θ rather than uniformly in x. This would improve accuracy of the method near the poles (see Figure 4.5). Better yet, the assignment could be done dynamically - that is, use an adaptive mesh that changes with each iteration in order to capture the most important section of the solutions most accurately . Indeed, *Numerical Recipes* [14, pp. 981, 982] contains as scheme for doing just this that would be suitable for this problem.

An adaptive integration method for the size of the discrete steps in the complex c plane would also be a sensible improvement. For some values of the parameters, a very large jump in c could be taken safely in one go while for others even sensibly small steps might prove too large. Hence, an adaptive integration method in the complex c plane would make the method both more robust and considerably faster.

The code has only been written for integer values of s. An extension to half integer s values would allow the algorithm to calculate spin-weighed spheroidal harmonics for fermionic field perturbations.

Of course, a very obvious change to make would be to improve the ABD matrix solve. No attempt was made to find the best of fastest ABD solver currently available. Given that the ABD matrix solver is where the bulk of the necessary computations are performed, any improvement here will drastically improve the codes speed. Improved methods - including parallel methods - can be found in [15].

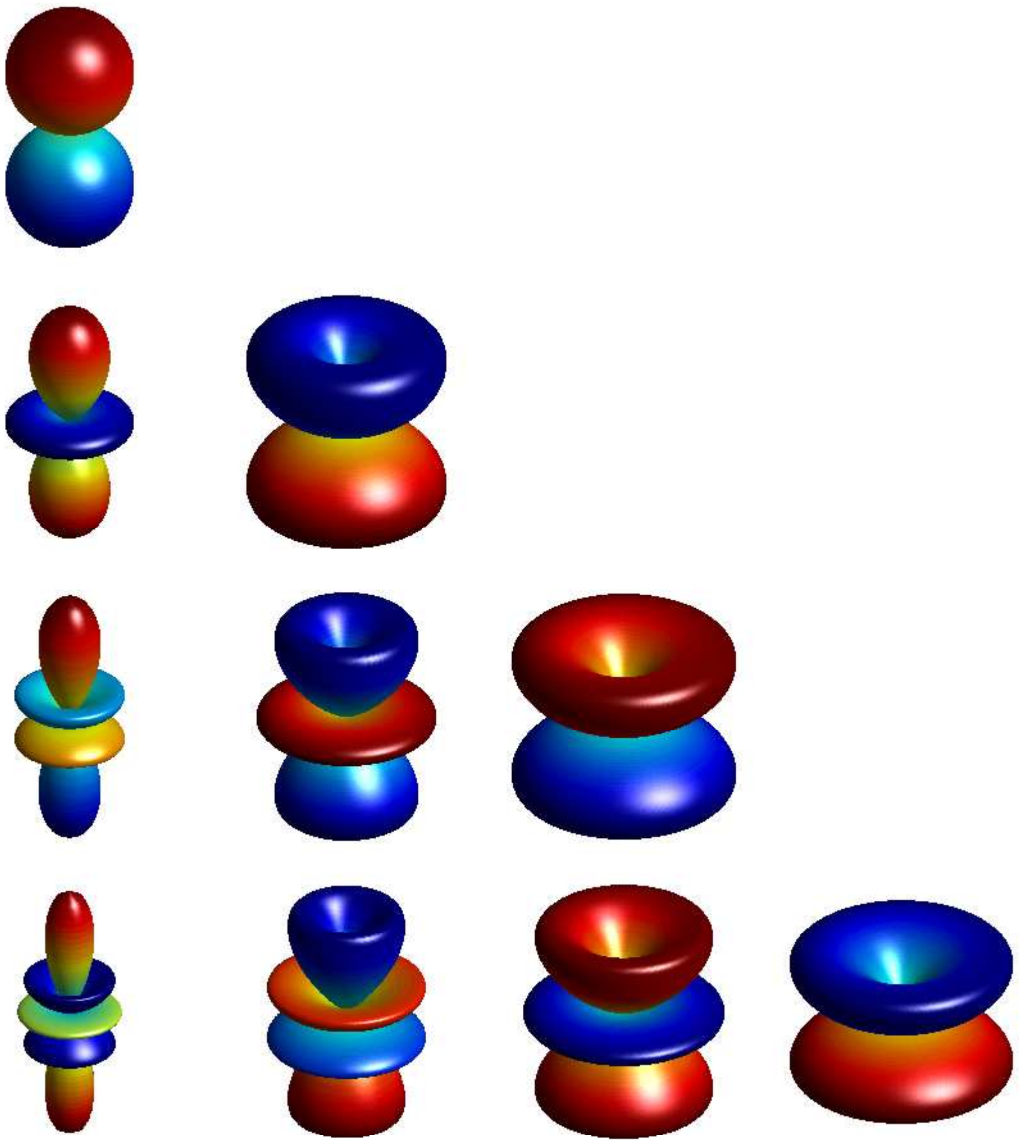


Figure 4.2: Spherical harmonics $|Y_{lm}(\theta, \phi)|$, *i.e.* the spin-weighted spheroidal harmonics with $s = 0$ and $c = 0$.

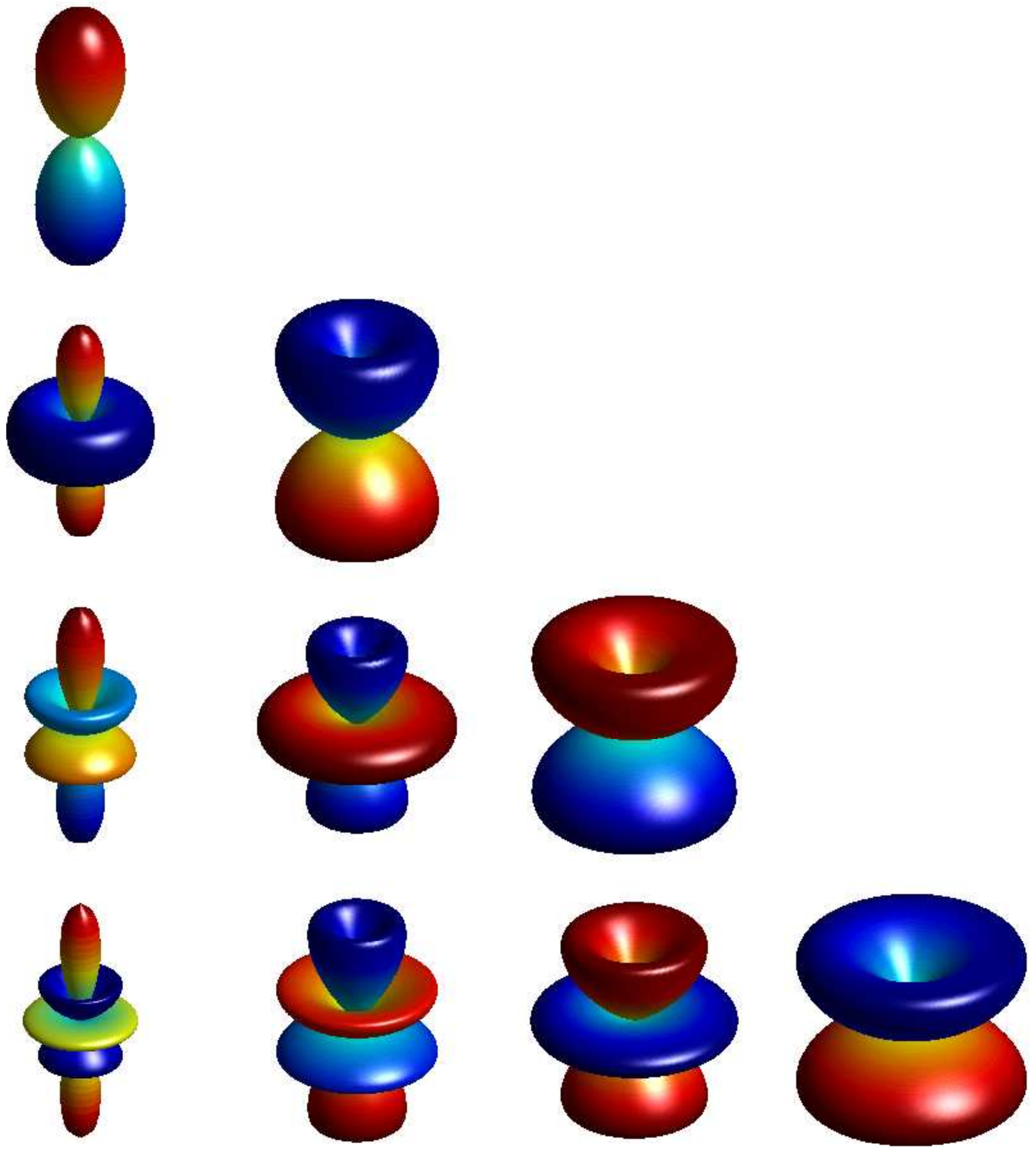


Figure 4.3: Spheroidal harmonics $|S_{lmc}(\theta,\phi)|$ with $c=4$, *i.e.* the spin-weighted spheroidal harmonics with $s=0$.

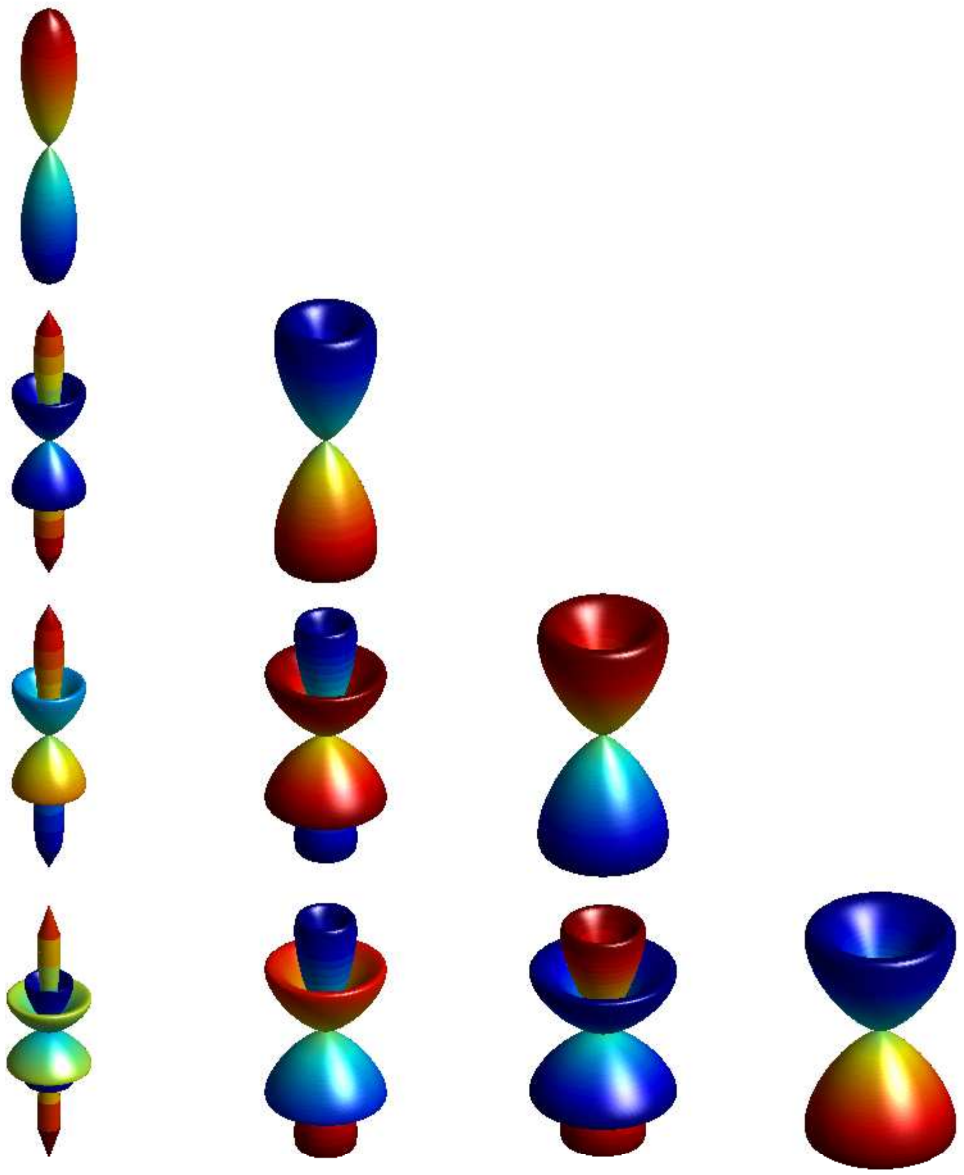


Figure 4.4: Spheroidal harmonics $|S_{lmc}(\theta,\phi)|$ with $c = 12$, *i.e.* the spin-weighted spheroidal harmonics with $s = 0$.

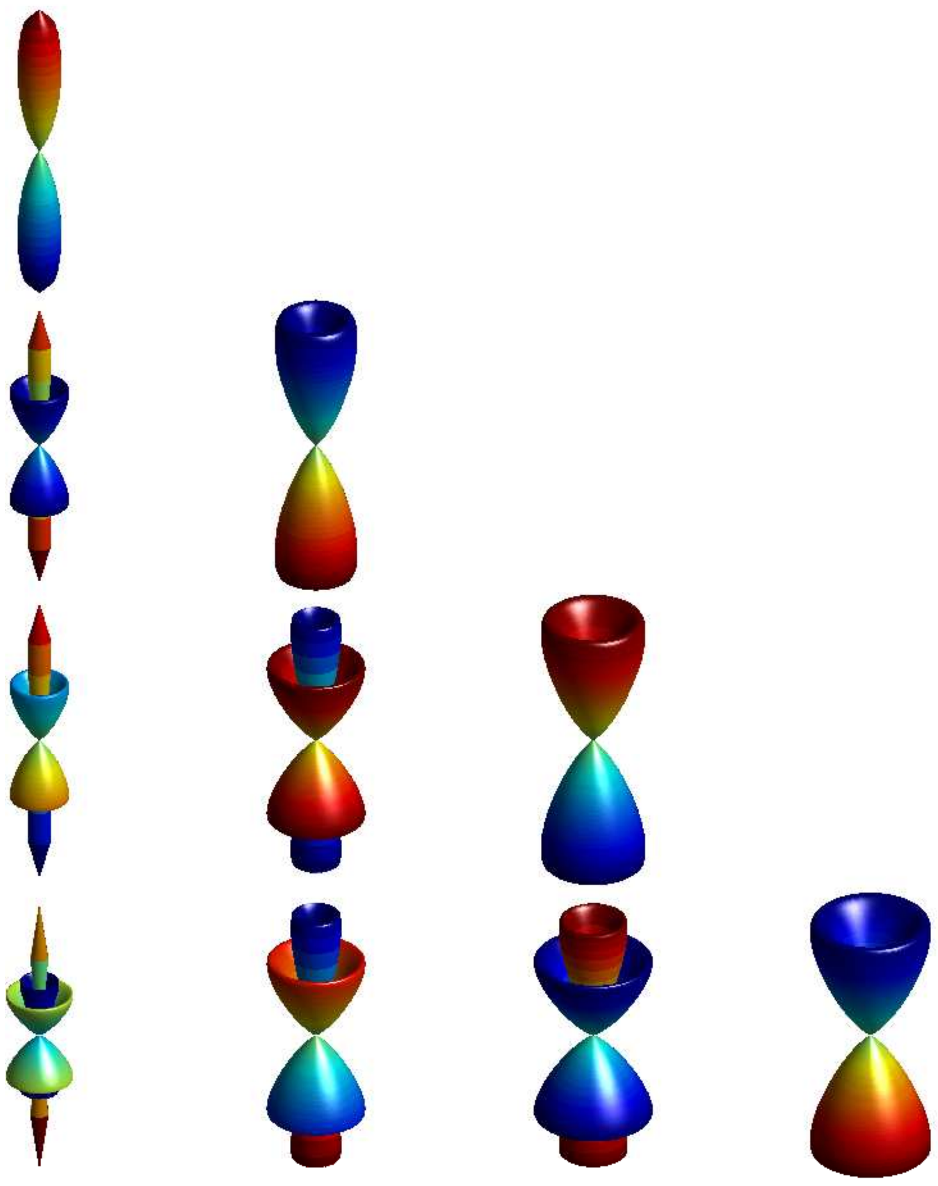


Figure 4.5: Spheroidal harmonics $|S_{lmc}(\theta,\phi)|$ with $c = 20$, *i.e.* the spin-weighted spheroidal harmonics with $s = 0$. Numerical artefacts are clearly present here in the $m = 0$ harmonics, particularly at the poles. Using a numerical grid uniform in θ rather than x would significantly reduces this since the x uniform grid is less dense near the poles than at the equator.

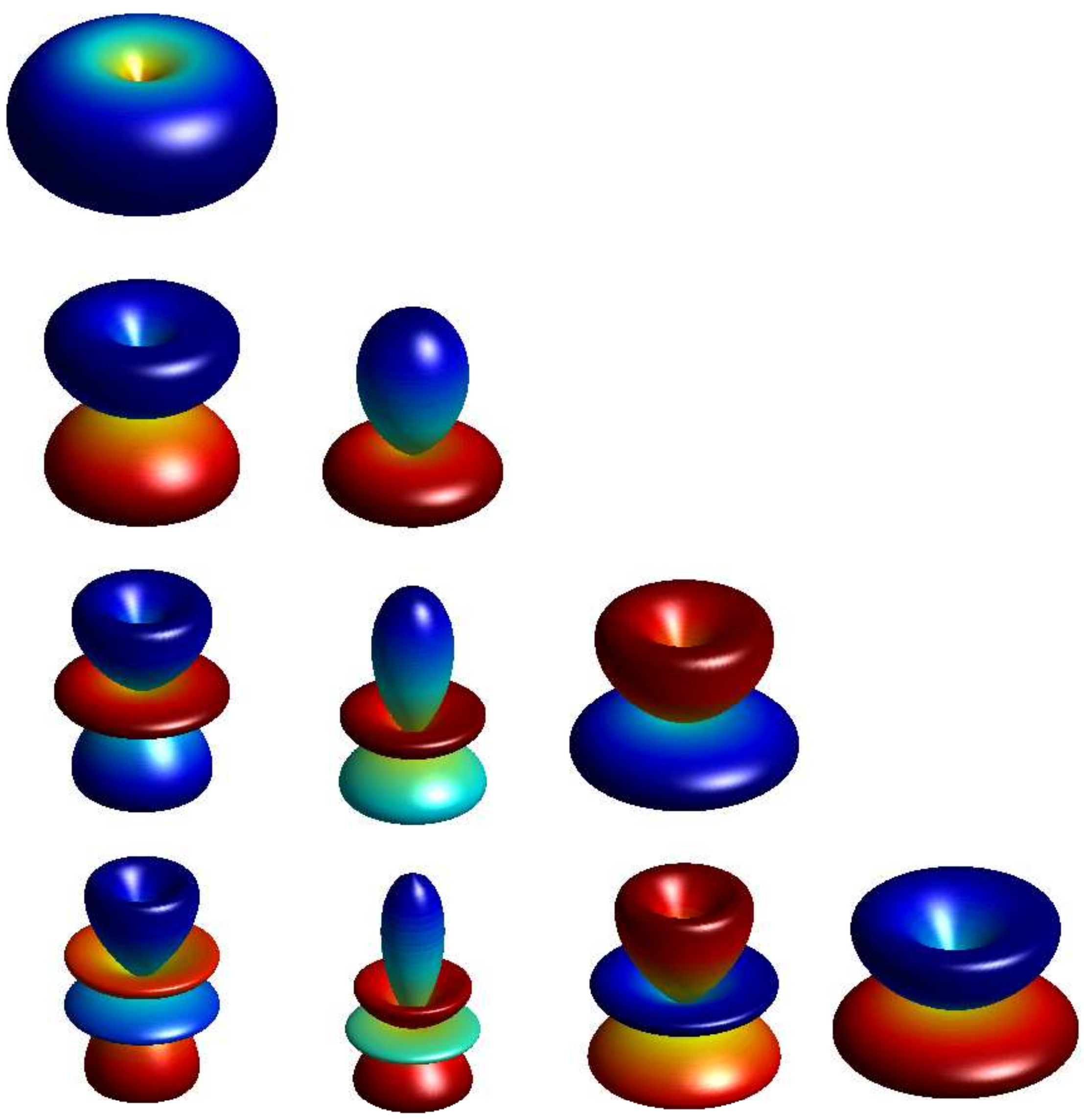


Figure 4.6: Spin-weighted spherical harmonics $|{}_sY_{lm}(\theta,\phi)|$ with $s=-1$, *i.e.* the spin-weighted spheroidal harmonics with $c=0$.

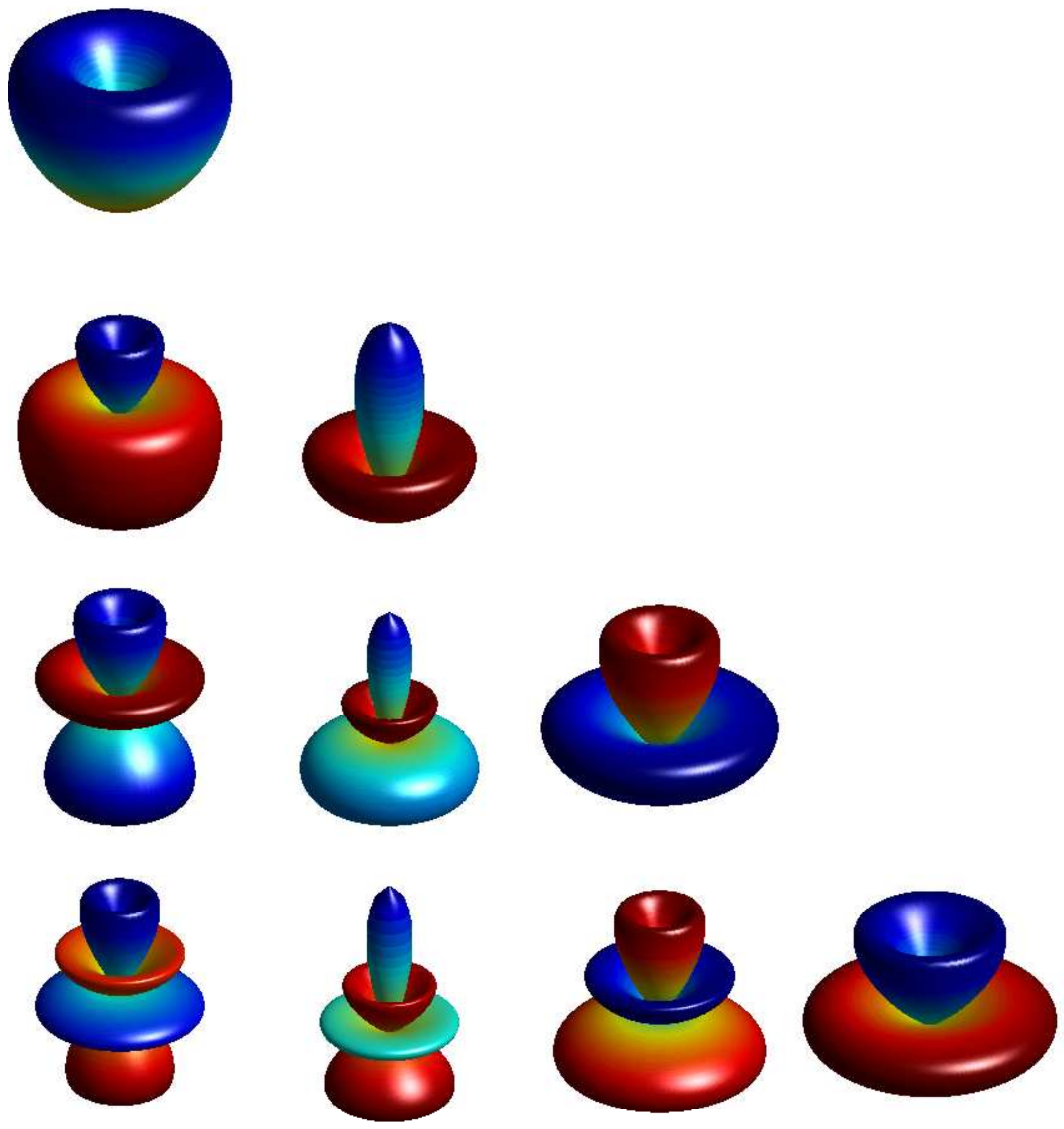


Figure 4.7: Spin-weighted spheroidal harmonics $|{}_sS_{lmc}(\theta,\phi)|$ with $s=-1$ and $c=4$.

Appendix

Preface to the C-code

Consider equation 3.11

$$_sS_{lmc}(x) = (1+x)^{k_-}(1-x)^{k_+}\,{}_sz_{lmc}(x).$$

If this series solution is inserted into the spin-weighted spheroidal differential equation 1.2 in a computation algebra package like say *Mathematica*, then equation 3.12 is not the result that immediately follows, rather

$$\Bigg\{\left(-1+x^2\right)^2\frac{\mathrm{d}^2}{\mathrm{d}x^2}+2\left(-1+x^2\right)\left(x-k_-(1-x)+k_+(1+x)\right)\frac{\mathrm{d}}{\mathrm{d}x}$$
$$\Big[-Ax^2+A-c^2x^4+c^2x^2+2csx^3-2csx+k_-^2(x-1)^2+k_-(2k_++1)\left(x^2-1\right)$$
$$+k_+^2+k_+x^2-k_+-m^2-2msx-s^2x^2-sx^2+s\Big]\Bigg\}\,{}_sz_{lmc}(x)=0,$$

is obtained. Now, this can be expressed equivalently as

$$\Bigg\{\left(1-x^2\right)\frac{\mathrm{d}^2}{\mathrm{d}x^2}-2\left[k_+-k_-+x\left(1+k_++k_-\right)\right]\frac{\mathrm{d}}{\mathrm{d}x}$$
$$+\left[{}_sA_{lm}+s(s+1)+c^2x^2-2csx-(k_++k_-)(k_++k_-+1)\right]\Bigg\}\,{}_sz_{lmc}(x)=0,$$

where the insertion $0=m^2+s^2-2k_+^2-2k_-^2$ has been used to reduced the ODE to this succinct form. However, this tidy form was not found until near the end of this thesis. As such, it is not this tidy form that has been used in the C-code but rather the top expression. This is a trivial distinction as the corresponding calculations occur automatically in the C-code.

In this form equation 3.25b becomes

$$E_{1,k}=y_{1,k}-y_{1,k-1}-\frac{h}{\left(1-x_\kappa^2\right)^2}\Bigg\{2\left[k_+-k_-+x_\kappa\left(1+k_++k_-\right)\left(1-x_\kappa^2\right)\right]y_{1,\kappa}$$
$$+\Big[k_++m^2-s+2csx+2msx+x^2\left(-c^2-k_++s(1+s)-2csx+c^2x^2\right)$$
$$-k_+^2(1+x)^2-k_-^2(1-x)^2+k_-(1+2k_+)\left(1-x^2\right)-y_{2,\kappa}+x^2y_{2,\kappa}\Big]y_{0,\kappa}\Bigg\}.$$

And equations 3.28b become

$$S_{1,0} = -\frac{h}{2\left(1-x_\kappa^2\right)^2}\Big[k_+ + m^2 - s - k_-^2\left(1-x\right)^2 + 2csx + 2msx - k_+^2\left(1+x\right)^2$$
$$+ k_-\left(1+2k_+\right)\left(1-x^2\right) + x^2\left(-c^2 - k_+ + s\left(1+s\right) - 2csx + c^2x^2\right) - y_{2,\kappa} + x^2 y_{2,\kappa}\Big],$$
$$S_{1,1} = -1 - \frac{h}{\left(1-x_\kappa^2\right)}\Big[k_+ - k_- + x_\kappa\left(1+k_+ + k_-\right)\Big], \qquad S_{1,2} = \frac{h}{2\left(1-x_\kappa^2\right)}y_{0,\kappa},$$
$$S_{1,3} = S_{1,0}, \qquad S_{1,4} = 2 + S_{1,1}, \qquad S_{1,5} = S_{1,2}.$$

And again, it was in this form that equations were coded up, as opposed to the form equations presented in the thesis.

C-code

SWSH.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include <windows.h>

#include <gsl/gsl_errno.h>
#include <gsl/gsl_vector.h>
#include <gsl/gsl_matrix.h>
#include <gsl/gsl_sf_legendre.h>

#define NE 3
#define M 401
#define NB 1
#define NSI NE
#define NYJ NE
#define NYK M
#define NCI NE
#define NCJ (NE-NB+1)
#define NCK (M+1)
#define NSJ (2*NE+1)

int mm = 0, ll = 1, ss = 0;
long double h = 0.0, cc = 0.0, anorm = 0.0, kn = 0.0, kp = 0.0, tar = 0.0;
long double anorml = 0.0, anormr = 0.0;
gsl_vector_long_double *x;


/* ====================================================================
*  Custom Rank 3 Array... not supported by default GSL
*  ==================================================================== */

long double cf_gsl_3dmatrix_get
  (gsl_matrix_long_double **ptr_vec,int i, int j, int k)
{
  return gsl_matrix_long_double_get(ptr_vec[i],j,k);
}

void cf_gsl_3dmatrix_set
  (gsl_matrix_long_double **ptr_vec,int i, int j, int k, long double x)
{
  gsl_matrix_long_double_set(ptr_vec[i],j,k,x);
}

gsl_matrix_long_double **cf_gsl_3dmatrix_alloc
  (int num_i, int num_j, int num_k)
{
  int p;
  gsl_matrix_long_double **ptr_vec
    = malloc((size_t) num_i*sizeof(gsl_matrix_long_double)-1);
  for (p=1;p<=num_i;p++)
    ptr_vec[p] = gsl_matrix_long_double_alloc(num_j,num_k);
  return ptr_vec;
}

void cf_gsl_3dmatrix_free
  (gsl_matrix_long_double **m, int num_i)
{
  int p;
  for (p=1;p<=num_i;p++)
    gsl_matrix_long_double_free(m[p]);
  free(m+1);
}

/* ==================================================================== */


/* ====================================================================
*  Functions for Block Diagonal Solver
*  ==================================================================== */

int pinvs
  (int ie1, int ie2, int je1, int jsf, int jc1, int  k,
   gsl_matrix_long_double **c, gsl_matrix_long_double *s)
{
  int js1, jpiv, jp, je2, jcoff ,j ,irow ,ipiv ,id ,icoff ,i;
  long double pivinv, piv, dum, big;
  gsl_vector_int *indxr;
  gsl_vector_long_double *pscl;

  indxr = gsl_vector_int_alloc(ie2-ie1+1);
  pscl  = gsl_vector_long_double_alloc(ie2-ie1+1);;

  je2 = je1 + ie2 - ie1;
  js1 = je2 + 1;

  for (i=ie1;i<=ie2;i++)
  {
    big = 0.0;
    for (j=je1;j<=je2;j++)
      if (fabsl(gsl_matrix_long_double_get (s, i-1, j-1)) > big)
        big = fabsl(gsl_matrix_long_double_get(s, i-1, j-1));
    if (big == 0.0)  {
      GSL_ERROR("Singular matrix - row all 0, in pinvs",GSL_ESING);
      }
    gsl_vector_long_double_set(pscl,i-ie1,1.0/big);
    gsl_vector_int_set(indxr,i-ie1,0);
  }
  for (id=ie1;id<=ie2;id++)
  {
    piv = 0.0;
    for (i=ie1;i<=ie2;i++)
    {
      if (gsl_vector_int_get(indxr,i-ie1) == 0)
      {
        big = 0.0;
        for (j=je1;j<=je2;j++)
        {
          if (fabsl(gsl_matrix_long_double_get(s, i-1, j-1)) > big)
          {
            jp  = j;
            big = fabsl(gsl_matrix_long_double_get(s, i-1, j-1));
          }
        }
        if (big*gsl_vector_long_double_get(pscl,i-ie1) > piv)
        {
          ipiv = i;
          jpiv = jp;
          piv  = big*gsl_vector_long_double_get(pscl,i-ie1);
        }
      }
    }
    if (gsl_matrix_long_double_get(s, ipiv-1, jpiv-1) == 0.0)
      GSL_ERROR("Singular matrix  in  routine pinvs",GSL_ESING);
    gsl_vector_int_set(indxr,ipiv-ie1,jpiv);
    pivinv = 1.0/gsl_matrix_long_double_get(s, ipiv-1, jpiv-1);
    for (j=je1;j<=jsf;j++)
      gsl_matrix_long_double_set(s, ipiv-1, j-1,
        gsl_matrix_long_double_get(s, ipiv-1, j-1)*pivinv );
    gsl_matrix_long_double_set(s, ipiv-1, jpiv-1, 1.0 );
    for (i=ie1;i<=ie2;i++)
    {
      if (gsl_vector_int_get(indxr,i-ie1) !=  jpiv)
      {
        if (gsl_matrix_long_double_get(s, i-1, jpiv-1))
        {
          dum = gsl_matrix_long_double_get(s, i-1, jpiv-1);
          for (j=je1;j<=jsf;j++)
            gsl_matrix_long_double_set(s, i-1, j-1,
              gsl_matrix_long_double_get(s,i-1,j-1)
                -dum*gsl_matrix_long_double_get(s,ipiv-1,j-1) );
          gsl_matrix_long_double_set(s, i-1, jpiv-1, 0.0 );
        }
      }
    }
  }
  jcoff = jc1 - js1;
  icoff = ie1 - je1;
  for  (i=ie1;i<=ie2;i++)
  {
    irow = gsl_vector_int_get(indxr,i-ie1) + icoff;
    for  (j=js1;j<=jsf;j++)
      cf_gsl_3dmatrix_set(c,irow,j+jcoff,k,
        gsl_matrix_long_double_get(s, i-1, j-1));
  }
  gsl_vector_long_double_free(pscl);
  gsl_vector_int_free(indxr);
  return GSL_SUCCESS;
}

void red
  (int iz1, int iz2, int jz1, int jz2, int jm1, int jm2, int  jmf,
   int ic1, int jc1, int jcf, int kc, gsl_matrix_long_double **c,
   gsl_matrix_long_double *s)
{
  int loff, l, j, ic, i;
  long double vx;
  loff = jc1 - jm1;
  ic   = ic1;
  for (j=jz1;j<=jz2;j++)
  {
    for (l=jm1;l<=jm2;l++)
    {
      vx = cf_gsl_3dmatrix_get(c,ic,l+loff,kc);
      for (i=iz1;i<=iz2;i++)
        gsl_matrix_long_double_set(s, i-1, l-1,
          gsl_matrix_long_double_get(s,i-1,l-1)
            -vx*gsl_matrix_long_double_get(s,i-1,j-1) );
    }
    vx = cf_gsl_3dmatrix_get(c,ic,jcf,kc);
    for  (i=iz1;i<=iz2;i++)
      gsl_matrix_long_double_set(s, i-1, jmf-1,
        gsl_matrix_long_double_get(s,i-1,jmf-1)
          -vx*gsl_matrix_long_double_get(s,i-1,j-1) );
    ic += 1;
  }
}

void bksub
  (int ne, int nb, int jf, int k1, int k2, gsl_matrix_long_double **c)
{
  int nbf, im, kp, k, j, i;
  long double xx;
  nbf = ne - nb;
  im  = 1;
  for (k=k2;k>=k1;k--)
  {
    if (k == k1)
      im = nbf + 1;
    kp = k + 1;
    for (j=1;j<=nbf;j++)
    {
      xx = cf_gsl_3dmatrix_get(c,j,jf,kp);
      //printf("\nxx = %Lf  ", xx);
      for (i=im;i<=ne;i++)
        cf_gsl_3dmatrix_set(c,i,jf,k,
          cf_gsl_3dmatrix_get(c,i,jf,k)
            -xx*cf_gsl_3dmatrix_get(c,i,j,k));
    }
  }
  for (k=k1;k<=k2;k++)
  {
    kp = k + 1;
    for (i=1;i<=nb;i++)
      cf_gsl_3dmatrix_set(c,i,1,k,cf_gsl_3dmatrix_get(c,i+nbf,jf,k));
    for (i=1;i<=nbf;i++)
      cf_gsl_3dmatrix_set(c,i+nb,1,k,cf_gsl_3dmatrix_get(c,i,jf,kp));
  }
}

/* ===================================================================== */


/* =====================================================================
*  Matrix structure for block diagonal solver. s[i][j] valuea are the
*  typically non-zero entries in the matrix.
*  ===================================================================== */

void difeq
  (int k, int k1, int k2, int jsf, int is1, int isf, int indexv[],
   int ne, gsl_matrix_long_double *s, gsl_matrix_long_double *y)
{
  long double tempx, tempy0, tempy1, tempy2, alpha,beta,gamma;
  if (k == k1)
  {
    /* x(-1) boundary */
    gsl_matrix_long_double_set(s,2,2+indexv[1],
      -(
        (
          -4.0*gsl_matrix_long_double_get(y,2,0)-4.0*powl(cc,2.0)+4.0*kn
          +4.0*powl(kn,2.0)+mm*mm-4.0*ss-8.0*cc*ss+2.0*mm*ss-3.0*ss*ss
        )
          /(
            16.0*kn+8.0*kn*kn-2.0*(-4.0+mm*mm-2.0*mm*ss+ss*ss)
          )
        -kp/2.0
      )
    );
    gsl_matrix_long_double_set(s,2,2+indexv[2],
      1.0
    );
    gsl_matrix_long_double_set(s,2,2+indexv[3],
      4.0*gsl_matrix_long_double_get(y,0,0)
        /(16.0*kn+8.0*kn*kn-2.0*(-4.0+mm*mm-2.0*mm*ss+ss*ss))
    );
    gsl_matrix_long_double_set(s,2,jsf-1,
      gsl_matrix_long_double_get(y,1,0)
      -gsl_matrix_long_double_get(y,0,0)
      *(
        (
          -4.0*gsl_matrix_long_double_get(y,2,0)-4.0*powl(cc,2.0)+4.0*kn
          +4.0*powl(kn,2.0)+mm*mm-4.0*ss-8.0*cc*ss+2.0*mm*ss-3.0*ss*ss
        )
          /(
            16.0*kn+8.0*kn*kn-2.0*(-4.0+mm*mm-2.0*mm*ss+ss*ss)
          )
        -kp/2.0
      )
    );
  }
  else if (k > k2)
  {
    /* x(+1) boundary */
    gsl_matrix_long_double_set(s,0,2+indexv[1],
      -(
        (
          4.0*gsl_matrix_long_double_get(y,2,M-1)+4.0*powl(cc,2.0)-4.0*kp
          -4.0*powl(kp,2.0)-mm*mm+4.0*ss-8.0*cc*ss+2.0*mm*ss
          +3.0*powl(ss,2.0)
        )
          /(
            16.0*kp+8.0*powl(kp,2.0)-2.0*(-4.0+mm*mm+2.0*mm*ss
            +powl(ss,2.0))
          )
        +kn/2.0
      )
    );
    gsl_matrix_long_double_set(s,0,2+indexv[2],
      1.0
    );
    gsl_matrix_long_double_set(s,0,2+indexv[3],
      -4.0*gsl_matrix_long_double_get(y,0,M-1)
        /(16.0*kp+8.0*kp*kp-2.0*(-4.0+mm*mm+2.0*mm*ss+ss*ss))
    );
    gsl_matrix_long_double_set(s,0,jsf-1,
      gsl_matrix_long_double_get(y,1,M-1)
      -gsl_matrix_long_double_get(y,0,M-1)
      *(
        (
          4.0*gsl_matrix_long_double_get(y,2,M-1)+4.0*powl(cc,2.0)-4.0*kp
          -4.0*powl(kp,2.0)-mm*mm+4.0*ss-8.0*cc*ss+2.0*mm*ss
          +3.0*powl(ss,2.0)
        )
          /(
            16.0*kp+8.0*powl(kp,2.0)-2.0*(-4.0+mm*mm+2.0*mm*ss
            +powl(ss,2.0))
          )
        +kn/2.0
      )
    );
    gsl_matrix_long_double_set(s,1,2+indexv[1],
      1.0 );
    gsl_matrix_long_double_set(s,1,2+indexv[2],
      0.0 );
    gsl_matrix_long_double_set(s,1,2+indexv[3],
      0.0 );
    gsl_matrix_long_double_set(s,1,jsf-1,
      gsl_matrix_long_double_get(y,0,M-1)-anormr );
  }
  else
  {
    /* Internal points */
    gsl_matrix_long_double_set(s, 0, indexv[1]-1,
      -1.0 );
    gsl_matrix_long_double_set(s, 0, indexv[2]-1,
      -0.5*h );
    gsl_matrix_long_double_set(s, 0, indexv[3]-1,
      0.0 );
    gsl_matrix_long_double_set(s, 0, 2+indexv[1],
      1.0 );
    gsl_matrix_long_double_set(s, 0, 2+indexv[2],
      -0.5*h );
    gsl_matrix_long_double_set(s, 0, 2+indexv[3],
      0.0 );
    tempx = (gsl_vector_long_double_get(x,k-1)
      +gsl_vector_long_double_get(x,k-2))/2.0;
    tempy0 = (gsl_matrix_long_double_get(y,0,k-1)
      +gsl_matrix_long_double_get(y,0,k-2))/2.0;
    tempy1 = (gsl_matrix_long_double_get(y,1,k-1)
      +gsl_matrix_long_double_get(y,1,k-2))/2.0;
    tempy2 = (gsl_matrix_long_double_get(y,2,k-1)
      +gsl_matrix_long_double_get(y,2,k-2))/2.0;

    gsl_matrix_long_double_set(s, 1, indexv[1]-1,
      -h
      *(
        (1.0/(2.0*powl(1.0-tempx*tempx,2.0)))
        *(
          kp+mm*mm-ss-kn*kn*powl(1.0-tempx,2.0)
          +2.0*cc*ss*tempx
          +2.0*mm*ss*tempx
          -kp*kp*powl(1+tempx,2.0)
          +kn*(1.0+2.0*kp)*(1.0-tempx*tempx)
          +tempx*tempx
            *(
              -powl(cc,2.0)-kp+ss*(1.0+ss)
              -2.0*cc*ss*tempx+powl(cc,2.0)*tempx*tempx
            )
          -tempy2
          +tempx*tempx*tempy2
        )
      )
    );
    gsl_matrix_long_double_set(s,1,indexv[2]-1,
      -1.0-(h*(kp-kn*(1.0-tempx)+tempx+kn*tempx))/(1.0-tempx*tempx)
    );
    gsl_matrix_long_double_set(s,1,indexv[3]-1,
      (h*(1.0-tempx*tempx)*tempy0)/(2.0*powl(1.0-tempx*tempx,2.0))
    );
    gsl_matrix_long_double_set(s,1,2+indexv[1],
      gsl_matrix_long_double_get(s,1,indexv[1]-1) );
    gsl_matrix_long_double_set(s,1,2+indexv[2],
      2.0+gsl_matrix_long_double_get(s,1,indexv[2]-1) );
    gsl_matrix_long_double_set(s,1,2+indexv[3],
      gsl_matrix_long_double_get(s,1,indexv[3]-1) );
    gsl_matrix_long_double_set(s,2,indexv[1]-1,
      0.0 );
    gsl_matrix_long_double_set(s,2,indexv[2]-1,
      0.0 );
    gsl_matrix_long_double_set(s,2,indexv[3]-1,
      -1.0 );
    gsl_matrix_long_double_set(s,2,2+indexv[1],
      0.0 );
    gsl_matrix_long_double_set(s,2,2+indexv[2],
      0.0 );
    gsl_matrix_long_double_set(s,2,2+indexv[3],
      1.0 );
    gsl_matrix_long_double_set(s,0,jsf-1,
      gsl_matrix_long_double_get(y,0,k-1)
        -gsl_matrix_long_double_get(y,0,k-2)
        -0.5*h*(gsl_matrix_long_double_get(y,1,k-1)
        +gsl_matrix_long_double_get(y,1,k-2)) );
    gsl_matrix_long_double_set(s,1,jsf-1,
      gsl_matrix_long_double_get(y,1,k-1)
      -gsl_matrix_long_double_get(y,1,k-2)
      -h
      *(
        (1.0/powl(1.0-tempx*tempx,2.0))
        *(
          2.0*(kp-kn*(1.0-tempx)+tempx+kp*tempx)*(1.0-tempx*tempx)*tempy1
          +tempy0
          *(
            kp+mm*mm-ss-kn*kn*powl(1.0-tempx,2.0)
            +2.0*cc*ss*tempx
            +2.0*mm*ss*tempx
            +tempx*tempx
              *(
                -powl(cc,2.0)-kp+ss*(1.0+ss)
                -2.0*cc*ss*tempx+powl(cc,2.0)*tempx*tempx
              )
            -kp*kp*powl(1.0+tempx,2.0)
            +kn*(1.0+2.0*kp)*(1.0-tempx*tempx)
            -tempy2
            +tempx*tempx*tempy2
          )
        )
      )
    );
    gsl_matrix_long_double_set(s,2,jsf-1,
      gsl_matrix_long_double_get(y,2,k-1)
        -gsl_matrix_long_double_get(y,2,k-2) );
  }
}

/* ===================================================================== */


/* =====================================================================
*   Block Diagonal Solver
*   ===================================================================== */

int solvde
  (int itmax, long double conv, long double slowc, long double scalv[],
   int indexv[], int ne, int nb, int m, gsl_matrix_long_double *y,
   gsl_matrix_long_double **c, gsl_matrix_long_double *s, FILE *fp)
{
  int ic1, ic2, ic3, ic4, it, j, j1, j2, j3, j4, j5, j6, j7, j8, j9;
  int jc1, jcf, jv, k, k1, k2, km, kp,nvars;
  long double err, errj, fac, vmax, vz;
  gsl_vector_long_double *ermax_gsl;
  gsl_vector_int *kmax_gsl;

  kmax_gsl = gsl_vector_int_alloc(ne);
  ermax_gsl = gsl_vector_long_double_alloc(ne);

  k1    = 1;
  k2    = m;
  nvars = ne*m;
  j1    = 1;
  j2    = nb;
  j3    = nb + 1;
  j4    = ne;
  j5    = j4 + j1;
  j6    = j4 + j2;
  j7    = j4 + j3;
  j8    = j4 + j4;
  j9    = j8 + j1;
  ic1   = 1;
  ic2   = ne - nb;
  ic3   = ic2+1;
  ic4   = ne;
  jc1   = 1;
  jcf   = ic3;
  for (it=1;it<=itmax;it++)
  {
    k = k1;
    difeq(k,k1,k2,j9,ic3,ic4,indexv,ne,s,y);
    pinvs(ic3,ic4,j5,j9,jc1,k1,c,s);
    for (k=k1+1;k<=k2;k++)
    {
      kp = k - 1;
      difeq(k,k1,k2,j9,ic1,ic4,indexv,ne,s,y);
      red(ic1,ic4,j1,j2,j3,j4,j9,ic3,jc1,jcf,kp,c,s);
      pinvs(ic1,ic4,j3,j9,jc1,k,c,s);
    }
    k = k2 + 1;

    difeq(k,k1,k2,j9,ic1,ic2,indexv,ne,s,y);
    red(ic1,ic2,j5,j6,j7,j8,j9,ic3,jc1,jcf,k2,c,s);
    pinvs(ic1,ic2,j7,j9,jcf,k2+1,c,s);
    bksub(ne,nb,jcf,k1,k2,c);
    err = 0.0;
    for (j=1;j<=ne;j++)
    {
      jv = indexv[j];
      errj = vmax = 0.0;
      km = 0;
      for (k=k1;k<=k2;k++)
      {
        vz = fabsl(cf_gsl_3dmatrix_get(c,j,1,k));
        //printf("\nvz = %Lf  k = %d  j = %d", vz, k, j);
        if (vz > vmax)
```

```
        {
          vmax = vz;
          km   = k;
        }
        errj += vz;
      }
      err += errj/scalv[jv];
      gsl_vector_long_double_set(ermax_gsl,j-1,
        cf_gsl_3dmatrix_get(c,j,1,km)/scalv[jv]);
      gsl_vector_int_set(kmax_gsl,j-1,km); // j-1 NR -> GSL indexing
    }
    err /= nvars;
    fac = (err > slowc ? slowc/err : 1.0);
    for (jv=1;jv<=ne;jv++)
    {
      j = indexv[jv];
      for (k=k1;k<=k2;k++)
        gsl_matrix_long_double_set (y, j-1,k-1,
          gsl_matrix_long_double_get(y,j-1,k-1)
            -fac*cf_gsl_3dmatrix_get(c,jv,1,k));
    }
    /*
    printf("\n%8s %9s %9s\n","Iter.","Error","FAC");
    printf("%6d %12.6Lf %11.6Lf\n",it,err,fac);
    printf("%8s %8s %14s\n","Var.","Kmax","Max. Error");
    for (j=1;j<=ne;j++)
    printf("%6d %9d %14.6Lf \n",indexv[j],
      gsl_vector_int_get(kmax_gsl,j-1),
      gsl_vector_long_double_get(ermax_gsl,j-1));
    */

    if (err < conv) {
      gsl_vector_long_double_free(ermax_gsl);
      gsl_vector_int_free(kmax_gsl);
      return GSL_SUCCESS;
    }
  }
  GSL_ERROR("Too many iterations in SOLVDE",GSL_EMAXITER);
}

/* ==================================================================== */


/* ====================================================================
*  Spin-Weighted Spherical Harmonics
*  ==================================================================== */

unsigned long fac(int n)
{
  if (n < 0)
    GSL_ERROR("\n\nFactorial function input domain error.\n",GSL_EDOM);
  if (n == 0)
    return 1;
  return (n*fac(n-1));
}
```

```

unsigned binomial(int n, int k)
{
  if (k > n || n < 0 || k < 0)
    return 0;
  else if (k == 0 || k == n)
    return 1;
  else
    return binomial(n-1,k-1)+binomial(n-1,k);
//  return fac(n)/fac(n-k)*fac(k);    FACTORIALS!!
}

long double Yslm(int s, int l, int m, long double x)
{
  long double ans;
  long double theta = acosl(x);
  long double sum = 0;
  int r;
  for(r=0;r<=l-s;r++)
    sum += binomial(l-s,r)*binomial(l+s,r+s-m)
      *pow(-1.0,l-r-s)*powl(1.0/tanl(theta/2.0),2.0*r+s-m);
  ans = pow(-1.0,m)
    *sqrtl((fac(l+m)*fac(l-m)*(2*l+1))/(fac(l+s)*fac(l-s)*4.0*M_PI))
    *powl(sinl(theta/2.0),2.0*l)*sum;
  return ans;
}

long double Pslm(int s, int l, int m, long double x)
{
  return sqrt((4.0*M_PI*fac(l+m))/((2.0*l+1.0)*fac(l-m)))
    *Yslm(s,l,m,x);
}

long double YslmPrime(int s, int l, int m, long double x)
{
  long double ans;
  long double theta = acosl(x);
  long double sum = 0;
  int r;
  for(r=0;r<=l-s;r++)
    sum += binomial(l-s,r)*binomial(l+s,r+s-m)*powl(-1.0,l-r-s)
      *(
        (
          -(l+m-2.0*r-s+l*x)*powl((1.0/(tanl(theta/2.0))),-m+2.0*r+s)
            *(1.0/(cosl(theta/2.0)))*powl(sinl(theta/2.0),-1.0+2.0*l)
        )/(2.0*sqrt(1.0-powl(x,2.0)))
      );
  ans = powl(-1.0,m)
    *sqrtl((fac(l+m)*fac(l-m)*(2*l+1))/(fac(l+s)*fac(l-s)*4.0*M_PI))
    *sum;
  return ans;
}

long double PslmPrime(int s, int l, int m, long double x)
{
  return sqrt((4.0*M_PI*fac(l+m))/((2.0*l+1.0)*fac(l-m)))
    *YslmPrime(s,l,m,x);
}

long double xm1xp1YslmLimitL(int s, int l, int m)
{
  if (m > s)
  {
    return powl(2.0,-fabsl(m+s)/2.0+(s-m)/2.0)
      *powl(-1.0,l)
      *sqrtl((fac(l-s)*fac(l+m)*(2*l+1))/(fac(l+s)*fac(l-m)*4.0*M_PI))
      *1.0/fac(m-s);
  }
  else
  {
    return powl(2.0,-fabsl(m+s)/2.0-(s-m)/2.0)
      *powl(-1.0,l-s+m)
      *sqrtl((fac(l+s)*fac(l-m)*(2*l+1))/(fac(l-s)*fac(l+m)*4.0*M_PI))
      *1.0/fac(s-m);
  }
}

 long double xm1xp1PslmLimitL(int s, int l, int m)
{
  return sqrt((4.0*M_PI*fac(l+m))/((2.0*l+1.0)*fac(l-m)))
    *xm1xp1YslmLimitL(s,l,m);
}

long double xm1xp1YslmLimitR(int s, int l, int m)
{
  if (m > -s)
  {
    return powl(2.0,-fabsl(m-s)/2.0-(s+m)/2.0)
      *powl(-1.0,m)
      *sqrtl((fac(l+s)*fac(l+m)*(2*l+1))/(fac(l-s)*fac(l-m)*4.0*M_PI))
      *1.0/fac(s+m);
  }
  else
  {
    return powl(2.0,-fabsl(m-s)/2.0+(s+m)/2.0)
      *powl(-1.0,s)
      *sqrtl((fac(l-s)*fac(l-m)*(2*l+1))/(fac(l+s)*fac(l+m)*4.0*M_PI))
      *1.0/fac(-s-m);
  }
}

 long double xm1xp1PslmLimitR(int s, int l, int m)
{
  return sqrt((4.0*M_PI*fac(l+m))/((2.0*l+1.0)*fac(l-m)))
    *xm1xp1YslmLimitR(s,l,m);
}

/* ==================================================================== */


/* ====================================================================
*  I/O Functions
*  ==================================================================== */

void output_slm(int s, int l, int m, long double c, FILE *fp)
{
  fprintf(fp,"%d %d %d %Lf.19",s,l,m,c);
  fprintf(fp,"\n");
}

void output_x(gsl_vector_long_double *x, FILE *fp)
{
  int i;
  for(i=0;i<=M-1;i++)
    fprintf(fp,"%.19Lf ",gsl_vector_long_double_get(x,i));
  fprintf(fp,"\n");
}

void output_y(gsl_matrix_long_double *y, FILE *fp)
{
  int i;
  for(i=0;i<=M-1;i++)
    fprintf(fp,"%.19Lf ",gsl_matrix_long_double_get(y,0,i));
  fprintf(fp,"\n");
  for(i=0;i<=M-1;i++)
    fprintf(fp,"%.19Lf ",gsl_matrix_long_double_get(y,1,i));
  fprintf(fp,"\n");
}

void output_SWSH(gsl_matrix_long_double *y, FILE *fp)
{
  int i;
  for(i=0;i<=M-1;i++)
    fprintf(fp,"%.19Lf ",
      powl(1.0-gsl_vector_long_double_get(x,i),kp)
        *powl(1.0+gsl_vector_long_double_get(x,i),kn)
        *gsl_matrix_long_double_get(y,0,i)
    );
  fprintf(fp,"\n");
  for(i=0;i<=M-1;i++)
    fprintf(fp,"%.19Lf ",
      -powl(1.0-gsl_vector_long_double_get(x,i),-1.0+kp)
        *powl(1.0+gsl_vector_long_double_get(x,i),-1.0+kn)
        *
        (
          (
            -kn+kp
            +kn*gsl_vector_long_double_get(x,i)
            +kp*gsl_vector_long_double_get(x,i)
          )
          *gsl_matrix_long_double_get(y,0,i)
          -(1.0-powl(gsl_vector_long_double_get(x,i),2.0))
          *gsl_matrix_long_double_get(y,1,i)
        )
    );
  fprintf(fp,"\n");
}

/* ====================================================================== */


/* ======================================================================
*   Spheroidal Harmonics
*   ====================================================================== */

int main()
{
  /* Declaration and allocation */
  int itmax,k,indexv[NE+1];
  long double conv,deriv,fac1,fac2,q1,slowc,scalv[NE+1];
  gsl_matrix_long_double *s;
  gsl_matrix_long_double *y;
  gsl_matrix_long_double **c;

  FILE *data;
  FILE *slm;
  data = fopen("Sfroid_test.txt", "w");
  slm = fopen("slm.txt", "w");

  x = gsl_vector_long_double_alloc(M);
  y = gsl_matrix_long_double_alloc(NYJ,NYK);
  s = gsl_matrix_long_double_alloc(NSI,NSJ);
  c = cf_gsl_3dmatrix_alloc(NCI,NCJ+1,NCK+1);

  /* Solver Parameters */
  itmax = 2000;
  conv  = 5.0e-6;
  slowc = 1.0;
  h     = 2.0/(M-1);

  printf("\nenter s l m\n");
  scanf("%d %d %d",&ss,&ll,&mm);

  kn = fabsl(mm-ss)/2.0;
  kp = fabsl(mm+ss)/2.0;

  printf("\nEnter c (long double) or 999 to cancel\n");
  double windows_work_around;
  scanf("%Lf",&windows_work_around);
  tar = windows_work_around;
  cc  = 0.0;

  /* Index to swap rows/columns to prevent zero pivot
  *  elements / singular matrix */
  indexv[1]=1;
  indexv[2]=2;
  indexv[3]=3;

  anorml = xm1xp1PslmLimitL(ss,ll,mm);
  anormr = xm1xp1PslmLimitR(ss,ll,mm);
```

```

if ( fabsl(anorml) > fabsl(anormr) )
  anorm = anorml;
else
  anorm = anormr;

/* Interior IC based on associated Legendre polynomials. This is
*  largely unchanged */
for (k=1;k<=(M-2);k++)
{
  gsl_vector_long_double_set(x,k,
    (k)*h-1.0
  );
  gsl_matrix_long_double_set(y,0,k,
    powl(1.0-gsl_vector_long_double_get(x,k),-kp)
      *powl(1.0+gsl_vector_long_double_get(x,k),-kn)
      *Pslm(ss,ll,mm,gsl_vector_long_double_get(x,k))
  );
  gsl_matrix_long_double_set(y,1,k,
    powl(1.0-gsl_vector_long_double_get(x,k),-1.0-kp)
      *powl(1.0+gsl_vector_long_double_get(x,k),-1.0-kn)
      *
      (
        powl(1.0-gsl_vector_long_double_get(x,k),kp)
          *powl(1.0+gsl_vector_long_double_get(x,k),kn)
          *(
              -kn+kp
              +kn*gsl_vector_long_double_get(x,k)
              +kp*gsl_vector_long_double_get(x,k)
            )
          *gsl_matrix_long_double_get(y,0,k)
        +(1.0-powl(gsl_vector_long_double_get(x,k),2.0))
          *PslmPrime(ss,ll,mm,gsl_vector_long_double_get(x,k))
      )
  );
  gsl_matrix_long_double_set(y,2,k,ll*(ll+1)-ss*(ss+1));
}

/* IC at x(-1) */
gsl_vector_long_double_set(x,0,-1.0);
gsl_matrix_long_double_set(y,0,0,anorml);
gsl_matrix_long_double_set(y,2,0,ll*(ll+1)-ss*(ss+1));
gsl_matrix_long_double_set(y,1,0,
  gsl_matrix_long_double_get(y,0,0)
    *(
      (
        -4.0*gsl_matrix_long_double_get(y,2,0)-4.0*powl(cc,2.0)+4.0*kn
        +4.0*powl(kn,2.0)+mm*mm-4.0*ss-8.0*cc*ss+2.0*mm*ss-3.0*ss*ss
      )
        /(
          16.0*kn+8.0*kn*kn-2.0*(-4.0+mm*mm-2.0*mm*ss+ss*ss)
        )
      +kp/2.0
    )
);
```

```

/* IC at x(1) */
gsl_vector_long_double_set(x,M-1,1.0);
gsl_matrix_long_double_set(y,0,M-1,anormr);
gsl_matrix_long_double_set(y,2,M-1,ll*(ll+1)-ss*(ss+1));
gsl_matrix_long_double_set(y,1,M-1,
  gsl_matrix_long_double_get(y,0,M-1)
    *(
      (
        4.0*gsl_matrix_long_double_get(y,2,M-1)+4.0*powl(cc,2.0)-4.0*kp
        -4.0*powl(kp,2.0)-mm*mm+4.0*ss-8.0*cc*ss+2.0*mm*ss
        +3.0*powl(ss,2.0)
      )
        /(
          16.0*kp+8.0*powl(kp,2.0)-2.0*(-4.0+mm*mm+2.0*mm*ss
          +powl(ss,2.0))
        )
      -kn/2.0
    )
);

output_slm(ss,ll,mm,cc,slm);
output_x(x,data);
output_SWSH(y,data);

/* Scaling for convergence rate control */
scalv[1] = fabsl(anorm);
if (gsl_matrix_long_double_get(y,1,M-1) > fabsl(anorm))
scalv[2] = gsl_matrix_long_double_get (y,1,M-1);
else
scalv[2] = fabsl(anorm);
if (gsl_matrix_long_double_get(y,2,M-1) > 1.0)
scalv[3] = gsl_matrix_long_double_get(y,2,M-1);
else
scalv[3] = 1.0;

while(1){
  if (windows_work_around == 999 || cc >= tar){
    output_SWSH(y,data);
    cf_gsl_3dmatrix_free(c,NCI);
    gsl_matrix_long_double_free(s);
    gsl_matrix_long_double_free(y);
    fclose(data);
    fclose(slm);
    printf("\nc = %.2Lf  lambda = %.15Lf  mathematica = %.15Lf\n",
      cc,gsl_matrix_long_double_get(y,2,0),
      gsl_matrix_long_double_get(y,2,0)-powl(cc,2.0));
    return 0;
  }

  cc += 0.1;

  solvde(itmax,conv,slowc,scalv,indexv,NE,NB,M,y,c,s,data);
 // printf("\nc = %.2Lf  lambda = %.15Lf  mathematica = %.15Lf\n",
 // cc,gsl_matrix_long_double_get(y,2,0),
    // gsl_matrix_long_double_get(y,2,0)-powl(cc,2.0));

    // printf("\nEnter c (long double) or 999 to finish\n");
  }
}
```

Bibliography

References

[1] Saul A Teukolsky. Rotating Black Holes: Separable Wave Equations for Gravitational and Electromagnetic Perturbations. *Phys. Rev. Lett.*, 29(16):1114–1118, October 1972.

[2] V.P. Frolov and I.D. Novikov. *Black Hole Physics: Basic Concepts and New Developments*. Fundamental theories of physics : an international book series on the fundamental theories of physics. Kluwer, 1998.

[3] Emanuele Berti, Vitor Cardoso, and Marc Casals. Eigenvalues and eigenfunctions of spin-weighted spheroidal harmonics in four and higher dimensions. *Phys. Rev. D*, 73:024013, Jan 2006.

[4] M. Abramowitz and I.A. Stegun. *Handbook of Mathematical Functions: With Formulas, Graphs, and Mathematical Tables*. Applied mathematics series. Dover Publications, 1964.

[5] F.W.J. Olver, National Institute of Standards, and Technology (U.S.). *NIST Handbook of Mathematical Functions*. Cambridge University Press, 2010.

[6] C. Flammer. *Spheroidal Wave Functions*. Monograph. Stanford University Press, 1957.

[7] J. Meixner, F.W. Schäfke, and G. Wolf. *Mathieu Functions and Spheroidal Functions and Their Mathematical Foundations: Further Studies*. Lecture Notes in Mathematics. Springer, 1980.

[8] L.W. Li, X.K. Kang, and M.S. Leong. *Spheroidal Wave Functions in Electromagnetic Theory*. Wiley Series in Microwave and Optical Engineering. Wiley, 2004.

[9] Le-Wei Li, Mook-Seng Leong, Tat-Soon Yeo, Pang-Shyan Kooi, and Kian-Yong Tan. Computations of spheroidal harmonics with complex arguments: A review with an algorithm. *Physical Review E*, 58(5):6792–6806, November 1998.

[10] Tomohiro Oguchi. Eigenvalues of spheroidal wave functions and their branch points for complex values of propagation constants. *Radio Science*, 5(September):1207–1214, 1970.

[11] JN Goldberg, AJ Macfarlane, Ezra T Newman, F Rohrlich, and ECG Sudarshan. Spin-s spherical harmonics and Ð. *Journal of Mathematical Physics*, 8:2155, 1967.

[12] Stephen Wolfram. Spin-Weighted Spherical Harmonics. `http://demonstrations.wolfram.com/SpinWeightedSphericalHarmonics/`. [Online; accessed 04-September-2013].

[13] Marianela Lentini, Michael R. Osborne, and Robert D. Russell. The close relationships between methods for solving two-point boundary value problems. *SIAM Journal on Numerical Analysis*, 22(2):pp. 280–309, 1985.

[14] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing.* Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[15] P. Amodio, J. R. Cash, G. Roussos, R. W. Wright, G. Fairweather, I. Gladwell, G. L. Kraut, and M. Paprzycki. Almost block diagonal linear systems: sequential and parallel solution techniques, and applications. *Numerical Linear Algebra with Applications*, 7(5):275–317, 2000.

[16] GNU. GNU Scientific Library. `http://www.gnu.org/software/gsl/`. [Online; accessed 05-September-2013].

[17] M.A. Schoonover, J.S. Bowie, and W.R. Arnold. *Gnu Emacs: Unix Text Editing and Programming.* Hewlett-Packard Press Series. ADDISON WESLEY Publishing Company Incorporated, 1992.