Equilibrium models of strongly-magnetized neutron stars

Konstantinos Palapanidis

MSc in Computational Physics
Department of Physics
Aristotle University of Thessaloniki

June 2014

Supervisor: Prof. Nikolaos Stergioulas

Abstract

We construct equilibrium configurations of strongly-magnetized, rotating neutron stars with mixed poloidal and toroidal components, using an iterative numerical method. The toroidal component contains only a small fraction of the total magnetic energy. The accuracy of the numerical solutions is evaluated using the virial theorem and by studying the convergence to high accuracy after a number of iterations. First, we consider a normal fluid interior, in the ideal magnetohydrodynamics approximation (and a vacuum exterior) and verify that our code reproduces published results in the literature with good accuracy. Next, we assume that the neutron star has two regions: the inner core, which is modelled as a two-component fluid consisting of type-II superconducting protons and superfluid neutrons, and the crust, a region composed of normal matter. In this case, we find significant differences compared to the case of a normal matter core and qualitatively confirm recently published results in the strong-field case. For magnetic field strengths typical of normal pulsars, we find that the very weak coupling of the Grad-Shafranov equation with the equation of hydrostatic equilibrium requires a significantly larger number of iterations, before the magnetic field configuration relaxes to a converged solution. In this respect, we improve over previously published results.

Acknowledgements

I would like to thank Prof. Nikolaos Stergioulas for supervising and helping me to accomplish this project and for motivating me during several occasions as well. His continuous guidance was of crucial importance throughout this project. Also I would like to thank Dr. Sam Lander for his helpful comments and discussions.

Furthermore, I would like to thank my friends for supporting me, especially Despina Pazouli for discussing various parts of this project and Zisis Karampaglis for providing his knowledge on C language programming. Finally, I would like to thank my family for providing continuous support and help.

Contents

Chapter 1

Introduction

In the current thesis we examine equilibrium states of strongly magnetized neutron stars that consist of superconducting and superfluid components. Thus, in this chapter, we will briefly introduce neutron stars, then we will describe the superconducting and superfluid properties of matter and finally we will discuss the role of superconductivity and superfluidity in neutron stars.

1.1 Neutron stars

Neutron stars are being extensively researched, since their first discovery. Their extreme properties demand an interdisciplinary approach, while most of the existing models attempt to synthesize the various feauures that neutron stars are predicted to possess.

Neutron stars are stellar objects created after the collapse of massive stars (with main-sequence mass of $M \gtrsim 10M_{\odot}$). Their radius is in an uncertain range of 9 km - 15 km while observed masses are in a range of $\sim 1.3 - 2M_{\odot}$. Their high compactness suggests that general relativistic effects are imporant. However, when including new physical effects, it is customary to first solve the problem in a Newtonian context, in order to gain insight. Neutrons stars are the most dense known objects, with core densities of the order of $10^{15} gr/cm^3$, a value that exceeds the standard the nuclear saturation density. Most of these stars rotate with periods that vary depending on their age, ranging between milliseconds and seconds. Despite the fact that most of the neutron stars exhibit a periodic rotation of extreme precision, having typical spin down of $\sim 10^{-13} s/s$, sometimes they accelerate or decelerate suddenly. These phenomena are conjectured to r elate with a structural re-organization, which is could be related to the superconducting and superfluid properties of matter. Neutron stars are also known for their strong magnetic field, which ranges

from $10^8\,G$ to about $10^{15}\,G$. The latter value characterizes a specific population of neutrons stars (magnetars).

The observed braking indices imply a strong dipolar component, but the existence of internal toroidal magnetic field components is also theorized (but there is currently no observational basis for their relative strength). There already exists formulations that consider a mixed poloidal/toroidal internal magnetic field, although in most of the cases (Tommimura & Eriguchi [25], Lander & Jones [14], Lander [18]) the internal toroidal component accounts for a very small fraction of the total magnetic energy stored in the star.

In spite of the fact that the internal composition of neutron stars has been studied for decades, their structure is highly uncertain, as the core may consist of exotic matter. The outer core is thought to be composed of normal electrons, as well as degenerate protons and neutrons, which exist in a superconducting and a superfluid phase, respectively. The inner crust consists of protons and neutrons and the outer crust is considered to be solid. The aforementioned regions are strongly dependent (both in composition and extent) on the (largely unknown) equation of state describing matter at extremely high densities. Finally, rapid rotation or very strong magnetic fields can deform the shape of the equilibrium configuration away from spherical.

1.2 Superconductivity and superfluidity

Superconductivity and superfluidity are exotic properties of matter existing under specific conditions. They are exotic in the sense that they happen under extreme, with respect to normal circumstances, conditions and moreover they appear to be counterintuitive. Here, we review these properties, providing a qualitative explanation as well as describing the conditions under which they exist. We follow Glampedakis, Andersson & Samuelsson [8] and Annett [2].

To begin with, superconductivity and superfluidity are phenomena with the common property that the quantum behavior of matter is apparent on macroscopic scales. Superconductivity refers to charged systems, such as electrons in metals, while superfluidity concerns neutral systems, such as the liquid helium ^{3}He. Such systems, when cooled to temperatures close to absolute zero ($T = 0\,K$) do not convert to solids, but remain in a liquid phase (a quantum-liquid phase). This happens because in this phase the zero energy level of the particles, as described by quantum mechanics, is relatively large with respect to their energy through interaction. The superconducting/superfluid properties occur as long as the system is below some temperature, the critical temperature $T_{\rm c}$ (denoted as T_λ for superfluids). For most materials this temperature is relatively small and very close to $T = 0\,K$.

The main macroscopic properties of superconductors and superfluids are the zero electrical resistivity $\varrho = 0$ and the flow with vanishing viscosity, correspondingly. Although both superconductors and superfluids exhibit interesting properties, we will concentrate on superconductivity since it is related to the different nature of the magnetic force existing in our model.

1.2.1 Type-I and type-II superconductors

Superconductors exhibit perfect diamagnetism, the so-called Meissner effect. During this phenomenon a superconductor expels any externally imposed weak magnetic field, leaving zero magnetic field inside the material, by creating an opposite field that cancels the outer one. The main difference between superconductive and normal perfectly conducting matter is that magnetic field expulsion cannot be entirely explained by zero resistivity and Maxwell's equations.
Ohm's law

$$\mathbf{j} = \sigma \mathbf{E}, \tag{1.1}$$

for a material with infinite conductivity σ, states that it is possible to have a non vanishing current density $\mathbf{j}$ for a vanishing electric field $\mathbf{E}$. The Maxwell-Faraday equation

$$\boldsymbol{\nabla} \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}, \tag{1.2}$$

for a vanishing $\mathbf{E}$ field is

$$\frac{\partial \mathbf{B}}{\partial t} = 0, \tag{1.3}$$

where $\mathbf{B}$ is the magnetic field strength, which means that the magnetic field does not have a time dependence.

Now, we assume a material in a superconducting phase ($T < T_{\mathrm{c}}$) and vanishing magnetic field. It is obvious through (1.3) that if an external magnetic field is applied, an opposite magnetic field will be created inside the superconductor, so that $\mathbf{B}$ will remain zero. In case that initially the material was not in superconductive phase ($T > T_{\mathrm{c}}$) and subject to an external magnetic field, there would be an internal non-vanishing magnetic field. If the temperature was gradually lowered below the critical, so the material would enter the superconducting phase, then the internal $\mathbf{B}$ field should remain unaltered as stated by (1.3). However, the fact that *the magnetic field is expelled in that case too*, implies that superconductivity is a phenomenon that cannot be described only by classical electromagnetism. Thus, an infinitely conducting material is identified as a superconductor if it exhibits the Meis sner effect.

Superconductors are divided into two categories, type I and type II, depending on their behavior when interacting with an external magnetic field. Type-I superconductors are in a superconducting

phase as long as the external field is below some critical magnetic intensity $H_{\rm c}$. Above $H_{\rm c}$, superconductivity is destroyed and the external field penetrates the superconductor (Fig. 1.1a). On the other hand, type-II superconductors are characterized by two critical fields, $H_{\rm c1}$ and $H_{\rm c2}$ (with $H_{\rm c1} < H_{\rm c2}$). When the external magnetic field has intensity smaller than $H_{\rm c1}$, the superconductor has similar behavior to type I, expelling the field entirely. In the case that the external field has intensity between $H_{\rm c1}$ and $H_{\rm c2}$, the type II superconductor retains a non-vanishing **B** field inside. When the imposed field is increased beyond $H_{\rm c2}$, the material is no longer superconductive (Fig . 1.1b).

The type of superconductivity is microscopically determined by the following ratio

$$\kappa_{\rm s} = \frac{\xi}{\sqrt{2}\Lambda_\star}, \tag{1.4}$$

where ξ is the *coherence length* of the particles that consist the superconductor and $\Lambda_\star$ is the *penetration length*. Microscopically, the particles form the so-called Cooper pairs, which are formed by particles that exist in interacting quantum states. The characteristic length at which such pairs are broken is the coherence length. The penetration length, on the other hand, is the distance in the superconductor beyond which the magnetic field vanishes, due to the cancellation of the external field by the internal field. Type II superconductors are characterized by $\kappa_{\rm s} < 1$, otherwise we are dealing with a type I superconductor.

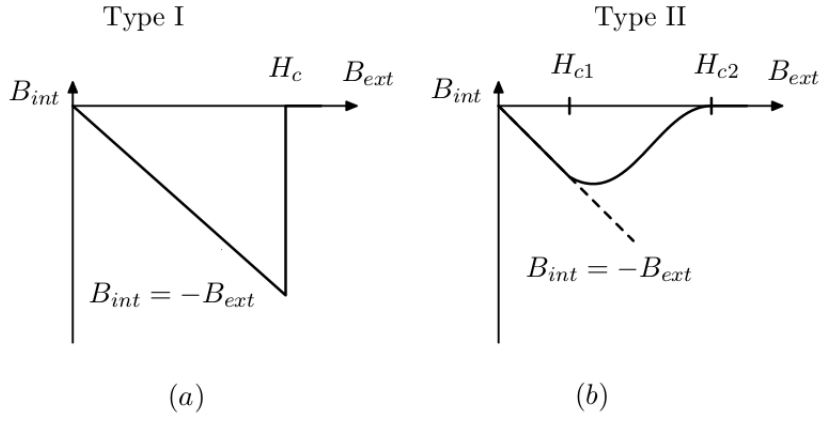


Figure 1.1: Internal magnetic field response to an externally imposed for type-I (a) and type-II (b) superconductors. Adapted from Annett [2].

1.3 Superconductivity and superfluidity in neutron stars

As already mentioned, neutron stars are theorized to contain superconducting and superfluid components in their core. Although, the critical temperatures for the known laboratory superconductive/superfluid materials are a few Kelvin (with the exception of high-temperature superconductors), hot neutron stars still possess those properties. This happens because the corresponding Fermi temperature for such dense matter ($T_{\rm F} \sim 10^{12} K$) is substantially higher than typical temperatures inside old neutron stars ($T_{\rm NS} \sim 10^{8} K$). The magnetic field can thus be expected to have been expelled from the superconducting core due to the Meissner effect. However, the predicted timescale of the expulsion is of the order of Myrs, so younger neutron stars may still be in a metastable phase. It can be estimated (see Glampedakis, Andersson & Samuelsson [8]) that $\kappa_{\rm s}$ for the outer core of a neutron star in equilibrium is less than one ($\kappa_{\rm s} < 1$) and hence that region should posses protons in a type II superconducting phase, with critical fields of $H_{\rm c1} \sim 10^{15} G$ and $H_{\rm c2} \sim 10^{16} G$. Although metastable, the outer core is in a type II superconducting phase also for $B < H_{\rm c1}$. The inner core, however, could exist in a type I superconducting phase, if it is of sufficiently high density.

In the present model, we are interested in type II superconductivity only, as it has been formulated in the context of magnetohydrodynamics (MHD) by Glampedakis, Andersson & Samuelsson [8]. Following Lander [18] we use an approximation of the total set of MHD equations. The magnetic force we use is

$$\mathbf{F}_{\rm mag} = -\frac{1}{4\pi}\left[\mathbf{B} \times (\boldsymbol{\nabla} \times \mathbf{H}_{\rm c1}) + \rho_{\rm p} \boldsymbol{\nabla}\left(B \frac{\partial H_{\rm c1}}{\partial \rho_{\rm p}}\right)\right]. \tag{1.5}$$

where $\rho_{\rm p}$ is the density of protons. This is obtained when one neglects the coupling between protons and neutrons and considers the star to be nonrotating.

Chapter 2

Rotating magnetized neutron stars

2.1 Introduction

First, we examine equilibrium configurations of one-component, magnetized, barotropic rotating neutron stars, in the Newtonian framework. Since we are interested in equilibrium states, we solve the integral form of the Euler-Lagrange equations. The implemented numerical scheme is an extension of the Hachisu Self Consistent Field method (HSCF) (Hachisu [10]), in which the magnetic field is included. The numerical code is based on a non-magnetized version, originally written by N. Stergioulas, which we extended here to include magnetic fields of mixed type for normal (and also for superconducting) cores. In the last section, we present our results and compare with previously published cases by Tomimura & Eriguchi [25].

2.2 Basic theory

2.2.1 Main equations

We assume a rotating, magnetized, axisymmetric neutron star (NS) in a stationary state. The magnetic dipole axis is aligned with the rotational axis, which is also the axis of symmetry. We use the ideal MHD approximation, with infinitely conducting matter. The exterior of the star is assumed to be vacuum. The star is also assumed to exhibit equatorial symmetry. Working in the Newtonian framework, we use a number of equation to specify the equilibrium. The first is the equation of motion

$$-\frac{1}{\rho}\boldsymbol{\nabla} P - \boldsymbol{\nabla}\Phi_{\rm g} + \boldsymbol{\nabla}\Phi_{\rm r} + \frac{\boldsymbol{\mathcal{L}}}{\rho} = 0, \tag{2.1}$$

where ρ is the density of the fluid, P the pressure, $\Phi_{\rm g}$ the gravitational potential, $\Phi_{\rm r}$ the centrifugal potential and $\boldsymbol{\mathcal{L}}$ the Lorentz force ($\boldsymbol{\mathcal{L}} = \mathbf{j} \times \mathbf{B}$) with $\mathbf{j}$ being the current density. In a more general context, the Lorentz force will be denoted as $\mathbf{F}_{\rm mag}$. The gravitational potential is related to the density of the fluid through Poisson's equation

$$\boldsymbol{\nabla}^2 \Phi_{\rm g} = 4\pi G \rho, \tag{2.2}$$

where G is the gravitational constant. In ideal MHD (for a vanishing electric field $\mathbf{E}$), Ampère's law becomes

$$\boldsymbol{\nabla} \times \mathbf{B} = 4\pi \mathbf{j}. \tag{2.3}$$

Gauss' law for the magnetic field yields

$$\boldsymbol{\nabla} \cdot \mathbf{B} = 0. \tag{2.4}$$

Finally, an equation of state for the matter is required so that the system is closed. We assume that the fluid is governed by a barotropic equation of state

$$P = P(\rho), \tag{2.5}$$

and specifically by the polytropic relation

$$P = K \rho^{1+\frac{1}{N}}, \tag{2.6}$$

where K is the polytropic constant and N is the polytropic index. Assuming rigid rotation, (all fluid elements have the same anglular velocity Ω_0) the centrifugal potential in spherical polar coordinates (with the $z-$axis aligned with the symmetry axis) is

$$\Phi_{\rm r} = \frac{1}{2} \Omega_0^2 \varpi^2, \tag{2.7}$$

where $\varpi = r \sin\theta$.

Defining the enthalpy as $H = \int \frac{dP}{\rho(P)}$, the first term of (2.1) can be written as $-\frac{1}{\rho}\boldsymbol{\nabla} P = -\boldsymbol{\nabla} H$.

2.2.2 The Grad-Shafranov equation

Applying the curl operator to (2.1) and due to the identity that the curl of the gradient is zero for any scalar, we obtain

$$\boldsymbol{\nabla} \times \left(\frac{\boldsymbol{\mathcal{L}}}{\rho} \right) = 0. \tag{2.8}$$

This yields that $\frac{\boldsymbol{\mathcal{L}}}{\rho}$ is equal to the gradient of some function M.

$$\frac{\boldsymbol{\mathcal{L}}}{\rho} = \boldsymbol{\nabla} M. \tag{2.9}$$

In axisymmetry, a scalar quantity u will dependent only on ϖ and z and we can describe the general form of the magnetic field as

$$\mathbf{B} = \frac{1}{\varpi}\boldsymbol{\nabla} u \times \mathbf{e}_\phi + B_\phi \mathbf{e}_\phi, \tag{2.10}$$

which is equivalent to writing the magnetic field as

$$\mathbf{B} = \mathbf{B}_{\rm pol} + \mathbf{B}_{\rm tor}, \tag{2.11}$$

where $\mathbf{B}_{\rm pol} = \frac{1}{\varpi}\boldsymbol{\nabla} u \times \mathbf{e}_\phi$ is the poloidal part (with components only along the unit vectors $\mathbf{e}_\varpi$ and $\mathbf{e}_z$) and $\mathbf{B}_{\rm tor} = B_\phi \mathbf{e}_\phi$ the toroidal part. Substituting in Ampère's law (2.3) we obtain

$$\boldsymbol{\nabla} \times (\mathbf{B}_{\rm pol} + \mathbf{B}_{\rm tor}) = 4\pi \left(\mathbf{j}_{\rm pol} + \mathbf{j}_{\rm tor}\right), \tag{2.12}$$

where similarly to the magnetic field, we assume that the current density can be decomposed into a poloidal part, $\mathbf{j}_{\rm pol}$, and a toroidal part, $\mathbf{j}_{\rm tor}$. After some manipulation, we derive

$$\boldsymbol{\nabla} \times \mathbf{B}_{\rm pol} = \boldsymbol{\nabla} \times \left(\frac{1}{\varpi}\boldsymbol{\nabla} u \times \mathbf{e}_\phi\right) = -\frac{1}{\varpi}\underbrace{\left(\frac{\partial^2 u}{\partial \varpi^2} - \frac{1}{\varpi}\frac{\partial u}{\partial \varpi} + \frac{\partial^2 u}{\partial z^2}\right)}_{\Delta_\star u}\mathbf{e}_\phi, \tag{2.13}$$

and

$$\boldsymbol{\nabla} \times \mathbf{B}_{\rm tor} = \boldsymbol{\nabla} \times (B_\phi \mathbf{e}_\phi) = -\frac{\partial B_\phi}{\partial z}\mathbf{e}_\varpi + \frac{1}{\varpi}\frac{\partial(\varpi B_\phi)}{\partial \varpi}\mathbf{e}_z = \frac{1}{\varpi}\boldsymbol{\nabla}(\varpi B_\phi) \times \mathbf{e}_\phi. \tag{2.14}$$

It is obvious that the right hand sides of (2.13) and (2.14) are toroidal and poloidal and hence equal to $4\pi\mathbf{j}_{\rm tor}$ and $4\pi\mathbf{j}_{\rm pol}$ respectively. The total current takes the form

$$\mathbf{j} = \underbrace{\frac{1}{4\pi\varpi}\boldsymbol{\nabla}(\varpi B_\phi) \times \mathbf{e}_\phi}_{\mathbf{j}_{\rm pol}} \underbrace{-\frac{1}{4\pi\varpi}\Delta_\star u\, \mathbf{e}_\phi}_{\mathbf{j}_{\rm tor}}, \tag{2.15}$$

which describes the current density in terms of u and B_ϕ.
Substituting these results in the Lorentz force, we obtain

$$\begin{aligned} \boldsymbol{\mathcal{L}} &= \mathbf{j} \times \mathbf{B} = \\ &= (\mathbf{j}_{\rm pol} + \mathbf{j}_{\rm tor}) \times (\mathbf{B}_{\rm pol} + \mathbf{B}_{\rm tor}) = \\ &= \underbrace{\mathbf{j}_{\rm pol} \times \mathbf{B}_{\rm pol}}_{\boldsymbol{\mathcal{L}}_{\rm tor}} + \underbrace{\mathbf{j}_{\rm pol} \times \mathbf{B}_{\rm tor} + \mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm pol} + \mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm tor}}_{\boldsymbol{\mathcal{L}}_{\rm pol}}. \end{aligned} \tag{2.16}$$

Since all quantities in the equation of motion (2.1) are independent of ϕ (due to axisymmetry) the same must hold for the Lorentz force and hence the toroidal component should be zero ($\boldsymbol{\mathcal{L}}_{\rm tor} = 0$). In order to produce mixed-field configurations (i.e. a $\mathbf{B}$ field with both toroidal and poloidal components) we assume that $\mathbf{B}_{\rm pol}$ is parallel to $\mathbf{j}_{\rm pol}$. If we set $\mathbf{B}_{\rm pol} = 0$ we obtain a purely toroidal $\mathbf{B}$ field.

Using $\mathbf{j}_{\rm pol} \times \mathbf{B}_{\rm pol} = 0$ and (2.10, 2.15) we obtain (see also A.1.1.1)

$$\boldsymbol{\nabla}\left(\varpi\, B_\phi\right) \times \boldsymbol{\nabla} u = 0. \tag{2.17}$$

It follows (see A.2.3.6) that $\varpi\, B_\phi$ must be a function of u

$$\varpi\, B_\phi = f(u). \tag{2.18}$$

Using (2.10, 2.15, and 2.18) we decompose $\boldsymbol{\mathcal{L}}_{\rm pol}$ (see A.1.1.2) so that we obtain a simpler form for the Lorentz force. The first term ($\mathbf{j}_{\rm pol} \times \mathbf{B}_{\rm tor}$) is

$$\mathbf{j}_{\rm pol} \times \mathbf{B}_{\rm tor} = -\frac{f(u)}{4\pi\varpi^2}\frac{df}{du}\boldsymbol{\nabla} u, \tag{2.19}$$

while the second term ($\mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm pol}$) is

$$\mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm pol} = -\frac{1}{4\pi\varpi^2}\Delta_\star u \boldsymbol{\nabla} u. \tag{2.20}$$

The last term ($\mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm tor}$) vanishes, since it is the cross product of parallel vectors

$$\mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm tor} = 0. \tag{2.21}$$

Substituting (2.19, 2.20 and 2.21) into (2.16) the Lorentz force takes the following form

$$\boldsymbol{\mathcal{L}} = \left(-\frac{f(u)}{4\pi\varpi^2}\frac{df}{du} - \frac{1}{4\pi\varpi^2}\Delta_\star u\right)\boldsymbol{\nabla} u. \tag{2.22}$$

Having derived a form for the Lorentz force that is directly related to u, we return to (2.9) and by equating the two parts, we have

$$\rho\,\boldsymbol{\nabla} M = \left(-\frac{f(u)}{4\pi\varpi^2}\frac{df}{du} - \frac{1}{4\pi\varpi^2}\Delta_\star u\right)\boldsymbol{\nabla} u, \tag{2.23}$$

which implies that

$$\rho\,\boldsymbol{\nabla} M \times \boldsymbol{\nabla} u = 0. \tag{2.24}$$

Using (A.2.3.6) we derive $\boldsymbol{\nabla} M = \frac{dM}{du}\boldsymbol{\nabla} u$. Substituting in (2.23) and assuming that $\boldsymbol{\nabla} u \neq 0$ we finally obtain

$$\rho\,\frac{dM}{du} = -\frac{f(u)}{4\pi\varpi^2}\frac{df}{du} - \frac{1}{4\pi\varpi^2}\Delta_\star u, \tag{2.25}$$

which is the Grad-Shafranov equation.
The current density (2.15) is related to the magnetic field through

$$\mathbf{j} = \frac{1}{4\pi}\frac{df}{du}\mathbf{B} + \rho\varpi\frac{dM}{du}\,\mathbf{e}_\phi, \tag{2.26}$$

where we used using (2.10) and (2.25). Setting $\frac{df}{du} = \alpha(u)$ and $\frac{dM}{du} = \kappa(u)$ we obtain

$$\mathbf{j} = \frac{1}{4\pi}\alpha(u)\mathbf{B} + \varpi\rho\kappa(u)\mathbf{e}_\phi. \tag{2.27}$$

The functions $\alpha(u)$ and $\kappa(u)$ describe the different aspects of the magnetic field. For $\alpha(u) = 0$, the field is purely poloidal (since the current is then purely toroidal).

The scalar u is related to the vector potential $\mathbf{A}$ as follows: It holds that

$$\boldsymbol{\nabla} \times \mathbf{A} = \mathbf{B}. \tag{2.28}$$

Decomposing the curl of $\mathbf{A}$ and equating it with the definition of $\mathbf{B}$ (2.10) we obtain

$$\begin{aligned} &\left(\frac{1}{\varpi}\frac{\partial A_z}{\partial\phi} - \frac{\partial A_\phi}{\partial z}\right)\mathbf{e}_\varpi + \frac{1}{\varpi}\left(\frac{\partial(\varpi A_\phi)}{\partial\varpi} - \frac{\partial A_\varpi}{\partial\phi}\right)\mathbf{e}_z + \left(\frac{\partial A_\varpi}{\partial z} - \frac{\partial A_z}{\partial\varpi}\right)\mathbf{e}_\phi \\ = \;& \frac{1}{\varpi}\frac{\partial u}{\partial\varpi}\mathbf{e}_z - \frac{1}{\varpi}\frac{\partial u}{\partial z}\mathbf{e}_\varpi + \frac{f(u)}{\varpi}\mathbf{e}_\phi. \end{aligned} \tag{2.29}$$

Due to axisymmetry, the derivatives with respect to ϕ vanish, so the previous relation is decomposed as

$$\frac{\partial(\varpi A_\phi)}{\partial z} = \frac{\partial u}{\partial z}, \tag{2.30}$$

$$\frac{\partial(\varpi A_\phi)}{\partial\varpi} = \frac{\partial u}{\partial\varpi}, \tag{2.31}$$

$$\left(\frac{\partial A_\varpi}{\partial z} - \frac{\partial A_z}{\partial\varpi}\right) = \frac{f(u)}{\varpi}, \tag{2.32}$$

which directly integrate to $u = A_\phi\varpi$ and hence A_ϕ contains all the information for $\mathbf{B}$.

2.3 Integral form of equations

Here we convert the differential form of the main equations into integral form, which is suitable for numerical integration.

2.3.1 Assumptions

A rotating barotrope with purely rotational velocity (absence of meridional currents) is symmetry about the equatorial plane. We use spherical polar coordinates (r, θ, ϕ) with the rotation axis along $z-$direction. The distance to the rotation axis, is $\varpi = r\sqrt{1-\mu^2}$, where we set $\mu = \cos\theta$ (see A.2.3.7).

We also assume that the function $\alpha(u)$ is of the following form

$$\alpha(u) = a(u - u_{\max})^{\zeta}\, \theta(u - u_{\max}), \tag{2.33}$$

where ζ and a are constants (here $\zeta = 1$) and $\theta(u)$ is the Heavyside step function as it is used in Tomimura & Eriguchi [25] and Lander & Jones [14]. Here, $u_{\max}$ is the maximum value attained by u on the surface of the star. The function $\alpha(u)$ is chosen such that it is guaranteed that there are no currents outside the star. Since the integral of $\alpha(u)$ is related to the $\phi-$component of the magnetic field through

$$\int_0^u \alpha(u')du' = \varpi\, B_\phi, \tag{2.34}$$

we choose the lower bound of this integral such that B_ϕ is continuous. Then

$$\int_0^u \alpha(u')du' = \frac{a}{\zeta+1}(u - u_{\max})^{\zeta+1}\ \theta(u - u_{\max}). \tag{2.35}$$

Furthermore, as a simple choice (as was done in Tomimura & Eriguchi [25] and Lander & Jones [14]) we assume that the magnetic function $\kappa(u)$ is constant, so that $\kappa(u) = \kappa_0$.

2.3.2 Gravitational potential

In order to compute the gravitational potential, Poisson's equation needs to be solved. Inverting (2.2) for $\Phi_{\rm g}$ we yields

$$\Phi_{\rm g} = -G \int \frac{\rho(\mathbf{r}')}{|\mathbf{r}' - \mathbf{r}|} d^3\mathbf{r}', \tag{2.36}$$

where the primed variables denote the source points, while the non-primed the points where the field is evaluated. Using a spherical harmonics expansion for the $\frac{1}{|\mathbf{r}'-\mathbf{r}|}$ Green's function and due to the existing symmetries, the integral takes the following form (see A.2.4)

$$\Phi_{\rm g}(r,\mu) = -4\pi G \int_{r'=0}^{+\infty} dr' \int_{\mu'=0}^{1} d\mu' r'^2 \sum_{n=0}^{+\infty} f_{2n}(r', r) P_{2n}(\mu) P_{2n}(\mu') \rho(\mu', r'), \tag{2.37}$$

where $f_n(r', r)$ is given by

$$f_n(r', r) = \begin{cases} \frac{r'^n}{r^{n+1}}, & r > r' \\ \frac{r^n}{r'^{n+1}}, & r < r' \end{cases}, \tag{2.38}$$

where $P_n(\mu)$ are the Legendre polynomials.

2.3.3 Vector Potential

The Grad-Shafranov equation (2.25) can be rewritten in vector Poisson form using the Laplacian operator equivalent of the $\Delta_\star$ operator (see A.2.3.5) as

$$\boldsymbol{\nabla}^2(A_\phi \sin\phi) = -\left(\frac{\alpha(\varpi A_\phi)}{\varpi}\int_0^{\varpi A_\phi}\alpha(u)du + 4\pi\kappa(\varpi A_\phi)\rho(r,\mu)\varpi\right)\sin\phi, \tag{2.39}$$

which, solved for A_ϕ becomes

$$A_\phi \sin\phi = -\frac{1}{4\pi}\int \frac{-\frac{\alpha(\varpi' A'_\phi)}{\varpi'}\int_0^{\varpi' A'_\phi}\alpha(u)du - 4\pi\kappa(\varpi' A'_\phi)\rho(r',\mu')\varpi'}{|\mathbf{r}'-\mathbf{r}|}\sin\phi d^3\mathbf{r}'. \tag{2.40}$$

Using, again the Green's function spherical harmonic expansion as well as the assumed symmetries, we derive (see A.2.4) a simpler form for (2.40)

$$A_\phi(r,\mu) = 4\pi\int_{r'=0}^{+\infty}dr'\int_{\mu'=0}^{1}d\mu'\left(\sum_{n=1}^{+\infty}f_{2n-1}(r',r)\frac{1}{2n(2n-1)}P^1_{2n-1}(\mu)P^1_{2n-1}(\mu')\right)F(r',\mu'), \tag{2.41}$$

where $F(r',\mu')$ is the *density function* for the vector potential given by

$$F(r',\mu') = \frac{\alpha(\varpi' A'_\phi)}{4\pi\varpi'}\int_0^{\varpi' A'_\phi}\alpha(u)du + \kappa(\varpi' A'_\phi)\rho(r',\mu')\varpi', \tag{2.42}$$

while f_n is again defined by (2.38) and $P^m_l(\mu)$ are the associated Legendre polynomials.

2.3.4 Enthalpy and the first integral of the equation of motion

Integrating the equation of motion (2.1) we obtain the enthalpy

$$H = C - \Phi_{\rm g} + \Phi_{\rm r} + \int_0^{\varpi A_\phi}\kappa(u')du', \tag{2.43}$$

where C is the integration constant. Combining (2.43) with (2.7) and since we chose $\kappa(u)$ to be constant, we obtain the first integral form

$$H = C - \Phi_{\rm g} + \frac{1}{2}\Omega_0^2\varpi^2 + \kappa_0\varpi A_\phi, \tag{2.44}$$

2.3.5 Density

For the polytropic equation of state it is possible to relate the enthalpy and density algebraically (see A.1.4)

$$H = (1+N)\frac{P}{\rho} = K(1+N)\rho^{\frac{1}{N}}, \tag{2.45}$$

It follows that the relation between enthalpy and density is

$$\rho = \left(\frac{H}{K(1+N)} \right)^N , \tag{2.46}$$

and for polytropes all hydrodynamical variables (pressure, density and enthalpy) vanish at the surface of the star.

2.3.6 Keplerian velocity and mass-shedding limit

Stars can rotate only up to their mass-shedding limit, when the angular velocity at the equator reaches the Keplerian angular velocity of a free particle in circular orbit. At this limit, the pressure gradient vanishes and the centrifugal force exactly balances the gravitational force, while the surface develops a cusp. If the star would rotate any faster, material from the surface would be shed through this cusp. Hence, at the mass-shedding limit, $\Omega = \Omega_{\rm K}$, and the angular velocity at the equator thus satisfies the condition

$$\frac{\partial}{\partial r}\Phi_{\rm g} = \Omega_{\rm K}^2 r_{\rm e}, \tag{2.47}$$

where $\Omega_{\rm K}$ is the Keplerian angular velocity and $r_{\rm e}$ is the equatorial radius of the star. From (2.44) we obtain the derivative

$$\frac{\partial}{\partial r}\Phi_{\rm g} = -\frac{\partial H}{\partial r} + \Omega_0^2 r + \sqrt{1-\mu^2}A_\phi\kappa_0 + r\sqrt{1-\mu^2}\frac{\partial A_\phi}{\partial r}\kappa_0, \tag{2.48}$$

and combining (2.47) and (2.48) we find

$$\Omega_{\rm K}^2 = -\frac{1}{r_{\rm e}}\frac{\partial H}{\partial r}\bigg|_{r_{\rm e},\mu=1} + \Omega_0^2 + \frac{1}{r_{\rm e}}A_\phi(r_{\rm e},\mu=1)\kappa_0 + \sqrt{1-\mu^2}\frac{\partial A_\phi}{\partial r}\bigg|_{r_{\rm e},\mu=1}\kappa_0. \tag{2.49}$$

2.3.7 Integral quantities

Here we describe various physical quantities that characterize the star. All integrals are calculated over the volume defined by the neutron star. Since the density vanishes outside the star, we can formally extend the integration region to all space.

The mass M is defined as

$$M = \int_{\text{all space}} \rho dV, \tag{2.50}$$

where dV is the infinitesimal volume element. The moment of inertia I is similarly defined by

$$I = \int_{\text{all space}} \rho\varpi^2 dV, \tag{2.51}$$

Furthermore, we define:
angular momentum

$$J = I\Omega, \tag{2.52}$$

kinetic energy

$$T = \frac{1}{2} I\Omega^2, \tag{2.53}$$

gravitational energy

$$W = \frac{1}{2} \int_{\text{all space}} \rho \Phi_{\text{g}} dV, \tag{2.54}$$

and internal energy

$$\Pi = \int_{\text{all space}} P dV. \tag{2.55}$$

Magnetic energy is defined as

$$\mathcal{E}_{\text{mag}} = \frac{1}{8\pi} \int_{\text{all space}} B^2 dV. \tag{2.56}$$

The magnetic field extends to infinity, but in our numerical approach we use a grid of finite extent. Implementing the above definition would thus result in a non-negligible truncation error. For this reason, we use another definition for the magnetic energy (Lander & Jones [14]), which contains the density ρ, so that the integral vanishes outside the star

$$\mathcal{E}_{\text{mag}} = \int_{\text{all space}} \mathbf{r} \cdot \boldsymbol{\mathcal{L}}\, dV. \tag{2.57}$$

For the toroidal part of the magnetic energy we still use

$$\mathcal{E}^{\text{tor}}_{\text{mag}} = \frac{1}{8\pi} \int_{\text{all space}} B_\phi^2 dV, \tag{2.58}$$

since the toroidal component of the magnetic field vanishes outside the star.

2.4 Numerical Method

Here we describe the iterative numerical method used to solve the system of integral equations governing the magneto-hydrostationary equilibrium. We will describe the system of units, the plan of the method and the numerical implementation.

2.4.1 Non-dimensional units

The numerical integrations must be performed with dimensionless quantities. We choose a system of non-dimensional units, which is derived by taking as the basis for our calculations the maximum density $\rho_{\max}$, the gravitational constant G and the equatorial radius $r_{\rm e}$. In this system, the length, mass and time units are

$$[L] = r_{\rm e}, \tag{2.59}$$

$$[M] = r_{\rm e}^3 \rho_{\max}, \tag{2.60}$$

$$[T] = \frac{1}{\sqrt{G\rho_{\max}}}. \tag{2.61}$$

Therefore, we use a combination of (2.59), (2.60) and (2.61) to create the units of each quantity and divide by that combination so that all physical variables become dimensionless:

$$\hat{r} = \frac{r}{r_{\rm e}}, \tag{2.62}$$

$$\hat{\rho} = \frac{\rho}{\rho_{\max}}, \tag{2.63}$$

$$\hat{\Phi}_{\rm g} = \frac{\Phi_{\rm g}}{G r_{\rm e}^2 \rho_{\max}}, \tag{2.64}$$

$$\hat{C} = \frac{C}{G r_{\rm e}^2 \rho_{\max}}, \tag{2.65}$$

$$\hat{\Omega}^2 = \frac{\Omega^2}{G\rho_{\max}}, \tag{2.66}$$

$$\hat{H} = \frac{H}{G r_{\rm e}^2 \rho_{\max}}, \tag{2.67}$$

$$\hat{P} = \frac{P}{G r_{\rm e}^2 \rho_{\max}^2}, \tag{2.68}$$

$$\hat{A}_\phi = \frac{A_\phi}{\sqrt{G} r_{\rm e}^2 \rho_{\max}}, \tag{2.69}$$

$$\hat{M} = \frac{M}{r_{\rm e}^3 \rho_{\max}}, \tag{2.70}$$

$$\hat{\kappa} = \frac{\kappa}{\frac{\sqrt{G}}{r_{\rm e}}}, \tag{2.71}$$

$$\hat{\alpha} = \frac{\alpha}{\frac{1}{r_{\rm e}}}, \tag{2.72}$$

$$\hat{a} = \frac{a}{\frac{1}{\sqrt{G}\rho_{\rm max} r_{\rm e}^4}}. \tag{2.73}$$

Furthermore, all quantities with dimensions of energy are non-dimensionalized as

$$\hat{E} = \frac{E}{G r_{\rm e}^5 \rho_{\rm max}}. \tag{2.74}$$

The main equations remain unaltered when using the dimensionless variables. For the definition of ρ and for polytropic equations of state we can relate the dimensionless density $\hat{\rho}$ independently of the polytropic constant K as (see A.1.4)

$$\hat{\rho} = \left(\frac{H}{H_{\rm max}}\right)^N. \tag{2.75}$$

Furthermore, for the polytropic fluid case, the polytropic equation (2.6) can be written as (see A.1.5)

$$P = P_{\rm max} \hat{\rho}^{1+\frac{1}{N}}. \tag{2.76}$$

where $P_{\rm max}$ is the maximum pressure attained inside the neutron star. Hereafter, we use only the dimensionless variables.

2.4.2 Plan of method

Here we describe the structure of the code used to evaluate the desired quantities. To begin with, the polytropic index N as well as the magnetic functions $\kappa(u)$ and $\alpha(u)$ have to be specified. Apart from these quantities, the ratio of the polar $r_{\rm p}$ to the equatorial radius $r_{\rm e}$ must be determined. This ratio is the one desired to be achieved at the equilibrium state. As mentioned before (par. 2.3.5) the enthalpy vanishes at the surface of the star and hence $H(r_{\rm p}, 1) = 0$ and $H(r_{\rm e}, 0) = 0$. Combining this with (2.43) we can determine Ω_0^2 and C

$$\Omega_0^2 = 2\frac{\Phi_{\rm g}(r_{\rm e}, 0) - \Phi_{\rm g}(r_{\rm p}, 1) - \kappa_0 r_{\rm e} A_\phi(r_{\rm e}, 0)}{r_{\rm e}^2}, \tag{2.77}$$

$$C = \Phi_{\rm g}(r_{\rm e}, 0) - \Omega_0^2 \frac{r_{\rm e}^2}{2} - \kappa_0 r_{\rm e} A_\phi(r_{\rm e}, 0). \tag{2.78}$$

Main iteration

1. Assign an initial value for ρ (one can simply set $\rho(r,\mu) = 1$ inside the star).
2. Compute $\Phi_{\rm g}$ from equation (2.37).
3. Assign an initial value for A_ϕ (one can simply set $A_\phi(\rho,\mu) = 0$ everywhere).
4. Compute through (2.41) an improved value for A_ϕ using the most recent values of A_ϕ and ρ.
5. Compute Ω_0^2 and C from (2.77) and (2.78).
6. Compute the enthalpy for all points from (2.43).
7. Compute a new value for ρ from (2.75).
8. Return to the first step and use the new values for ρ and A_ϕ for the next iteration.

The iterations are repeated until the following relative differences are less than a specified value, in the range between 10^{-4} and 10^{-6}

$$\Delta_H = \left| \frac{H_{\rm max,new} - H_{\rm max,old}}{H_{\rm max,new}} \right|, \tag{2.79}$$

$$\Delta_{\Omega_0^2} = \left| \frac{\Omega^2_{0,\rm new} - \Omega^2_{0,\rm old}}{\Omega^2_{0,\rm new}} \right|, \tag{2.80}$$

$$\Delta_C = \left| \frac{C_{\rm new} - C_{\rm old}}{C_{\rm new}} \right|. \tag{2.81}$$

As a test of the global accuracy of the numerical solution, we also use the scalar virial theorem for stationary equilibrium sotutions:

$$2T + \mathcal{E}_{\rm mag} + 3\Pi + W = 0, \tag{2.82}$$

We divide by W to obtain a relative value which serves as an accuracy check:

$$VC = \frac{|2T + \mathcal{E}_{\rm mag} + 3\Pi + W|}{|W|}. \tag{2.83}$$

2.4.3 Numerical implementation

On of the properties characterizing an equilibrium model is the ratio of the polar to the equatorial radius $r_{\rm p}/r_{\rm e}$. For simplicity, we set $r_{\rm e} = 1$ and thus the ratio is defined by the polar radius only. In order to numerically solve the equations, we construct a 2D r vs. μ grid with KDIV points in the

μ-direction and NDIV points in the r-direction. The μ-direction is defined between 0 and 1 while the r-direction between 0 and $r_{\max}$ (initially, wet set $r_{\max}$=16/15.) Hence, the two arrays will be

$$r_j = r_{\max}\frac{(j-1)}{NDIV-1}, \tag{2.84}$$

and

$$\mu_i = \frac{(i-1)}{KDIV-1}. \tag{2.85}$$

For the integrals, we use Simpon's rule, which is of fourth-order accurate (see A.3.1) while for derivatives we use a four-point or a five-point stencil, of fourth order accuracy, depending on position (see A.3.2). Also, we calculate up to $n = LMAX$ terms of the Legendre and associated Legendre polynomials (although $LMAX$ can have any integer value, we obtain accurate results by setting $LMAX = 16$). Furthermore, we use extrapolation for all points on the edges of the grid ($i = 1..KDIV\ j = 1$, $i = 1..KDIV\ j = NDIV$, $i = 1\ j = 1..NDIV$, $i = KDIV\ j = 1..NDIV$) to obtain values for various quantities describing the neutron star (see A.3.4, A.3.3).

Integration of the mass density (2.37) or vector potential (2.41) is implemented in similar ways. We represent r' with subscript k, r with j, μ' with l and μ with i. Subscript n is related to the degree of the polynomials. Integrating (2.37 or 2.41) over μ' we obtain

$$W^{(1)}_{k,n} = \sum_{l=1}^{KDIV-2} \frac{1}{3(KDIV-1)}\left[P_{2n}(\mu_l)\rho_{k,l} + 4P_{2n}(\mu_{l+1})\rho_{k,l+1} + P_{2n}(\mu_{l+2})\rho_{k,l+2}\right], \tag{2.86}$$

for the mass density integration and

$$V^{(1)}_{k,n} = \sum_{l=1}^{KDIV-2} \frac{1}{3(KDIV-1)}\left[P^1_{2n-1}(\mu_l)F_{k,l} + 4P^1_{2n-1}(\mu_{l+1})F_{k,l+1} + P_{2n-1}(\mu_{l+2})F_{k,l+2}\right], \tag{2.87}$$

for the vector potential, while integration over r' gives

$$\begin{aligned} W^{(2)}_{n,j} \;=\; \sum_{k=1}^{NDIV-2} \frac{r_{\max}}{3(NDIV-1)} \Big(& r_k^2 f_{2n}(r_k, r_j) W^{(1)}_{k,n} + 4r_{k+1}^2 f_{2n}(r_{k+1}, r_j) W^{(1)}_{k+1,n} \\ & + r_{k+2}^2 f_{2n}(r_{k+2}, r_j) W^{(1)}_{k+2,n} \Big), \end{aligned} \tag{2.88}$$

for the mass density and

$$\begin{aligned} V^{(2)}_{n,j} = \sum_{k=1}^{NDIV-2} \frac{r_{\max}}{3(NDIV-1)} \Big(& r_k^2 f_{2n-1}(r_k, r_j) V^{(1)}_{k,n} + 4r_{k+1}^2 f_{2n-1}(r_{k+1}, r_j) V^{(1)}_{k+1,n} \\ & + r_{k+2}^2 f_{2n-1}(r_{k+2}, r_j) V^{(1)}_{k+2,n} \Big), \end{aligned} \tag{2.89}$$

for A_ϕ. The resulting integrals are then:

$$\Phi_{i,j} = -4\pi \sum_{n=0}^{LMAX} W_{n,j}^{(2)} P_{2n}(\mu_i), \tag{2.90}$$

and

$$A_{\phi\ i,j} = 4\pi \sum_{n=1}^{LMAX} \frac{1}{2n(2n-1)} V_{n,j}^{(2)} P_{2n-1}(\mu_i). \tag{2.91}$$

The 4π multiplication factor arises out of the $\phi-$itegration.

The integration of other quantities are implemented in the following, more simple manner. Using the previous subscript notation and assuming the integrand is a function of both (r, μ) denoted by $S_{k,l}$ ($S_{k,l}$ also contains the r'^2 term of the volume element) we first integrate over μ'

$$T_k = \sum_{l=1}^{KDIV-2} \frac{1}{3(KDIV-1)} \left(S_{k,l} + 4S_{k,l+1} + S_{k,l+2}\right), \tag{2.92}$$

creating the aforementioned quantity T_k, which depends only on r'. Then, integrating over r' we have

$$R = 4\pi \sum_{k=1}^{NDIV-2} \frac{r_{\max}}{3(NDIV-1)} \left(T_k + 4T_{k+1} + T_{k+2}\right). \tag{2.93}$$

As before, the 4π factor is due to integration over ϕ. The source code can be found in Appendix C.1.

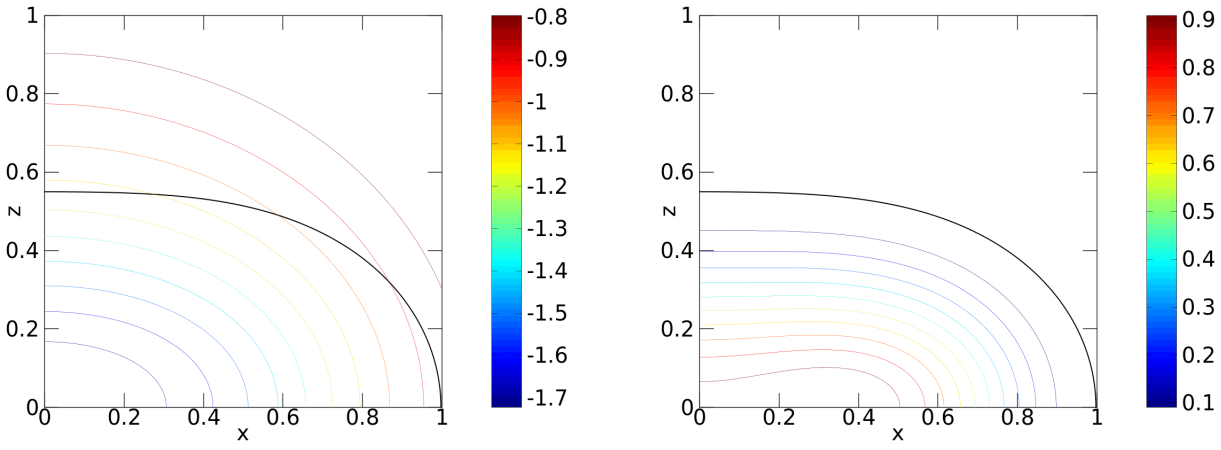
2.5 Results

As a first test of our numerical code, we use $a = 200$, $k_0 = 0.4$ and $N = 1.5$ to compare our results with Tommimura & Eriguchi [25]. The detailed comparison in shown in Table 2.1. The results are in good agreement, within $\sim 1\%$, except the fastest rotating model, where some quantities show larger disagreement (but notice that our models have significantly better virial tests). Comparisons to additional models in Tommimura & Eriguchi [25] are shown in Appendix B.1.

We also provide contour plots of the gravitational potential (Fig. 2.1a), matter density (Fig. 2.1b), poloidal (Fig. 2.2a) and toroidal (Fig. 2.2b) magnetic field norms as well as enthalpy (Fig. 2.3) for a model with $\alpha = 200$, $\kappa_0 = 0.4$, $N = 1.5$ and $r_{\rm p}/r_{\rm e} = 0.55$ constructed on a 451×451 grid. The current configuration describes a deformed neutron star due to rapid rotation and strong magnetic field (Table 2.1). Although the current configuration contains both poloidal and toroidal magnetic fields, the maximum toroidal field strenght is considerably smaller than the poloidal field strength, as the toroidal magnetic energy is about 0.2% of the total magnetic energy.

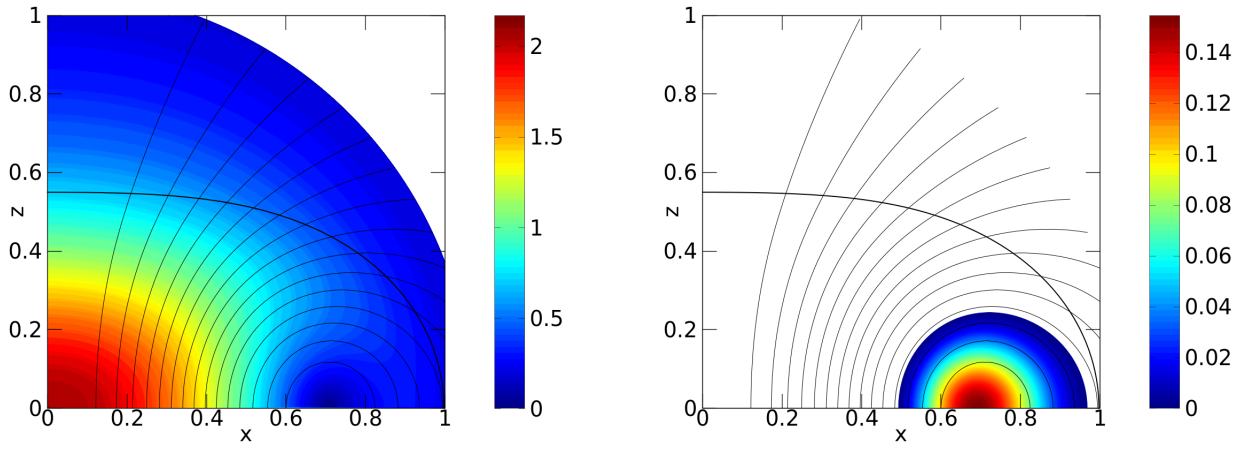
Table 2.1: Our results compared to Tomimura & Eriguchi (TE) [25], with $\alpha = 200$, $\kappa_0 = 0.4$, $N = 1.5$ (using a 751×751 grid).

Model	$r_{\rm p}/r_{\rm e}$	$\mathcal{E}_{\rm mag}/\lvert W\rvert$	$3\Pi/\lvert W\rvert$	$T/\lvert W\rvert$	$\lvert W\rvert$	Ω_0^2	C	M	Virial test
TE	0.589	0.146	0.285	3.53E-04	4.83E-02	1.50E-04	-0.0912	0.834	7.23E-05
current	0.589	0.144	0.285	4.07E-04	4.82E-02	1.73E-04	-0.0911	0.832	3.03E-06
TE	0.55	0.152	0.276	0.0106	4.61E-02	4.31E-03	-0.0919	0.812	7.61E-05
current	0.55	0.152	0.276	0.0105	4.61E-02	4.29E-03	-0.0919	0.812	3.04E-06
TE	0.5	0.166	0.264	0.0205	4.34E-02	7.76E-03	-0.0927	0.788	8.23E-05
current	0.499	0.166	0.264	0.0207	4.33E-02	7.82E-03	-0.0926	0.787	3.26E-06
TE	0.45	0.190	0.256	0.0220	4.02E-02	7.44E-03	-0.0922	0.764	9.23E-05
current	0.449	0.190	0.255	0.0220	4.01E-02	7.46E-03	-0.0922	0.763	3.60E-06
TE	0.4	0.222	0.252	0.0110	3.59E-02	3.20E-03	-0.0892	0.730	1.05E-04
current	0.4	0.223	0.252	0.0111	3.58E-02	3.21E-03	-0.0891	0.729	3.95E-06
TE	0.372	0.242	0.252	7.83E-04	3.32E-02	2.07E-04	-0.0866	0.707	1.15E-04
current	0.371	0.243	0.252	5.50E-04	3.31E-02	1.45E-05	-0.0865	0.706	4.28E-06



(a) (b)

Figure 2.1: Contours of (a) the gravitational potential $\Phi_{\rm g}$ and (b) the matter density ρ . The black line represents the surface of the star.



(a) (b)

Figure 2.2: Contours of the function u for (a) the poloidal and (b) the toroidal magnetic field components. The toroidal component is in the region where the poloidal component has its minimum value and is confined inside the star. Notice that the toroidal component is significantly weaker than the poloidal one.

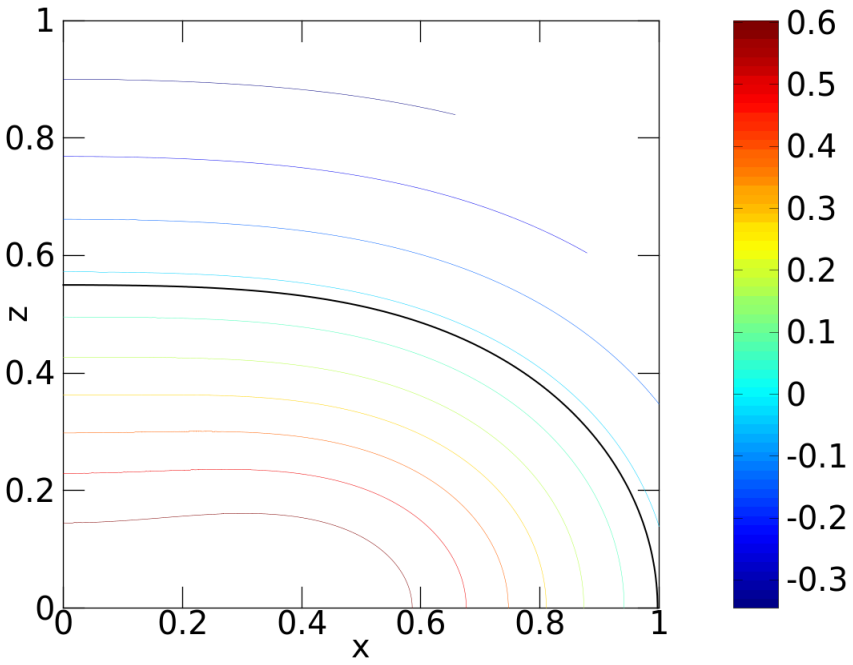


Figure 2.3: Contours of the enthalpy H (positive values), which vanishes at the surface of the star (black line). Outside the star, the contours correspond to those of a negative effective potential.

Chapter 3

Rotating magnetized superconducting neutron stars

3.1 Introduction

In this part we look into equilibrium configurations of superconducting, two-component, magnetized, barotropic, rotating neutron stars. Initially, we describe the physical system and the corresponding equations. As in the previous Chapter, following Lander [18], we use an iterative method to obtain equilibria, solving the integral form of the equations of motion. Compared to the previous Chapter, the main differences are the two-fluid composition and the different magnetic force acting on protons. For a more general account on the theoretical description of this subject, see Glampedakis, Andersson and Samuelsson [8].

3.2 Basic theory

3.2.1 Two-fluid description

We assume a rotating, magnetized, axisymetric neutron star in a stationary state. The axis of symmetry is parallel to the magnetic dipole axis, as well as to the rotational axis. The star is assumed to have only two regions: the core and the crust. The core, a region that starts at the center and extends to roughly 90% of the radius of the star (the crust-core boundary), is assumed to consist of superfluid neutrons and type-II superconducting protons, where the protons are subject to

a magnetic force different from the normal ideal MHD case[1]. Superfluidity allows for relative flows among these interpenetrating components. The crust, which lies between the crust-core boundary and the surface of the star, is composed of normal matter, which is subject to the standard Lorentz force. The crust is assumed to be in a relaxed state, without tagential stresses and can thus be described as a single fluid. Outside the star we assume vacuum.

In the core, we need to consider two separate equations for the hydrostationary equilibrium:

$$\boldsymbol{\nabla}\tilde{\mu}_{\mathrm{n}} + \boldsymbol{\nabla}\Phi_{\mathrm{g}} - \boldsymbol{\nabla}\Phi_{\mathrm{r,n}} = 0, \tag{3.1}$$

and

$$\boldsymbol{\nabla}\tilde{\mu}_{\mathrm{p}} + \boldsymbol{\nabla}\Phi_{\mathrm{g}} - \boldsymbol{\nabla}\Phi_{\mathrm{r,p}} = \frac{\mathbf{F}_{\mathrm{mag}}}{\rho_{\mathrm{p}}}, \tag{3.2}$$

where $\tilde{\mu}_{\mathrm{n}}, \tilde{\mu}_{\mathrm{p}}$ are the neutron and proton chemical potentials, Φ_{g} is the gravitational potential and $\Phi_{\mathrm{r,n}}, \Phi_{\mathrm{r,p}}$ are the rotational potentials for neutrons and protons respectively. $\mathbf{F}_{\mathrm{mag}}$ is the magnetic force acting on protons. We will discuss chemical potentials, as well as the form of the magnetic force in the following sections.

We consider that the neutrons and protons co-rotate rigidly ($\Omega_{\mathrm{n}}^2 = \Omega_{\mathrm{p}}^2 = \Omega_0^2$) and hence the rotational potential is given by

$$\Phi_{\mathrm{r,n}} = \Phi_{\mathrm{r,p}} = \frac{\varpi^2\Omega_0^2}{2}, \tag{3.3}$$

while the gravitational field is, as before, related to the total density ρ through

$$\boldsymbol{\nabla}^2\Phi = 4\pi G\rho = 4\pi G(\rho_{\mathrm{n}} + \rho_{\mathrm{p}}). \tag{3.4}$$

We subtract (3.1) from (3.2) to find

$$\boldsymbol{\nabla}(\tilde{\mu}_{\mathrm{p}} - \tilde{\mu}_{\mathrm{n}}) = \frac{\mathbf{F}_{\mathrm{mag}}}{\rho_{\mathrm{p}}}, \tag{3.5}$$

which we use along with (3.2) (instead of using (3.2) and (3.1)). The magnetic field strength $\mathbf{B}$ needs to satisfy Gauss' law for magnetism

$$\boldsymbol{\nabla}\cdot\mathbf{B} = 0, \tag{3.6}$$

and in axisymmetry it is decomposed as in (2.10)

$$\mathbf{B} = \frac{1}{\varpi}\boldsymbol{\nabla}u\times\mathbf{e}_\phi + B_\phi\mathbf{e}_\phi. \tag{3.7}$$

The system is closed with an equation of state in terms of an energy functional, which will be discussed in the next section.

[1]For simplicity, we ignore the presence of electrons, as well as more exotic particle species that may appear at high densities in the core.

3.2.2 Equation of state

In the two-fluid description, equations (3.1), (3.2) are formulated in terms of the chemical potentials $\tilde{\mu}_{\rm p}, \tilde{\mu}_{\rm n}$ instead of the pressure P. These quantities are related through

$$\boldsymbol{\nabla} P = \rho_{\rm n} \boldsymbol{\nabla} \tilde{\mu}_{\rm n} + \rho_{\rm p} \boldsymbol{\nabla} \tilde{\mu}_{\rm p}, \tag{3.8}$$

which is obvious if we multiply (3.1) with $\rho_{\rm n}$ and (3.2) with $\rho_{\rm p}$ and then add them, which results in the form (2.1) of the equation of motion for a single fluid, if the two components are corotating. We assume an equation of state in terms of the *energy functional* $\mathcal{E}(\rho_{\rm n}, \rho_{\rm p})$ given by

$$\mathcal{E} = k_{\rm n} \rho_{\rm n}^{1+\frac{1}{N_{\rm n}}} + k_{\rm p} \rho_{\rm p}^{1+\frac{1}{N_{\rm p}}}, \tag{3.9}$$

where $k_{\rm n}, k_{\rm p}$ are constants, and $N_{\rm n}, N_{\rm p}$ are the polytropic indices for neutrons and protons respectively. The form of the equation of state is similar to the previously used polytropic for the one-fluid case. The chemical potentials are defined through

$$\tilde{\mu}_{\rm p} \equiv \left. \frac{\partial \mathcal{E}}{\partial \rho_{\rm p}} \right|_{\rho_{\rm n}} = k_{\rm n} \left(1 + \frac{1}{N_{\rm n}} \right) \rho_{\rm n}^{\frac{1}{N_{\rm n}}}, \tag{3.10a}$$

and

$$\tilde{\mu}_{\rm n} \equiv \left. \frac{\partial \mathcal{E}}{\partial \rho_{\rm n}} \right|_{\rho_{\rm p}} = k_{\rm p} \left(1 + \frac{1}{N_{\rm p}} \right) \rho_{\rm p}^{\frac{1}{N_{\rm p}}}. \tag{3.10b}$$

Here we point out that in the case that the two fluids are not corotating, the energy functional will also be a function of the relative speed between the protons and neutrons $\mathbf{w}_{\rm np} \equiv \mathbf{v}_{\rm n} - \mathbf{v}_{\rm p}$, which would provide a coupling between the two fluids, a form of entrainment.

3.2.3 Superconducting core

We assume type-II superconductivity for the protons in NS core. Thus, the magnetic force is no longer the familiar Lorentz force, but is given replaced by a flux tune tension force (see Glampedakis, Andersson and Samuelsson [8])

$$\mathbf{F}_{\rm mag} = -\frac{1}{4\pi} \left[\mathbf{B} \times (\boldsymbol{\nabla} \times \mathbf{H}_{\rm c1}) + \rho_{\rm p} \boldsymbol{\nabla} \left(B \frac{\partial H_{\rm c1}}{\partial \rho_{\rm p}} \right) \right], \tag{3.11}$$

where $\mathbf{H}_{\rm c1}$ is the first critical field given by $\mathbf{H}_{\rm c1} = H_{\rm c1} \hat{\mathbf{B}}$ and $\hat{\mathbf{B}}$ is the unit tangent vector to the magnetic field ($\hat{\mathbf{B}} = \mathbf{B}/B = \hat{B}_{\varpi} \mathbf{e}_{\varpi} + \hat{B}_{\phi} \mathbf{e}_{\phi} + \hat{B}_z \mathbf{e}_z$), with B the norm of the magnetic field. The norm of the first critical field is given by

$$H_{\rm c1} = h_{\rm c} \frac{\rho_{\rm p}}{\varepsilon_\star}, \tag{3.12}$$

where $h_{\rm c}$ is some constant and $\varepsilon_\star$ is the entrainment parameter. Neglecting the coupling between protons and neutrons, we set $\varepsilon_\star = 1$ (otherwise, $\varepsilon_\star$ would be a function of particle densities and there would also be a force acting on neutrons, see Glampedakis, Andersson & Lander [7]). In this case (3.11) becomes

$$-\frac{4\pi}{h_{\rm c}}\mathbf{F}_{\rm mag} = \rho_{\rm p}\boldsymbol{\nabla}B + \mathbf{B}\times\left(\rho_{\rm p}\boldsymbol{\nabla}\times\hat{\mathbf{B}} + \boldsymbol{\nabla}\rho_{\rm p}\times\hat{\mathbf{B}}\right). \tag{3.13}$$

Next, we define a unit current as

$$\hat{\mathbf{j}} \equiv \boldsymbol{\nabla}\times\hat{\mathbf{B}} = \underbrace{\frac{1}{\varpi}\boldsymbol{\nabla}(\varpi\hat{B}_\phi)\times\mathbf{e}_\phi}_{\hat{\mathbf{j}}_{\rm pol}} + \underbrace{\hat{j}_\phi\mathbf{e}_\phi}_{\hat{\mathbf{j}}_{\rm tor}}, \tag{3.14}$$

(see A.1.1.3). Substituting (3.14) in the first term of the parenthesis in (3.13) and using (3.7) to substitute $\hat{\mathbf{B}}$ in the second term, we obtain

$$-\frac{4\pi}{h_{\rm c}}\mathbf{F}_{\rm mag} = \rho_{\rm p}\boldsymbol{\nabla}B + \frac{1}{\varpi}\mathbf{B}\times\left[\boldsymbol{\nabla}\left(\rho_{\rm p}\varpi\hat{B}_\phi\right)\times\mathbf{e}_\phi\right] + \left(\rho_{\rm p}\hat{j}_\phi - \frac{\boldsymbol{\nabla}u\cdot\boldsymbol{\nabla}\rho_{\rm p}}{\varpi B}\right)\mathbf{B}\times\mathbf{e}_\phi. \tag{3.15}$$

The last term is purely poloidal, since

$$\mathbf{B}\times\mathbf{e}_\phi = \left(\frac{1}{\varpi}\boldsymbol{\nabla}u\times\mathbf{e}_\phi\right)\times\mathbf{e}_\phi + \hat{B}_\phi\mathbf{e}_\phi\times\mathbf{e}_\phi = -\frac{1}{\varpi}\boldsymbol{\nabla}u, \tag{3.16}$$

and substituting the above result in (3.15) yields (see A.1.2.2)

$$-\frac{4\pi}{h_{\rm c}}\mathbf{F}_{\rm mag} = \rho_{\rm p}\boldsymbol{\nabla}B + \frac{1}{\varpi^2}\boldsymbol{\nabla}u\times\boldsymbol{\nabla}\left(\rho_{\rm p}\varpi\hat{B}_\phi\right) + \frac{B_\phi}{\varpi}\boldsymbol{\nabla}\left(\rho_{\rm p}\varpi\hat{B}_\phi\right) + \left(\frac{\boldsymbol{\nabla}\rho_{\rm p}\cdot\boldsymbol{\nabla}u}{\varpi B} - \rho_{\rm p}\hat{j}_\phi\right)\frac{\boldsymbol{\nabla}u}{\varpi}. \tag{3.17}$$

As in the previous section, the magnetic force does not have a $\phi-$component, since it is the gradient of an axisymmetric scalar function. Thus, any toroidal component in the right side of (3.17) should vanish. The only toroidal component in this equation is the second term on the right side and therefore

$$\boldsymbol{\nabla}u\times\boldsymbol{\nabla}\left(\rho_{\rm p}\varpi\hat{B}_\phi\right) = 0, \tag{3.18}$$

implying that either $\boldsymbol{\nabla}u$ is parallel to $\boldsymbol{\nabla}\left(\rho_{\rm p}\varpi\hat{B}_\phi\right)$ and we obtain a mixed poloidal-toroidal field, or $\boldsymbol{\nabla}u = 0$ and we obtain a purely toroidal $\mathbf{B}$ field. Here, *we only consider the mixed-field case.* Setting

$$\rho_{\rm p}\varpi\hat{B}_\phi \equiv f, \tag{3.19}$$

it follows that $f = f(u)$ (see A.2.3.6).

On the other hand, taking the curl of (3.5) shows that its right side should be the gradient of a scalar M

$$\frac{\mathbf{F}_{\text{mag}}}{\rho_{\text{p}}} = \boldsymbol{\nabla} M. \tag{3.20}$$

Equation (3.17) then becomes (see A.1.2.2)

$$-\frac{4\pi}{h_{\text{c}}}\boldsymbol{\nabla} M - \boldsymbol{\nabla} B = \left(\frac{Bf}{\rho_{\text{p}}^2 \varpi}\frac{df}{du} + \frac{\boldsymbol{\nabla}\rho_{\text{p}} \cdot \boldsymbol{\nabla} u}{\varpi B \rho_{\text{p}}} - \hat{j}_\phi\right)\frac{\boldsymbol{\nabla} u}{\varpi}. \tag{3.21}$$

Setting

$$y := \frac{4\pi}{h_{\text{c}}} M + B, \tag{3.22}$$

we have $\boldsymbol{\nabla} y = \tilde{C}\,\boldsymbol{\nabla} u$ where $\tilde{C}$ is the term in parenthesis in (3.21). Taking the cross product of both sides with $\boldsymbol{\nabla} u$ yields that $y = y(u)$ and hence

$$\frac{dy}{du} = -\frac{Bf}{\varpi^2 \rho_{\text{p}}^2}\frac{df}{du} - \frac{\boldsymbol{\nabla}\rho_{\text{p}} \cdot \boldsymbol{\nabla} u}{\varpi^2 B \rho_{\text{p}}} + \frac{\hat{j}_\phi}{\varpi}. \tag{3.23}$$

Although in the previous section we showed that M is a function of u, this assumption does not hold in the present context and thus we can not further specify the functional form of either M or B. At this point we will manipulate $\hat{j}_\phi$ in order to transform it into a more suitable form. Using (3.7, 3.14) we have (see A.1.2.3)

$$\hat{j}_\phi = \frac{1}{\varpi B}\left[\underbrace{\left(\frac{\partial^2}{\partial \varpi^2} - \frac{1}{\varpi}\frac{\partial}{\partial \varpi} + \frac{\partial^2}{\partial z^2}\right)}_{\Delta_\star} u - \frac{1}{B}\boldsymbol{\nabla} B \cdot \boldsymbol{\nabla} u\right], \tag{3.24}$$

where $\Delta_\star$ is used as in the previous part. Replacing in (3.23) we obtain the equivalent of the Grad-Shafranov equation for the superconducting matter

$$\frac{dy}{du} = -\frac{Bf}{\varpi^2 \rho_{\text{p}}^2}\frac{df}{du} - \frac{\boldsymbol{\nabla}\rho_{\text{p}} \cdot \boldsymbol{\nabla} u}{\varpi^2 B \rho_{\text{p}}} - \frac{1}{\varpi^2 B}\left(\Delta_\star u - \frac{1}{B}\boldsymbol{\nabla} B \cdot \boldsymbol{\nabla} u\right). \tag{3.25}$$

Defining $\Pi \equiv \frac{B}{\rho_{\text{p}}}$, we obtain

$$\Delta_\star u = \frac{\boldsymbol{\nabla}\Pi \cdot \boldsymbol{\nabla} u}{\Pi} - \varpi^2 \rho_{\text{p}} \Pi \frac{dy}{du} - \Pi^2 f \frac{df}{du}, \tag{3.26}$$

where we moved $\frac{\boldsymbol{\nabla}\Pi \cdot \boldsymbol{\nabla} u}{\Pi}$ to the right side. Although this term could be manipulated as an operator acting on u (since it contains $\boldsymbol{\nabla} u$), moving it to the right side and regarding it as part of the source function results in an easier numerical implementation. If one would keep $\frac{\boldsymbol{\nabla}\Pi \cdot \boldsymbol{\nabla} u}{\Pi}$ as part of the

operator acting on u, it would be necessary to find the corresponding Green's function. The norm of the magnetic field can be written in terms of u using (3.7, 3.19) as (see A.1.2.4)

$$B \equiv \sqrt{\mathbf{B}\cdot\mathbf{B}} = \rho_{\rm p}\frac{|\boldsymbol{\nabla} u|}{\sqrt{\rho_{\rm p}^2\varpi^2 - f^2}}, \tag{3.27}$$

while Π is

$$\Pi = \frac{|\boldsymbol{\nabla} u|}{\sqrt{\rho_{\rm p}^2\varpi^2 - f^2}}. \tag{3.28}$$

As in the first part, $\Delta_\star u$ can be written as a Laplacian operator (see A.2.3.5) and thus the superconducting Grad-Shafranov equation obtains the following form

$$\boldsymbol{\nabla}^2\left(\frac{u\sin\phi}{\varpi}\right) = \left(\frac{\boldsymbol{\nabla}\Pi\cdot\boldsymbol{\nabla} u}{\varpi\Pi} - \varpi\rho_{\rm p}\Pi\frac{dy}{du} - \frac{\Pi^2}{\varpi}f\frac{df}{du}\right)\sin\phi. \tag{3.29}$$

3.2.4 Normal crust and exterior

3.2.4.1 The crust

In this section we discuss the form for the **B** field in the crust region. We assume normal perfectly conducting matter and hence, the governing equations are those of the perfect MHD used in the first part. The magnetic force is

$$\mathbf{F}_{\rm mag} = \frac{1}{4\pi}(\boldsymbol{\nabla}\times\mathbf{B})\times\mathbf{B}. \tag{3.30}$$

One difference though, is that u can not be decomposed into $A_\phi\varpi$. For this reason the Grad-Shafranov equation is simply (2.26)

$$\Delta_\star u = -4\pi\varpi^2\rho_{\rm p}\frac{dM_{\rm N}}{du} - f_{\rm N}\frac{df_{\rm N}}{du}, \tag{3.31}$$

where we have denoted the functions M, f with subscript N to distinguish them from their superconducting counterparts. The Grad-Shafranov equation, written in the form of a Poisson equation, is

$$\boldsymbol{\nabla}^2\left(\frac{u\sin\phi}{\varpi}\right) = \left(-4\pi\varpi\rho_{\rm p}\frac{dM_{\rm N}}{du} - \frac{f_{\rm N}}{\varpi}\frac{df_{\rm N}}{du}\right)\sin\phi. \tag{3.32}$$

3.2.4.2 The exterior

The exterior of the star is assumed to be perfect vacuum as we have not assumed the presence of a magnetosphere. Therefore, the matter density vanishes in the outer region and the equation governing the magnetic field is

$$\Delta_\star u = \boldsymbol{\nabla}^2\left(\frac{u\sin\phi}{\varpi}\right) = 0. \tag{3.33}$$

3.2.4.3 Global magnetic equations

Gathering together the equations for all regions (core, crust and exterior), we write them in piecewise form as follows

$$\boldsymbol{\nabla}^2\left(\frac{u\sin\phi}{\varpi}\right)=\begin{cases}\left(\frac{\boldsymbol{\nabla}\Pi\cdot\boldsymbol{\nabla}u}{\varpi\Pi}-\varpi\rho_{\rm p}\Pi\frac{dy}{du}-\frac{\Pi^2}{\varpi}f\frac{df}{du}\right)\sin\phi, & \text{core},\\ \left(-4\pi\varpi\rho_{\rm p}\frac{dM_{\rm N}}{du}-\frac{f_{\rm N}}{\varpi}\frac{df_{\rm N}}{du}\right)\sin\phi, & \text{crust},\\ 0, & \text{exterior}.\end{cases}\tag{3.34}$$

3.3 Mathematical Manipulation

In this section we discuss the mathematical manipulation of the equations so that they can be implemented numerically.

3.3.0.4 Crust-core boundary conditions

In order for our model to be consistent, we need to specify the boundary conditions on the crust-core boundary. To begin with, we define the crust-core (cc) surface as

$$\rho_{\rm p\,cc}(\varpi,z)=\rho_{\rm p}(0.9\,r^{\rm p}_{\rm eq},0),\tag{3.35}$$

and the surface of the star by

$$\rho_{\rm p\,surf}(\varpi,z)=0.\tag{3.36}$$

The first boundary condition to be met is the continuity of the magnetic force on the crust-core boundary

$$\left[\rho_{\rm p}^{\rm core}\boldsymbol{\nabla}M_{\rm sc}\right]_{\rm cc}=\left[\rho_{\rm p}^{\rm crust}\boldsymbol{\nabla}M_{\rm N}\right]_{\rm cc}.\tag{3.37}$$

In the previous equation we have denoted proton density with core and crust superscripts implying that they could have different values on the corresponding regions. The presence of this discontinuity can be reasoned due to the transition from the superconducting to the normal matter region, however here we will only use continuous functions for the density and hence $\rho_{\rm p}^{\rm core}$ and $\rho_{\rm p}^{\rm crust}$ are equal. Substituting (3.22) in (3.37) we obtain (see A.1.3)

$$[\boldsymbol{\nabla}B]_{\rm cc}=\left[\left(\frac{dy}{du}-\frac{4\pi}{h_{\rm c}}\frac{\rho_{\rm p}^{\rm crust}}{\rho_{\rm p}^{\rm core}}\right)\boldsymbol{\nabla}u\right]_{\rm cc}.\tag{3.38}$$

As in the previous cases, $B_{\rm cc}=B_{\rm cc}(u)$ (see A.2.3.6). Since we do not know B as a function of u on the core-crust boundary explicitly, we will use a polynomial approximation for $B_{\rm cc}$, denoted by

$\tilde{B}_{\rm cc}(u)$. We employ a second order formula of the following form

$$\tilde{B}_{\rm cc}(u) = c_0 + c_1 u + c_2 u(u - u_{\rm cc}^{\rm eq}), \tag{3.39}$$

where c_0, c_1, c_2 are constants and $u_{\rm cc}^{\rm eq}$ is the equatorial value of u on the crust-core boundary. The constants are chosen in such a way that the polynomial values coincide with the numerical ones at the pole and equator. Since the polynomial is of second order, we also need a third point, which we choose to be at the middle of the θ direction ($\theta = \frac{\pi}{4}$ or $\mu = 0.5$). Therefore, we have

$$c_0 = B_{\rm cc}^{\rm pole}, \tag{3.40a}$$

$$c_1 = \frac{B_{\rm cc}^{\rm eq} - c_0}{u_{\rm cc}^{\rm eq}}, \tag{3.40b}$$

$$c_2 = \frac{B_{\rm cc}^{\rm mid} - c_0 - c_1 u_{\rm cc}^{\rm mid}}{u_{\rm cc}^{\rm mid}\,(u_{\rm cc}^{\rm mid} - u_{\rm cc}^{\rm eq})}. \tag{3.40c}$$

This polynomial approximation produces acceptable results, since it only induces a negligible error. Substituting (3.39) in (3.38) we obtain

$$y(u) = \tilde{B}_{\rm cc}(u) + \frac{4\pi}{h_{\rm c}} \left[\frac{\rho_{\rm p}^{\rm crust}}{\rho_{\rm p}^{\rm core}}\right]_{\rm cc} M_{\rm N}(u), \tag{3.41}$$

which relates $y(u)$ with $M_{\rm N}(u)$.

The second boundary condition suggests that B_ϕ is continuous at the crust-core boundary (see A.1.3)

$$f(u) \equiv f_{\rm sc}(u) = \left[\rho_{\rm p}^{\rm core}\right]_{\rm cc} \frac{f_{\rm N}(u)}{\tilde{B}_{\rm cc}(u)}, \tag{3.42}$$

which relates the superconducting and normal matter f functions. It is obvious that the independent functions are now two, instead of four.

Even though (3.29) does not depend explicitly on the superconducting function M, this function is used to obtain the equilibrium configurations, as will be shown later. The expression for $M_{\rm sc}$ is found be equating the left side of (3.22) with the right side of (3.41) and solving for $M_{\rm sc}$

$$M_{\rm sc} = \frac{4\pi}{h_{\rm c}} \left[y(u) - B\right]. \tag{3.43}$$

3.3.1 Assumptions

As in the first part, we use spherical polar coordinates with the symmetry axis along the positive z direction. In the previous section, we derived the relations between the superconducting magnetic functions y, f and their normal matter counterparts $M_{\rm N}$ and $f_{\rm N}$. We choose the functional form of

normal matter functions (as in Lander [18]) and define the superconducting through (3.38), (3.42) as follows

$$M_{\rm N}(u) = \kappa u, \tag{3.44}$$

and

$$f_{\rm N}(u) = a\,(u - u_{\rm int})^{\zeta+1}\,\theta(u - u_{\rm int}), \tag{3.45}$$

where κ, a and ζ are some constants and $u_{\rm int}$ is the largest u line which closes in the star (i.e. the u value on the surface of the star at the equator).

3.3.2 Gravitational potential

The gravitational potential is found using the same equation as in the previous part, by inverting the Poisson equation and using the Green's function Legendre polynomial expansion. The integrated density is the total density $\rho = \rho_{\rm p} + \rho_{\rm n}$.

3.3.3 Solving for u

In order, to solve the Poisson equation (3.34) we work as in the first part, with the difference that now can not be directly related to the vector potential. Therefore, solving for u yields,

$$u(r,\mu) = -\frac{\varpi}{4\pi\sin\phi}\int_{\text{all space}}\frac{F(\mathbf{r}')}{|\mathbf{r}'-\mathbf{r}|}\sin\phi' dV', \tag{3.46}$$

where $F(\mathbf{r})$ is defined through

$$F = \begin{cases} \left(-\frac{\boldsymbol{\nabla}\Pi\cdot\boldsymbol{\nabla}u}{\varpi\Pi} + \varpi\rho_{\rm p}\Pi\frac{dy}{du} + \frac{\Pi^2}{\varpi}f\frac{df}{du}\right)\sin\phi, & \text{core,} \\ \left(4\pi\varpi\rho_{\rm p}\frac{dM_{\rm N}}{du} + \frac{f_{\rm N}}{\varpi}\frac{df_{\rm N}}{du}\right)\sin\phi, & \text{crust,} \\ 0, & \text{exterior.} \end{cases} \tag{3.47}$$

Inverting (3.46) yields (see A.2.4)

$$u(r,\mu) = \varpi\int_{r'=0}^{+\infty}dr'\int_{\mu'=0}^{1}d\mu'\left[\sum_{n=1}^{+\infty}f_{2n-1}(r',r)\frac{1}{2n\,(2n-1)}P^1_{2n-1}\,(\mu)\,P^1_{2n-1}\left(\mu'\right)\right]F(r',\mu'). \tag{3.48}$$

3.3.4 Integral equations

Equations (3.2) and (3.5) can be written in integral form using (3.20) as

$$\tilde{\mu}_{\rm p} + \Phi_{\rm g} - \frac{\varpi^2\Omega_0^2}{2} - M - C_{\rm p} = 0, \tag{3.49}$$

and

$$\tilde{\mu}_{\rm p} - \tilde{\mu}_{\rm n} - M - C_{\rm dif} = 0, \tag{3.50}$$

where M is the following piecewise function, depending on the region

$$M = \begin{cases} \frac{4\pi}{h_{\rm c}} \left(y(u) - B \right), & \text{core}, \\ \kappa u, & \text{crust}. \end{cases} \tag{3.51}$$

3.3.5 Keplerian Velocity

The Keplerian velocity has the same physical meaning as in the previous part, defined by

$$\frac{\partial}{\partial r}\Phi_{\rm g} = \Omega_{\rm K}^2 r_{\rm eq}^{\rm p}, \tag{3.52}$$

where $r_{\rm eq}^{\rm p}$ is the equatorial radius of the protons (i.e. the equatorial radius of the star) and substituting (3.2) we obtain

$$\frac{\partial}{\partial r}\Phi_{\rm g} = \left.\frac{\partial M}{\partial r}\right|_{r_{\rm eq}^{\rm p}} + \Omega_0^2 r_{\rm eq}^{\rm p} - \left.\frac{\partial \tilde{\mu}_{\rm p}}{\partial r}\right|_{r_{\rm eq}^{\rm p}}. \tag{3.53}$$

Hence, $\Omega_{\rm K}^2$ is

$$\Omega_{\rm K}^2 = -\frac{1}{r_{\rm eq}^{\rm p}} \left.\frac{\partial \tilde{\mu}_{\rm p}}{\partial r}\right|_{r_{\rm eq}^{\rm p}} + \Omega_0^2 + \frac{1}{r_{\rm eq}^{\rm p}} \left.\frac{\partial M}{\partial r}\right|_{r_{\rm eq}^{\rm p}}. \tag{3.54}$$

3.3.6 Various physical quantities

As in the first part, we calculate some quantities that describe the equilibrium. The total mass M, moment of inertia I, angular momentum J, kinetic energy T, gravitational energy W are given by equations (2.50) to (2.54). The internal energy $U_{\rm p}$ for protons and $U_{\rm n}$ for neutrons are given by

$$U_{\rm p} = \int_{\text{all space}} \mathcal{E}_{\rm p}\, dV, \tag{3.55}$$

and

$$U_{\rm n} = \int_{\text{all space}} \mathcal{E}_{\rm n}\, dV, \tag{3.56}$$

where $\mathcal{E}_{\rm p}$, $\mathcal{E}_{\rm n}$ are the two terms entering the equation of state, i.e.

$$\mathcal{E}_{\rm p} = k_{\rm p} \rho_{\rm p}^{1+\frac{1}{N_{\rm p}}}, \tag{3.57}$$

and

$$\mathcal{E}_{\rm n} = k_{\rm n} \rho_{\rm n}^{1+\frac{1}{N_{\rm n}}}, \tag{3.58}$$

respectively. The magnetic energy is evaluated using

$$\mathcal{E}_{\rm mag} = \int_{\text{all space}} \mathbf{r} \cdot \mathbf{F}_{\rm mag}\, dV, \tag{3.59}$$

where $\mathbf{F}_{\rm mag}$ is given by (3.11) for the core and by (3.30) for the crust.

3.4 Numerical Method

In this section we describe the implementation of the iterative method for computing the equilibrium mdoels model. First, we describe the non-dimensional units, then the plan of the method and finally the numerical implementation.

3.4.1 Non-dimensional Units

We define the length, mass and time units through

$$[L] = r_{\rm eq}^{\rm p}, \tag{3.60}$$

$$[M] = \left(r_{\rm eq}^{\rm p}\right)^3 \rho_{\rm max}, \tag{3.61}$$

$$[T] = \frac{1}{\sqrt{G\rho_{\rm max}}}, \tag{3.62}$$

where $\rho_{\rm max} = (\rho_{\rm p})_{\rm max} + (\rho_{\rm n})_{\rm max}$. Using the appropriate combination of the aforementioned definitions, we derive the dimensionless form of the following quantities

$$\hat{r} = \frac{r}{r_{\rm eq}^{\rm p}}, \tag{3.63}$$

$$\hat{\rho}_{\rm p} = \frac{\rho_{\rm p}}{\rho_{\rm max}}, \tag{3.64}$$

$$\hat{\rho}_{\rm n} = \frac{\rho_{\rm n}}{\rho_{\rm max}}, \tag{3.65}$$

$$\hat{\Phi}_{\rm g} = \frac{\Phi_{\rm g}}{G\left(r_{\rm eq}^{\rm p}\right)^2 \rho_{\rm max}}, \tag{3.66}$$

$$\hat{C}_{\rm p} = \frac{C_{\rm p}}{G\left(r_{\rm eq}^{\rm p}\right)^2 \rho_{\rm max}}, \tag{3.67}$$

$$\hat{C}_{\rm dif} = \frac{C_{\rm dif}}{G\left(r_{\rm eq}^{\rm p}\right)^2 \rho_{\rm max}}, \tag{3.68}$$

$$\hat{\Omega}^2 = \frac{\Omega^2}{G\rho_{\rm max}}, \tag{3.69}$$

$$\hat{\tilde{\mu}}_{\rm p} = \frac{\tilde{\mu}_{\rm p}}{G\left(r_{\rm eq}^{\rm p}\right)^2 \rho_{\rm max}}, \tag{3.70}$$

$$\hat{\tilde{\mu}}_{\rm n} = \frac{\tilde{\mu}_{\rm n}}{G\left(r_{\rm eq}^{\rm p}\right)^2 \rho_{\rm max}}, \tag{3.71}$$

$$\hat{u} = \frac{u}{\sqrt{G}\left(r_{\rm eq}^{\rm p}\right)^3 \rho_{\rm max}}, \tag{3.72}$$

$$\hat{M} = \frac{M}{\left(r_{\rm eq}^{\rm p}\right)^3 \rho_{\max}}, \tag{3.73}$$

$$\hat{\kappa} = \frac{\kappa}{\frac{G}{r_{\rm eq}^{\rm p}}}, \tag{3.74}$$

$$\hat{a} = \frac{\kappa}{\frac{1}{\sqrt{G}\rho_{\max}\left(r_{\rm eq}^{\rm p}\right)^4}}, \tag{3.75}$$

while for all quantities with energy units the dimensionless form is

$$\hat{E} = \frac{E}{G\left(r_{\rm eq}^{\rm p}\right)^5 \rho_{\max}}. \tag{3.76}$$

Similarly to the first part, the equations are exactly the same when we use the dimensionless quantities. Although we have defined the dimensionless densities $\hat{\rho}_{\rm p}$, $\hat{\rho}_{\rm n}$ through (3.64) and (3.65) we can also define them using the proton and neutron fractions, which are given by $x_{\rm p} = \frac{\rho_{\rm p}}{\rho}$ and $x_{\rm n} = \frac{\rho_{\rm n}}{\rho}$ respectively and are related through $x_{\rm p} + x_{\rm n} = 1$, or

$$x_{\rm n} = 1 - x_{\rm p}. \tag{3.77}$$

Since the maximum density is attained for both protons and neutrons at the center of the star, the central proton and neutron fractions, denoted by $x_{\rm p}(0)$ and $x_{\rm n}(0)$, are related to the densities through

$$x_{\rm p}(0) = \frac{\rho_{\rm p\,max}}{\rho_{\max}}, \tag{3.78}$$

and

$$x_{\rm n}(0) = 1 - x_{\rm p}(0) = \frac{\rho_{\rm n\,max}}{\rho_{\max}}. \tag{3.79}$$

Dividing the proton chemical potential with the maximum value $\tilde{\mu}_{\max}$ and using (3.10a), (3.10b), (3.78) and (3.79) we obtain

$$\hat{\rho}_{\rm p} = x_{\rm p}(0)\left(\frac{\tilde{\mu_{\rm p}}}{\tilde{\mu}_{\rm p\,max}}\right)^{N_{\rm p}}, \tag{3.80}$$

and similarly

$$\hat{\rho}_{\rm n} = (1 - x_{\rm p}(0))\left(\frac{\tilde{\mu_{\rm n}}}{\tilde{\mu}_{\rm n\,max}}\right)^{N_{\rm p}}. \tag{3.81}$$

As in the first part, hereafter we will use only dimensionless variables, omitting the notation " ˆ ".

3.4.2 Plan of the method

In this section we discuss the algorithm for evaluating the various quantities. We specify the polytropic indices $N_{\rm p}$, $N_{\rm n}$ and the ratio of polarto equatorial radii for protons $r^{\rm p}_{\rm pol}/r^{\rm p}_{\rm eq}$, which is simply equal to $r^{\rm p}_{\rm pol}$, since the dimensionless value of $r^{\rm p}_{\rm eq}=1$. The equatorial radius of neutrons $r^{\rm n}_{\rm eq}$ is determined through the definition of the crust-core surface (3.35) and coincides with the ratio $r^{\rm n}_{\rm eq}/r^{\rm p}_{\rm eq}$. The constants κ and a for (3.45) and (3.30) and the superconductivity parameter $h_{\rm c}$, as well as the central proton fraction $x_{\rm p}(0)$ are specified. Finally, we may need to specify the under-relaxation parameter $\omega<1$ (see A.3.5) so that our scheme converges.

As in the first part, the proton chemical potential vanishes at the surface of the star, while the neutron chemical potential vanishes at the crust-core surface. Evaluating $\tilde{\mu}_{\rm p}$ at $r^{\rm p}_{\rm eq}$, $r^{\rm p}_{\rm pol}$ (where $\tilde{\mu}_{\rm p}(r^{\rm p}_{\rm eq},0)\equiv\tilde{\mu}_{\rm p}(1,0)=0$, $\tilde{\mu}_{\rm p}(r^{\rm p}_{\rm pol},1)=0$ respectively) and $\tilde{\mu}_{\rm n}$ at $r^{\rm n}_{\rm eq}$ (where $\tilde{\mu}_{\rm n}(r^{\rm n}_{\rm eq},0)=0$) we derive the equation for the angular velocity Ω_0^2

$$\Omega_0^2=2\left[\Phi_{\rm g}(1,0)-M(1,0)+M(r^{\rm p}_{\rm pol},1)-\Phi_{\rm g}(r^{\rm p}_{\rm pol},1)\right], \tag{3.82}$$

and for the integration constants $C_{\rm p}$ and $C_{\rm dif}$

$$C_{\rm p}=\Phi_{\rm g}(1,0)-M(1,0)-\frac{\Omega_0^2}{2}, \tag{3.83}$$

$$C_{\rm dif}=\tilde{\mu}_{\rm p}(r^{\rm n}_{\rm eq},0)-M(r^{\rm n}_{\rm eq},1). \tag{3.84}$$

Therefore, the main iteration algorithm is:

Main iteration

1. Assign initial values $\rho_{\rm p}=1$, $\rho_{\rm n}=1$ and $u=1$.
2. Compute $\Phi_{\rm g}$ from (2.36).
3. Calculate Π from u (3.28).
4. Compute intermediate u from (3.48). Before evaluating, divide by $\Pi_{\rm max}$ and multiply again after integration.
5. Employ under-relaxation to find the new u (A.64).
6. Evaluate the angular velocity Ω_0^2, and the proton integration constant $C_{\rm p}$ from (3.82) and (3.83).

7. Use the proton integral equation (3.55) to evaluate $\tilde{\mu}_{\rm p}$.

8. Evaluate the difference integration constant $C_{\rm dif}$ from (3.84).

9. Using the difference integral equation (3.50), compute $\tilde{\mu}_{\rm n}$.

10. Compute new proton and neutron densities from (3.80) and (3.81).

11. Return to first step and use for the next iteration the new values of $\rho_{\rm p}$, $\rho_{\rm n}$ and u.

The aforementioned algorithm is repeated until

$$\Delta_{\tilde{\mu}_{\rm p}} = \left| \frac{\tilde{\mu}_{\rm p\ max,new} - \tilde{\mu}_{\rm p\ max,old}}{\tilde{\mu}_{\rm p\ max,new}} \right|, \tag{3.85}$$

$$\Delta_{\tilde{\mu}_{\rm n}} = \left| \frac{\tilde{\mu}_{\rm n\ max,new} - \tilde{\mu}_{\rm n\ max,old}}{\tilde{\mu}_{\rm n\ max,new}} \right|, \tag{3.86}$$

$$\Delta_{C_{\rm p}} = \left| \frac{C_{\rm p\ ,new} - C_{\rm p,old}}{C_{\rm p,new}} \right|, \tag{3.87}$$

$$\Delta_{C_{\rm dif}} = \left| \frac{C_{\rm dif\ ,new} - C_{\rm dif,old}}{C_{\rm dif,new}} \right|, \tag{3.88}$$

$$\Delta_{\Omega_0^2} = \left| \frac{\Omega^2_{0,\rm new} - \Omega^2_{0,\rm old}}{\Omega^2_{0,\rm new}} \right|, \tag{3.89}$$

are all less than a specified value (usually chosen between 10^{-4} and 10^{-6}). The scalar virial theorem is

$$\frac{1}{2}\frac{d^2 I}{dt^2} = 2T + \mathcal{E}_{\rm mag} + W + 3\frac{U_{\rm n}}{N_{\rm n}} + 3\frac{U_{\rm p}}{N_{\rm p}}, \tag{3.90}$$

and since the moment of inertia is constant with respect to time, the right-hand part is equal to zero. Numerically this quantity will never vanish, so we construct the virial test quantity,

$$VC = \frac{\left| 2T + \mathcal{E}_{\rm mag} + W + 3\frac{U_{\rm n}}{N_{\rm n}} + 3\frac{U_{\rm p}}{N_{\rm p}} \right|}{|W|}, \tag{3.91}$$

which provides a test for the global convergence of the algorithm.

3.4.3 Numerical implementation

The numerical implementation is the same as in the first part. We employ a 2D r versus μ grid with NDIV, KDIV points in the respective directions. The r and μ points are given by (2.84) and (2.85) and we compute $n = LMAX$ terms of the Legendre and associated Legendre

polynomials. The numerical evaluation of the gravitational potential is given by (2.86), (2.88) and (2.90), while the integration of the other quantities is obtained by (2.92) and (2.93). As before, we extrapolate quantities on points that are on the edges of the grid ($i = 1..KDIV$, $j = 1$; $i = 1..KDIV$, $j = NDIV$; $i = 1$, $j = 1..NDIV$; $i = KDIV$, $j = 1..NDIV$). We also extrapolate the magnetic density function $F(r, \mu)$ (3.47) on the crust core boundary, to obtain values for various quantities describing the star (see A.3.4, A.3.3). The only difference is in the equation for obtaining u. Using the sa me notation as in (2.4.3), integrating (3.46) over μ' yields

$$U^{(1)}_{k,n} = \sum_{l=1}^{KDIV-2} \frac{1}{3(KDIV-1)} \left(P^1_{2n-1}(\mu_l)F_{k,l} + 4\,P^1_{2n-1}(\mu_{l+1})F_{k,l+1} + P^1_{2n-1}(\mu_{l+2})F_{k,l+2}\right), \tag{3.92}$$

while integration over r' gives

$$U^{(2)}_{n,j} = \sum_{k=1}^{NDIV-2} \frac{r_{\max}}{3(NDIV-1)} \left(r_k^2 f_{2n-1}(r_k, r_j)U^{(1)}_{k,n} + 4\,r_{k+1}^2 f_{2n-1}(r_{k+1}, r_j)U^{(1)}_{k+1,n} + r_{k+2}^2 f_{2n-1}(r_{k+2}, r_j)U^{(1)}_{k+2,n}\right). \tag{3.93}$$

Then, u is given by

$$u_{i,j} = r_j\sqrt{1-\mu_i^2} \sum_{n=1}^{LMAX} \frac{1}{2n\,(2n-1)} U^{(2)}_{n,j} P_{2n-1}(\mu_i). \tag{3.94}$$

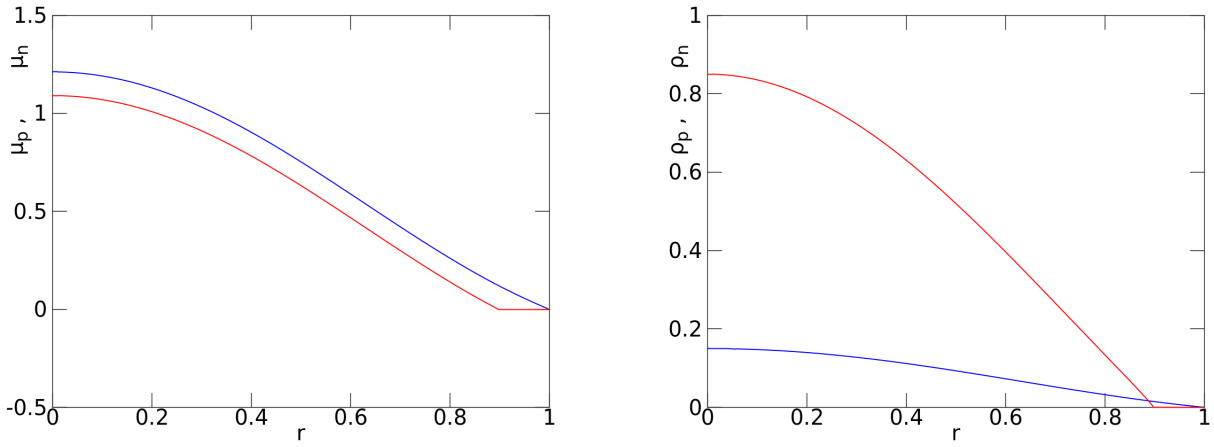
The full source code of the implementation of the numerical scheme can be found in (C.2).

3.5 Results

We focus on nonrotating equilibria and provide contours of the proton and neutron densities $\rho_{\rm p}$, $\rho_{\rm n}$, chemical potentials $\tilde{\mu}_{\rm p}$, $\tilde{\mu}_{\rm n}$, gravitational field $\Phi_{\rm g}$ as well as of the norms of the poloidal and toroidal magnetic field components, together with along with contours of the function u. The qualitative behavior of the magnetic field lines is dependent on the strength of the magnetic field, so we have "strong field", "medium field" and "weak field" configurations. In all three cases, the stars has $\rho_{\max} = 10^{15}\,{\rm gr/cm}^3$, $r_{\rm e} = 15\,{\rm km}$, $N_{\rm p} = 1$, $\alpha = 200$, $\zeta = 1$, a central proton fraction $x_{\rm p}(0) = 0.15$ and a superconductivity constant of $h_{\rm c} = 0.1$. Notice that the toroidal component is much weaker than the poloidal one.

3.5.1 Strong field case

The strong field configuration is the one most similar to the normal matter configurations. The neutron polytropic index is $N_{\rm n} = 0.9$ and $\kappa = 0.03$. Chemical potentials are shown in Fig. 3.1a, while Fig. 3.1b. displays proton and neutron densities. The chemical potential $\tilde{\mu}_{\rm p}$ vanishes at the surface of the star while $\tilde{\mu}_{\rm n}$ vanishes at the crust core boundary. The gravitational field is shown in Fig. 3.2. Since all quantities in Fig. 3.1, 3.2 exhibit symmetry with respect to θ we show the various quantities at $\theta = 45^o$. D The norms of the poloidal and toroidal magnetic field components are shown as density plots in Fig. 3.3, which also displays contours of the function u. The contours are smoot h. The norm of the magnetic field at the pole is 1.09×10^{15} G. A numerical grid of 481×481 was used, with an under-relaxation parameter $\omega = 0.023$ and the computation was stopped after 238 iterations, when all five accuracy measures (3.85)-(3.89) became less than 10^{-5}. A similar case is shown in Fig. 3 in Lander [18]. The virial test for this configuration is 3.53×10^{-6}.



(a) (b)

Figure 3.1: (a) Chemical potential of protons, $\tilde{\mu}_{\rm p}$ (blue) and of neutrons $\tilde{\mu}_{\rm n}$ at $\theta = 45^o$. The surface of the star is at $r = 1$ while the assumed crust-core boundary is at $r = 0.9$. (b) Proton density $\rho_{\rm p}$ (blue) and neutron density $\rho_{\rm n}$ (red).

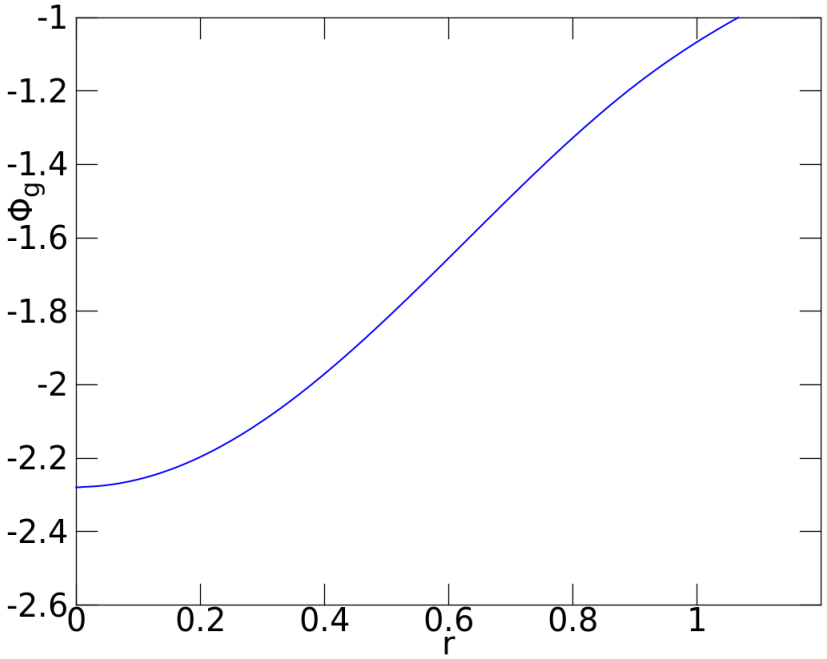
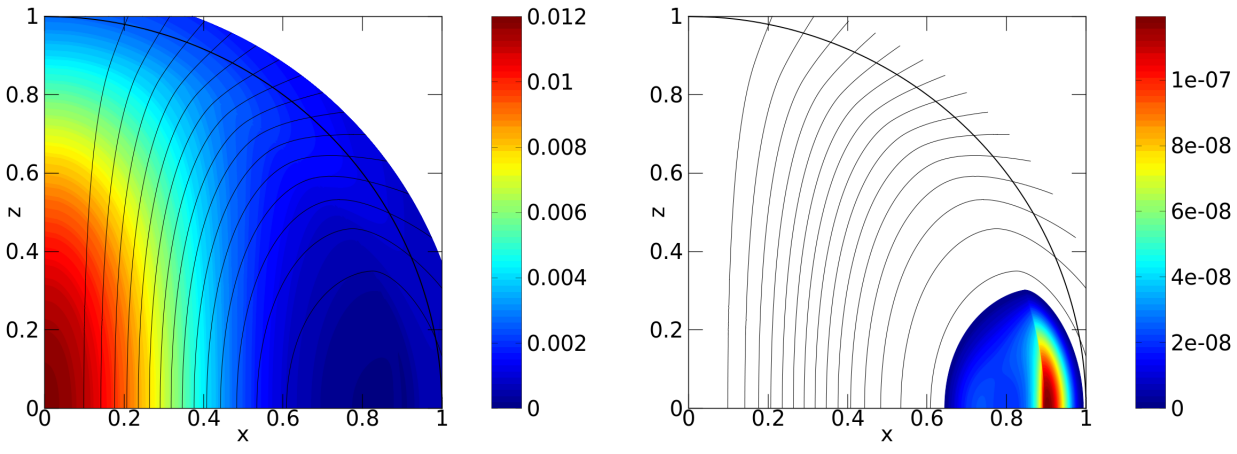


Figure 3.2: Gravitational potential $\Phi_{\rm g}$.

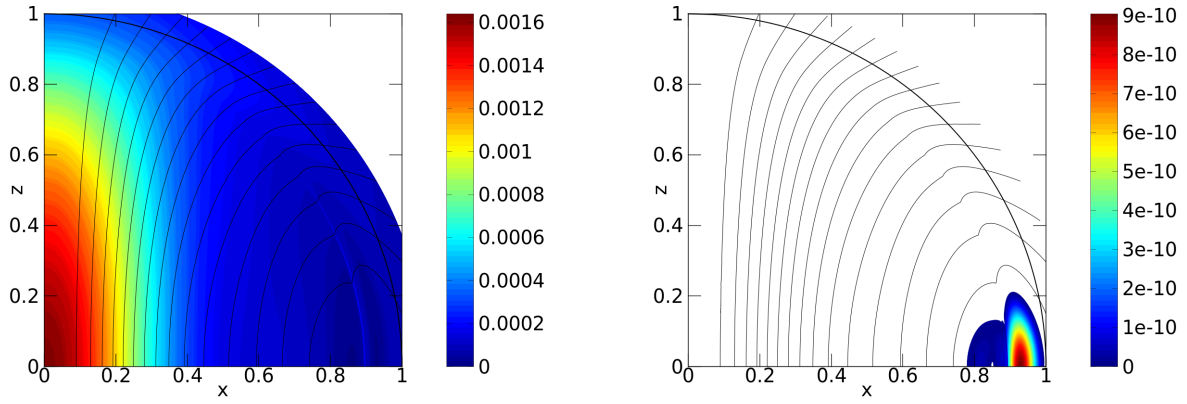


(a) (b)

Figure 3.3: (a) Density plot of the norm of the poloidal magnetic field component and (b) of the norm of the toroidal field component. The black lines are contours of the function u. The crust-core boundary is not shown in these plots.

3.5.2 Medium field case

For a medium field configurations, we choose a neutron polytropic index $N_{\rm n} = 1$ and $\kappa = 0.007$, which gives $B_{\rm pole} = 1.39 \times 10^{14}\,{\rm G}$. Fig. 3.4 shows density plots of the norms of the poloidal and toroidal magnetic field components. The region of closed field lines is displaced towards the surface of the star, compared to the strong field case. A numerical grid of 481×481 was used, with an under-relaxation parameter $\omega = 0.17$ and the computation was stopped after 47 iterations, when all five accuracy measures (3.85)-(3.89) became less than 10^{-6} while the virial test is 3.85×10^{-6}. In this case, one can clearly notice kinks in the contours of u. A similar case is shown in Fig. 3 in Lander [18].

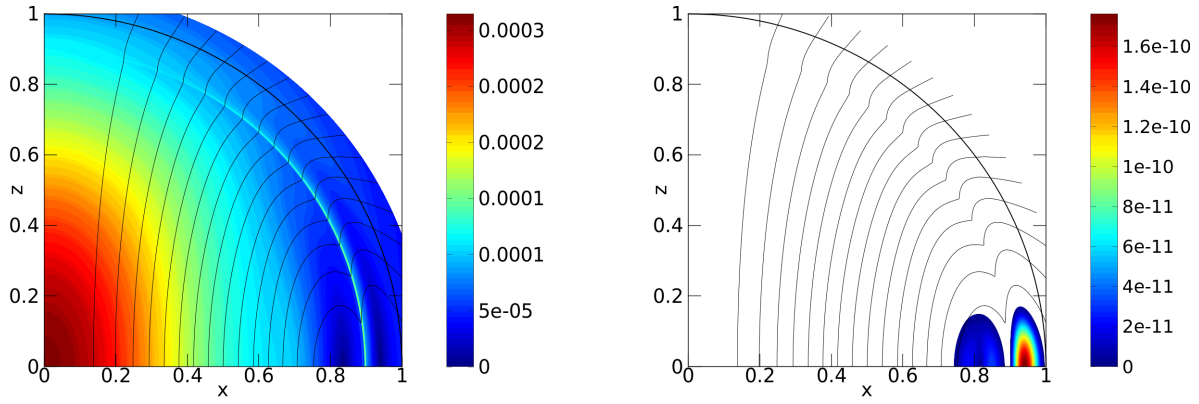


(a) (b)

Figure 3.4: (a) Density plot of the norm of the poloidal magnetic field component and (b) of the norm of the toroidal field component, for the medium field case. The black lines are contours of the function u. The region of closed field lines is displaced towards the surface of the star, compared to the strong field case. When the computation is stopped at 47 iterations, noticeable kinks are present in the contours of u. The crust-core boundary is not shown in these plots.

3.5.3 Weak field case

For a weak field configuration, we choose $N_{\rm n} = 0.9$ and $\kappa = 0.005$, which gives $B_{\rm pole} = 3.32 \times 10^{13}$ G. A numerical grid of 511×511 was used, with an under-relaxation parameter $\omega = 0.25$ and the computation was stopped after 31 iterations, when all five accuracy measures (3.85)-(3.89) became less than 10^{-6}. In this case, Fig. 3.5 shows very strong kinks in the contours of u. Here the virial test value is 4.53×10^{-6}. A similar case is shown in Fig. 3 in Lander [18].



(a) (b)

Figure 3.5: (a) Density plot of the norm of the poloidal magnetic field component and (b) of the norm of the toroidal field component, for the weak field case. The black lines are contours of the function u. The region of closed field lines is displaced towards the surface of the star, compared to the strong field case. When the computation is stopped at 31 iterations, strong kinks are present in the contours of u. The crust-core boundary is not shown in these plots.

3.6 Convergence Tests

Here we examine the convergence properties of the numerical scheme. Although the virial test provides a good measure of the global accuracy of the numerical solution, it can hide non-convergence of the magnetic field, in the case that the magnetic energy is much smaller than the other energies involved in the virial theorem. Here we provide two additional tests, specific to the magnetic field, that can uncover potential inaccuracies of the numerical solution.

A first test is provided by the violation of the divergence-free constraint $\boldsymbol{\nabla} \cdot \mathbf{B} = 0$. The finite numerical precision produces a non-zero result (which can be initially quite large at some grid points), which would correspond to the existence of a "magnetic charge" density. Using (2.10) and (A.30) we obtain

$$\boldsymbol{\nabla} \cdot \mathbf{B} = \frac{1}{r^2} \left(\frac{\partial^2 u}{\partial \mu \partial r} - \frac{\partial^2 u}{\partial r \partial \mu} \right) = 4\pi \rho_{\rm mag}^{\rm num}. \tag{3.95}$$

Integrating (3.95) within the computational grid, we obtain the total numerical "magnetic charge" $Q_{\rm mag}^{\rm num}$ as

$$Q_{\rm mag}^{\rm num} = \int_{\mu=0}^{1} \int_{r=0}^{RMAX} \left(\frac{\partial^2 u}{\partial \mu \partial r} - \frac{\partial^2 u}{\partial r \partial \mu} \right) dr\, d\mu, \tag{3.96}$$

and we examine $\log Q_{\rm mag}^{\rm num}$ so that very small deviations from zero become apparent. The magnetic

charge units are the same as the magnetic flux units $\Phi_{\rm B}$. The magnetic flux through an open surface is given by

$$\Phi_{\rm B} = \int\int_{\rm surface} \mathbf{B} \cdot d\mathbf{S}. \tag{3.97}$$

We calculate the magnetic flux on upper hemisphere of the star in order to compare it with the total magnetic charge. $\Phi_{\rm B}$ through the upper hemisphere surface of the star is (see A.3.6)

$$\Phi_{\rm B} = 2\pi \int_{\mu=0}^{1} B_r(r_{\rm e}, \mu) d\mu. \tag{3.98}$$

The relations between the dimensionless and the physical units are

$$\hat{Q}_{\rm mag}^{\rm num} = \frac{Q_{\rm mag}^{\rm num}}{\sqrt{G}\rho_{\rm max} r_{\rm e}^3}, \tag{3.99}$$

$$\hat{\Phi}_{\rm B} = \frac{\Phi_{\rm B}}{\sqrt{G}\rho_{\rm max} r_{\rm e}^3}. \tag{3.100}$$

As a second test, we use the function u, which is the fundamental quantity describing the magnetic field. We test its convergence between consecutive iterations by computing the measure

$$\sigma_u = \sqrt{\frac{1}{NDIV \cdot KDIV} \sum_{i=1}^{KDIV} \sum_{j=1}^{NDIV} \left(u_{i,j\ \rm new} - u_{i,j\ \rm old}\right)^2}, \tag{3.101}$$

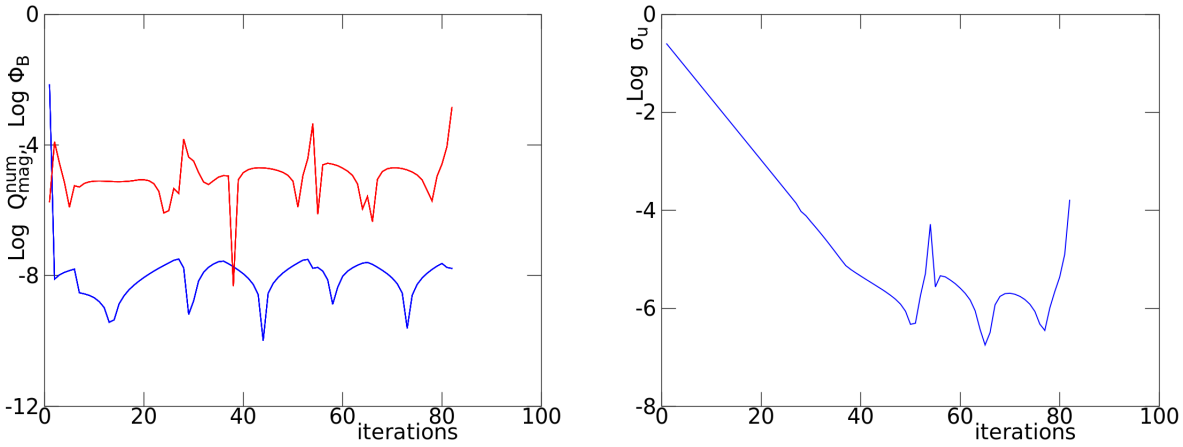
which is motivated by the definition of the usual standard deviation. The deviation of the above measure from zero is a strong indication for the convergence of the magnetic part of the numerical solution.

3.6.1 Higher convergence results

Here we focus on the weak field case, which was stopped at 31 iterations in Section 3.6 and show the dependence of the new tests s and $Q_{\rm mag}^{\rm num}$ on the number of iterations. We also show the $\log \Phi_{\rm B}$ together with $\log Q_{\rm mag}^{\rm num}$. Fig. 3.6a shows that the integrated numerical "magnetic charge" $Q_{\rm mag}^{\rm num}$ decreases rapidly with increasing number of iterations and reaches a plateau (set by the finite accuracy of the grid spacing) at extremely small values within a few iterations. Also, it is apparent that $\Phi_{\rm B}$ is about four orders of magnitude larger and thus the $Q_{\rm mag}^{\rm num}$ is negligible. This shows that the divergence-free property is strongly enforced by the iterative scheme and is preserved during a large number of iterations.

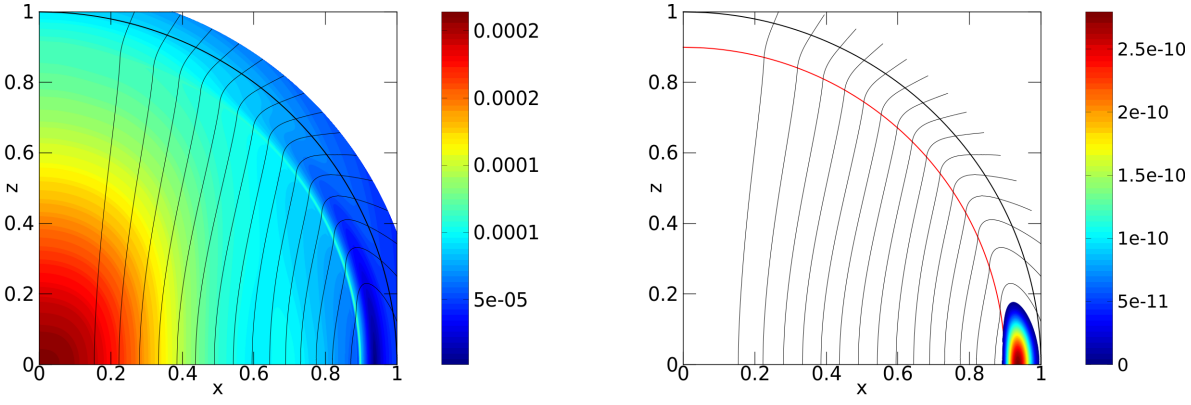
Fig. 3.6b shows the convergence of the logarithm of the "standard deviation" σ_u with increasing number of iterations. Its value decreases as a power law while the least square line fit for underrelaxation parameter $\omega = 0.25$ is $\log \sigma_u = -0.494 - 0.124\, n$, where n is the number of iterations. The

slope of the least square line is related to the underrelaxation parameter since it becomes steeper i.e. the algorithm converges faster, when the underrelaxation parameter is increased until it reaches a plateau at around 40 iterations. This indicates that the accuracy criteria used in Section 3.6 (which concerned the matter fields) terminated the computation at 31 iterations, while there was still room for improving the accuracy of the solution and in particular its magnetic part (the matter fields are only weakly coupled to the magnetic field in the weak field case). Because the matter fields have already converged to sufficient accuracy after 31 iterations and are weakly coupled to the magnetic field, we freeze the matter fields at that iteration number and continue to iterate only the equation for the magnetic field for ten more iterations. At 41 iterations the magnitude of the magnetic field at the pole is $B_{\rm pole} = 3.88 \times 10^{13}\,{\rm G}$, somewhat higher than before.



(a) (b)

Figure 3.6: (a)Logarithm of numerical "magnetic charge" $Q^{\text{num}}_{\text{mag}}$ (blue) and logarithm of total upper hemisphere magnetic flux Φ_{B} (red). (b) Logarithm of standard deviation-like quantity σ_u.



(a) (b)

Figure 3.7: Contours of the poloidal (a) and the toroidal (b) field along with the u contours (black lines) for the higher convergence weak field case.

The resulting magnetic field structure is shown in Fig. 3.7. It is obvious that the additional iterations have smoothed out the kinks that were present in the contours of u in the numerical solution shown in Fig. 3.5. In addition, the toroidal field region has now moved entirely inside the crust region. *This demonstrates that in the weak field case the magnetic field (and especially its toroidal part) needs additional iterations to relax to a converged solution, compare to the matter fields.*

For the converged weak field case, we expand the numerical grid to $RMAX = 3\,r_{\rm e}$. Thre resulting external vacuum configuration has a usual dipole-like character, as in the ideal MHD case (Fig. 3.8).

Furthermore, we provide convergence results for the magnetic field for the strong field case, for three different grids (Table. 3.1), while the mass of the neutron star is 1.81 $M_{\odot}$. Even for the fields that are considered as realistically strong, the total magnetic energy is still several orders of magnitude smaller than the gravitational binding energy. As a result, very high resolution is required in order to reach convergence of the magnetic part.

Table 3.1: Convergence test for the strong magnetic field case, using three different grids , $N_{\rm p} = 1$, $N_{\rm n} = 0.9$, $\alpha = 200$, $\zeta = 1$, $x_{\rm p}(0) = 0.15$, $\kappa = 0.03$, $h_{\rm c} = 0.1$, $\omega = 0.02$.

Grid dimensions	$\mathcal{E}_{\rm mag}/\lvert W\rvert$	Mass	$\sigma_{u\ \rm min}$	Virial test
301× 301	1.52E-05	1.06	1.44E-07	1.31E-05
601× 601	1.19E-05	1.07	5.26E-07	1.41E-05
1201× 1201	1.32E-05	1.07	2.76E-07	2.84E-06

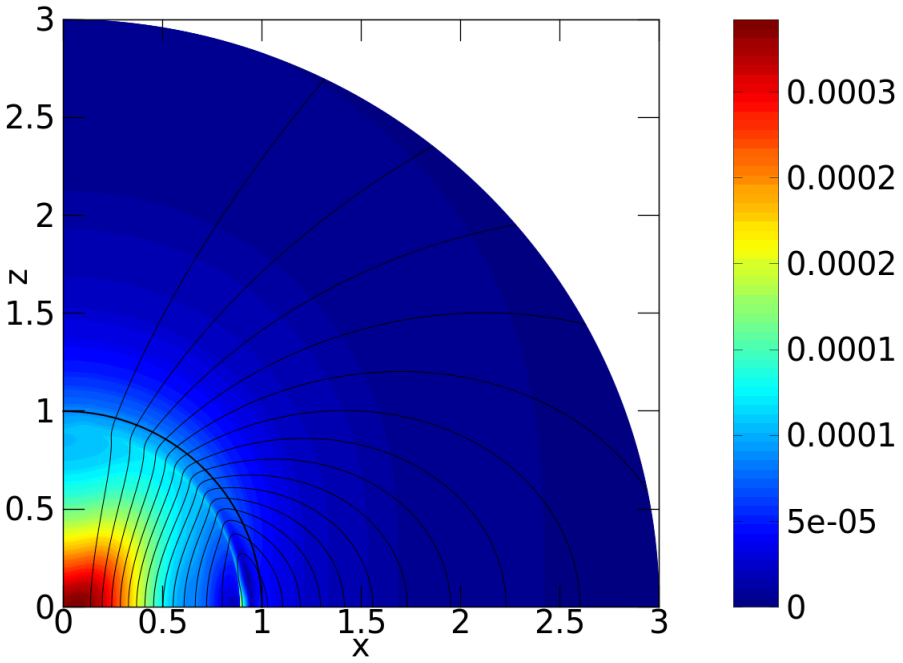


Figure 3.8: The weak field configuration calculated for $RMAX = 3\,r_{\rm eq}$.

Chapter 4

Discussion

In the current thesis we constructed equilibrium configurations of magnetized neutron stars. In the first part, we constructed rotating normal matter magnetized models reproducing the results in Tomimura & Eriguchi [25] with high accuracy. In the second, part we constructed models with a superconducting core and a normal matter crust, reproducing the qualitative behavior of the results shown in Lander [18]. Although the algorithm in the second part is constructed in such way that rotation is included, we focused here on non-rotating models and studied the influence of the number of iterations on the accuracy of the numerical solutions. We find that especially for weak magnetic fields it is not sufficient to terminate the iterative procedure when the matter fields have converged to a desired accuracy. The reason is that in this limit, matter is weakly coupled to the magnetic field, so that the latter needs an additional number of iterations i n order to converge to an accurate solution. In particular, we show that kinks in the magnetic field configuration are smoothed out if one allows for a larger number of iterations, until the cumulative numerical errors reach a plateau.

The next step to be done is to include realistic equations of state instead of polytropes as well as to assume a nonconstant entrainment parameter $\varepsilon_\star$, which would imply, as mentioned in Glampedakis, Andersson & Lander [7], the presence of a force acting on neutrons. The numerical code can also easily be extended to include a magnetosphere as well as differential rotation.

Appendix A

Mathematical tools

A.1 Mathematical derivations

In this section we provide more details about the derivation of various equations in Ch. 2 and Ch. 3.

A.1.1 The normal MHD case

A.1.1.1 The $\mathbf{j}_{\rm pol}$ and $\mathbf{B}_{\rm pol}$ cross product

$$\begin{aligned}
\mathbf{j}_{\rm pol} \times \mathbf{B}_{\rm pol} &= 0, \\
\left(\frac{1}{4\pi\varpi}\boldsymbol{\nabla}\left(\varpi B_\phi\right) \times \mathbf{e}_\phi\right) \times \left(\frac{1}{\varpi}\boldsymbol{\nabla} u \times \mathbf{e}_\phi\right) &= 0, \\
\left(\frac{1}{4\pi\varpi}\boldsymbol{\nabla}\left(\varpi B_\phi\right) \cdot \left(\frac{1}{\varpi}\boldsymbol{\nabla} u \times \mathbf{e}_\phi\right)\right)\mathbf{e}_\phi &= 0, \\
\frac{1}{4\pi\varpi^2}\left(\mathbf{e}_\phi \cdot \left(\boldsymbol{\nabla}\left(\varpi B_\phi\right) \times \boldsymbol{\nabla} u\right)\right)\mathbf{e}_\phi &= 0, \\
\boldsymbol{\nabla}\left(\varpi\, B_\phi\right) \times \boldsymbol{\nabla} u &= 0.
\end{aligned} \tag{A.1}$$

A.1.1.2 The decomposition of $\mathcal{L}$

The first term ($\mathbf{j}_{\rm pol} \times \mathbf{B}_{\rm tor}$) is

$$\begin{aligned}
\mathbf{j}_{\rm pol} \times \mathbf{B}_{\rm tor} &= \left(\frac{1}{4\pi\,\varpi} \boldsymbol{\nabla} f(u) \times \mathbf{e}_\phi \right) \times \frac{f(u)}{\varpi}\,\mathbf{e}_\phi \\
&= \left(\frac{1}{4\pi\,\varpi} \boldsymbol{\nabla} f(u) \cdot \frac{f(u)}{\varpi} \mathbf{e}_\phi \right) \mathbf{e}_\phi - \left(\mathbf{e}_\phi \cdot \frac{f(u)}{\varpi} \mathbf{e}_\phi \right) \frac{1}{4\pi\,\varpi} \boldsymbol{\nabla} f(u) \\
&= -\frac{f(u)}{4\pi\varpi^2} \boldsymbol{\nabla} f(u) \\
&= -\frac{f(u)}{4\pi\varpi^2} \frac{df}{du} \boldsymbol{\nabla} u.
\end{aligned} \tag{A.2}$$

The second term ($\mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm pol}$) is

$$\begin{aligned}
\mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm pol} &= -\frac{1}{4\pi\varpi} \Delta_\star u\, \mathbf{e}_\phi \times \left(\frac{1}{\varpi} \boldsymbol{\nabla} u \times \mathbf{e}_\phi \right) \\
&= \left(-\frac{1}{4\pi\varpi} \Delta_\star u\, \mathbf{e}_\phi \cdot \mathbf{e}_\phi \right) \frac{1}{\varpi} \boldsymbol{\nabla} u - \left(-\frac{1}{4\pi\varpi} \Delta_\star u\, \mathbf{e}_\phi \cdot \frac{1}{\varpi} \boldsymbol{\nabla} u \right) \mathbf{e}_\phi \\
&= -\frac{1}{4\pi\varpi^2} \Delta_\star u \boldsymbol{\nabla} u.
\end{aligned} \tag{A.3}$$

The last term ($\mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm tor}$) vanishes, since it is the cross product of parallel vectors.

$$\mathbf{j}_{\rm tor} \times \mathbf{B}_{\rm tor} = -\frac{1}{\varpi} \Delta_\star u\, \mathbf{e}_\phi \times \frac{f(u)}{\varpi}\,\mathbf{e}_\phi = 0. \tag{A.4}$$

A.1.1.3 The j and B relation

The current density (2.15) is related to the magnetic field through (using (2.10), (2.25))

$$\begin{aligned}
\mathbf{j} &= \frac{1}{4\pi\varpi} \frac{df}{du} \boldsymbol{\nabla} u \times \mathbf{e}_\phi + \left(\rho\varpi \frac{dM}{du} + \frac{f(u)}{4\pi\varpi} \frac{df}{du} \right) \mathbf{e}_\phi = \\
&= \frac{1}{4\pi} \frac{df}{du} \mathbf{B} + \rho\varpi \frac{dM}{du}\,\mathbf{e}_\phi.
\end{aligned} \tag{A.5}$$

A.1.2 The superconducting case

A.1.2.1 The unit current definition

We define a unit current $\hat{\mathbf{j}} \equiv \boldsymbol{\nabla} \times \hat{\mathbf{B}}$. The right hand side of the definition is

$$\begin{aligned}
\boldsymbol{\nabla} \times \hat{\mathbf{B}} &= -\frac{\partial \hat{B}_\phi}{\partial z} \mathbf{e}_\varpi + \left(\frac{\partial \hat{B}_\phi}{\partial \varpi} + \frac{\hat{B}_\phi}{\varpi} \right) \mathbf{e}_z + \left(\frac{\partial \hat{B}_z}{\partial z} - \frac{\partial \hat{B}_z}{\partial \varpi} \right) \mathbf{e}_\phi \\
&= \underbrace{\frac{1}{\varpi} \boldsymbol{\nabla} (\varpi \hat{B}_\phi) \times \mathbf{e}_\phi}_{\hat{\mathbf{j}}_{\rm pol}} + \underbrace{\hat{j}_\phi \mathbf{e}_\phi}_{\hat{\mathbf{j}}_{\rm tor}}.
\end{aligned} \tag{A.6}$$

A.1.2.2 The superconducting magnetic force manipulation

$$
\begin{aligned}
-\frac{4\pi}{h_{\mathrm{c}}}\mathbf{F}_{\mathrm{mag}} &= \rho_{\mathrm{p}}\boldsymbol{\nabla}B + \mathbf{B}\times\left[\rho_{\mathrm{p}}\left(\frac{1}{\varpi}\boldsymbol{\nabla}\left(\varpi\hat{B}_{\phi}\right)\times\mathbf{e}_{\phi}+\hat{j}_{\phi}\mathbf{e}_{\phi}\right)\right.\\
&\qquad\left.+\boldsymbol{\nabla}\rho_{\mathrm{p}}\times\left(\frac{1}{\varpi B}\boldsymbol{\nabla}u\times\mathbf{e}_{\phi}+\hat{B}_{\phi}\mathbf{e}_{\phi}\right)\right]\\
&= \rho_{\mathrm{p}}\boldsymbol{\nabla}B + \mathbf{B}\times\left[\frac{1}{\varpi}\rho_{\mathrm{p}}\boldsymbol{\nabla}\left(\varpi\hat{B}_{\phi}\right)\times\mathbf{e}_{\phi}+\hat{B}_{\phi}\boldsymbol{\nabla}\rho_{\mathrm{p}}\times\mathbf{e}_{\phi}+\hat{j}_{\phi}\rho_{\mathrm{p}}\mathbf{e}_{\phi}\right.\\
&\qquad\left.+\frac{1}{\varpi B}\boldsymbol{\nabla}\rho_{\mathrm{p}}\times\left(\boldsymbol{\nabla}u\times\mathbf{e}_{\phi}\right)\right]\\
&= \rho_{\mathrm{p}}\boldsymbol{\nabla}B + \mathbf{B}\times\left[\frac{1}{\varpi}\left(\rho_{\mathrm{p}}\boldsymbol{\nabla}\left(\varpi\hat{B}_{\phi}\right)\times\mathbf{e}_{\phi}+\varpi\hat{B}_{\phi}\boldsymbol{\nabla}\rho_{\mathrm{p}}\times\mathbf{e}_{\phi}\right)\right.\\
&\qquad\left.+\rho_{\mathrm{p}}\hat{j}_{\phi}\mathbf{e}_{\phi}-\frac{1}{\varpi B}\left(\boldsymbol{\nabla}u\cdot\boldsymbol{\nabla}\rho_{\mathrm{p}}\right)\mathbf{e}_{\phi}\right]\\
&= \rho_{\mathrm{p}}\boldsymbol{\nabla}B + \mathbf{B}\times\left[\frac{1}{\varpi}\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_{\phi}\right)\times\mathbf{e}_{\phi}+\rho_{\mathrm{p}}\hat{j}_{\phi}\mathbf{e}_{\phi}-\frac{\boldsymbol{\nabla}u\cdot\boldsymbol{\nabla}\rho_{\mathrm{p}}}{\varpi B}\mathbf{e}_{\phi}\right]\\
&= \rho_{\mathrm{p}}\boldsymbol{\nabla}B + \frac{1}{\varpi}\mathbf{B}\times\left(\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_{\phi}\right)\times\mathbf{e}_{\phi}\right)+\left(\rho_{\mathrm{p}}\hat{j}_{\phi}-\frac{\boldsymbol{\nabla}u\cdot\boldsymbol{\nabla}\rho_{\mathrm{p}}}{\varpi B}\right)\mathbf{B}\times\mathbf{e}_{\phi}.
\end{aligned}
\tag{A.7}
$$

Using the definition of $\mathbf{B}$ yields that the last term is purely poloidal

$$
\mathbf{B}\times\mathbf{e}_{\phi}=\left(\frac{1}{\varpi}\boldsymbol{\nabla}u\times\mathbf{e}_{\phi}\right)\times\mathbf{e}_{\phi}+\hat{B}_{\phi}\mathbf{e}_{\phi}\times\mathbf{e}_{\phi}=-\frac{1}{\varpi}\boldsymbol{\nabla}u. \tag{A.8}
$$

Expanding $\mathbf{B}$ in (A.7) and substituting the aforementioned result we obtain

$$
\begin{aligned}
-\frac{4\pi}{h_{\mathrm{c}}}\mathbf{F}_{\mathrm{mag}} &= \rho_{\mathrm{p}}\boldsymbol{\nabla}B + \frac{1}{\varpi}\left(\frac{1}{\varpi}\boldsymbol{\nabla}u\times\mathbf{e}_\phi + \hat{B}_\phi\mathbf{e}_\phi\right)\times\left(\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)\times\mathbf{e}_\phi\right)\\
&\quad + \left(-\rho_{\mathrm{p}}\hat{j}_\phi + \frac{\boldsymbol{\nabla}u\cdot\boldsymbol{\nabla}\rho_{\mathrm{p}}}{\varpi B}\right)\frac{\boldsymbol{\nabla}}{\varpi}\\
&= \rho_{\mathrm{p}}\boldsymbol{\nabla}B + \frac{1}{\varpi^2}\left(\boldsymbol{\nabla}u\times\mathbf{e}_\phi\right)\times\left(\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)\times\mathbf{e}_\phi\right)\\
&\quad + \frac{1}{\varpi}\hat{B}_\phi\mathbf{e}_\phi\times\left(\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)\times\mathbf{e}_\phi\right) + \left(\frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla}u}{\varpi B} - \rho_{\mathrm{p}}\hat{j}_\phi\right)\frac{\boldsymbol{\nabla}u}{\varpi}\\
&= \rho_{\mathrm{p}}\boldsymbol{\nabla}B\frac{1}{\varpi^2}\left[\boldsymbol{\nabla}u\cdot\left(\underbrace{\mathbf{e}_\phi\times\mathbf{e}_\phi}_{0}\right)\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right) - \boldsymbol{\nabla}u\cdot\left(\mathbf{e}_\phi\times\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)\right)\mathbf{e}_\phi\right]\\
&\quad + \frac{1}{\varpi}B_\phi\mathbf{e}_\phi\times\left(\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)\times\mathbf{e}_\phi\right) + \left(\frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla}u}{\varpi B} - \rho_{\mathrm{p}}\hat{j}_\phi\right)\frac{\boldsymbol{\nabla}u}{\varpi}\\
&= \rho_{\mathrm{p}}\boldsymbol{\nabla}B - \frac{1}{\varpi^2}\left[\left(\underbrace{\boldsymbol{\nabla}u\cdot\mathbf{e}_\phi}_{0}\right)\left(\mathbf{e}_\phi\times\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)\right)\right.\\
&\quad \left. + \left(\mathbf{e}_\phi\cdot\mathbf{e}_\phi\right)\left(\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)\times\boldsymbol{\nabla}u\right) + \left(\underbrace{\mathbf{e}_\phi\cdot\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)}_{0}\right)\left(\boldsymbol{\nabla}u\times\mathbf{e}_\phi\right)\right]\\
&\quad + \frac{B_\phi}{\varpi}\left[\left(\mathbf{e}_\phi\cdot\mathbf{e}_\phi\right)\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right) - \left(\underbrace{\mathbf{e}_\phi\cdot\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)}_{0}\right)\mathbf{e}_\phi\right]\\
&\quad + \left(\frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla}u}{\varpi B} - \rho_{\mathrm{p}}\hat{j}_\phi\right)\frac{\boldsymbol{\nabla}u}{\varpi}\\
&= \rho_{\mathrm{p}}\boldsymbol{\nabla}B + \frac{1}{\varpi^2}\boldsymbol{\nabla}u\times\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right) + \frac{B_\phi}{\varpi}\boldsymbol{\nabla}\left(\rho_{\mathrm{p}}\varpi\hat{B}_\phi\right)\\
&\quad + \left(\frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla}u}{\varpi B} - \rho_{\mathrm{p}}\hat{j}_\phi\right)\frac{\boldsymbol{\nabla}u}{\varpi}.
\end{aligned}
\tag{A.9}
$$

Since $f(u) = \rho_{\mathrm{p}}\varpi\hat{B}_\phi$, the aforementioned equation becomes

$$
\begin{aligned}
-\frac{4\pi}{h_{\mathrm{c}}}\rho_{\mathrm{p}}\boldsymbol{\nabla}M &= \rho_{\mathrm{p}}\boldsymbol{\nabla}B + \frac{B_\phi}{\varpi}\boldsymbol{\nabla}f(u) + \left(\frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla}u}{\varpi B} - \rho_{\mathrm{p}}\hat{j}_\phi\right)\frac{\boldsymbol{\nabla}u}{\varpi},\\
-\frac{4\pi}{h_{\mathrm{c}}}\boldsymbol{\nabla}M - \boldsymbol{\nabla}B &= \frac{B_\phi}{\rho_{\mathrm{p}}\varpi}\frac{df}{du}\boldsymbol{\nabla}u + \left(\frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla}u}{\varpi B\rho_{\mathrm{p}}} - \hat{j}_\phi\right)\frac{\boldsymbol{\nabla}u}{\varpi},\\
-\frac{4\pi}{h_{\mathrm{c}}}\boldsymbol{\nabla}M - \boldsymbol{\nabla}B &= \left(\frac{Bf}{\rho_{\mathrm{p}}^2\varpi}\frac{df}{du} + \frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla}u}{\varpi B\rho_{\mathrm{p}}} - \hat{j}_\phi\right)\frac{\boldsymbol{\nabla}u}{\varpi},\\
\boldsymbol{\nabla}\left(-\frac{4\pi}{h_{\mathrm{c}}}M - B\right) &= \left(\frac{Bf}{\rho_{\mathrm{p}}^2\varpi}\frac{df}{du} + \frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla}u}{\varpi B\rho_{\mathrm{p}}} - \hat{j}_\phi\right)\frac{\boldsymbol{\nabla}u}{\varpi}.
\end{aligned}
\tag{A.10}
$$

Using

$$y = \frac{4\pi}{h_{\mathrm{c}}} M + B, \tag{A.11}$$

and since $y = y(u)$, we can further derive

$$\begin{aligned} -\frac{dy}{du}\boldsymbol{\nabla} u &= \frac{1}{\varpi}\left(\frac{Bf}{\rho_{\mathrm{p}}^2 \varpi}\frac{df}{du} + \frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla} u}{\varpi B \rho_{\mathrm{p}}} - \hat{j}_\phi\right)\boldsymbol{\nabla} u \\ \frac{dy}{du} &= -\frac{Bf}{\varpi^2\rho_{\mathrm{p}}^2}\frac{df}{du} - \frac{\boldsymbol{\nabla}\rho_{\mathrm{p}}\cdot\boldsymbol{\nabla} u}{\varpi^2 B\rho_{\mathrm{p}}} + \frac{\hat{j}_\phi}{\varpi}. \end{aligned} \tag{A.12}$$

A.1.2.3 The $\hat{j}_\phi$ decomposition

Using (3.7, 3.14) we obtain

$$\begin{aligned} \hat{j}_\phi &= \left(\boldsymbol{\nabla}\times\hat{\mathbf{B}}\right)\cdot\mathbf{e}_\phi \\ &= \left(\boldsymbol{\nabla}\times\left(\frac{\mathbf{B}}{B}\right)\right)\cdot\mathbf{e}_\phi \\ &= \frac{1}{B}\left(\boldsymbol{\nabla}\times\mathbf{B}\right)\cdot\mathbf{e}_\phi + \left(\boldsymbol{\nabla}\frac{1}{B}\times\mathbf{B}\right)\cdot\mathbf{e}_\phi \\ &= \frac{1}{B}\left(\boldsymbol{\nabla}\times(B_\phi\mathbf{e}_\phi)\right)\cdot\mathbf{e}_\phi + \frac{1}{B}\left(\boldsymbol{\nabla} u\times\mathbf{e}_\phi\right)\cdot\mathbf{e}_\phi - \frac{1}{B^2}\left(\boldsymbol{\nabla} B\times\mathbf{B}\right)\cdot\mathbf{e}_\phi \\ &= \frac{1}{B}\left[\underbrace{B_\phi\left(\boldsymbol{\nabla}\times\mathbf{e}_\phi\right)\cdot\mathbf{e}_\phi}_{=0} + \underbrace{\left(\boldsymbol{\nabla} B_\phi\times\mathbf{e}_\phi\right)\cdot\mathbf{e}_\phi}_{=0} + \underbrace{\frac{1}{\varpi}\boldsymbol{\nabla} u\left(\boldsymbol{\nabla}\cdot\mathbf{e}_\phi\right)\mathbf{e}_\phi}_{=0} - \left(\mathbf{e}_\phi\cdot\mathbf{e}_\phi\right)\left(\boldsymbol{\nabla}\cdot\left(\frac{1}{\varpi}\boldsymbol{\nabla} u\right)\right)\right. \\ &\quad \left. + \left(\mathbf{e}_\phi\cdot\boldsymbol{\nabla}\right)\left(\frac{1}{\varpi}\boldsymbol{\nabla} u\right)\mathbf{e}_\phi - \underbrace{\left(\left(\frac{1}{\varpi}\boldsymbol{\nabla} u\cdot\boldsymbol{\nabla}\right)\mathbf{e}_\phi\right)\mathbf{e}_\phi}_{=0}\right] - \frac{1}{B^2}\left(\boldsymbol{\nabla} B\times\mathbf{B}\right)\cdot\mathbf{e}_\phi \\ &= \frac{1}{B}\left[-\frac{1}{\varpi}\left(\frac{\partial^2 u}{\partial\varpi^2} + \frac{\partial^2 u}{\partial z^2}\right) + \frac{1}{\varpi^2}\frac{\partial u}{\partial\varpi}\right] + \frac{1}{\varpi B^2}\boldsymbol{\nabla} u\cdot\boldsymbol{\nabla} B \\ &= -\frac{1}{\varpi B}\left[\underbrace{\left(\frac{\partial^2}{\partial\varpi^2} - \frac{1}{\varpi}\frac{\partial}{\partial\varpi} + \frac{\partial^2}{\partial z^2}\right)}_{\Delta_\star} u - \frac{1}{B}\boldsymbol{\nabla} B\cdot\boldsymbol{\nabla} u\right], \end{aligned} \tag{A.13}$$

where $\Delta_\star$ is given by (A.35).

A.1.2.4 The norm of B.

Using (3.7, 3.19) the norm of the magnetic field B in terms of u is

$$B \equiv \sqrt{\mathbf{B}\cdot\mathbf{B}} = \left\{ B_\phi^2 + \frac{1}{\varpi^2}\left[\left(\frac{\partial u}{\partial \varpi}\mathbf{e}_\varpi + \frac{\partial u}{\partial z}\right)\times \mathbf{e}_\phi\right]\cdot\left[\left(\frac{\partial u}{\partial \varpi}\mathbf{e}_\varpi + \frac{\partial u}{\partial z}\right)\times \mathbf{e}_\phi\right]\right\}^{1/2}, \tag{A.14}$$

Squaring both sides yields

$$\begin{aligned} B^2 &= \frac{f^2}{\varpi^2\rho_{\rm p}^2}B^2 + \frac{1}{\varpi^2}\left|\boldsymbol{\nabla} u\right|^2, \\ B &= \rho_{\rm p}\frac{\left|\boldsymbol{\nabla} u\right|}{\sqrt{\rho_{\rm p}^2\varpi^2 - f^2}}. \end{aligned} \tag{A.15}$$

A.1.3 The boundary conditions

At the crust core boundary, the first condition yields

$$\begin{aligned} \left[\rho_{\rm p}^{\rm core}\right]_{\rm cc}\left[\boldsymbol{\nabla}\frac{h_{\rm c}}{4\pi}\left(y-B\right)\right]_{\rm cc} &= \left[\rho_{\rm p}^{\rm crust}\right]_{\rm cc}\left[\frac{dM_{\rm N}}{du}\right]_{\rm cc}\left[\boldsymbol{\nabla} u\right]_{\rm cc}, \\ \left[\frac{dy}{du}\boldsymbol{\nabla} u - \boldsymbol{\nabla} B\right]_{\rm cc} &= \frac{4\pi}{h_{\rm c}}\left[\frac{\rho_{\rm p}^{\rm crust}}{\rho_{\rm p}^{\rm core}}\right]_{\rm cc}\left[\frac{dM_{\rm N}}{du}\right]_{\rm cc}\left[\boldsymbol{\nabla} u\right]_{\rm cc}, \\ \left[\boldsymbol{\nabla} B\right]_{\rm cc} &= \left[\left(\frac{dy}{du} - \frac{4\pi}{h_{\rm c}}\frac{\rho_{\rm p}^{\rm crust}}{\rho_{\rm p}^{\rm core}}\right)\boldsymbol{\nabla} u\right]_{\rm cc}. \end{aligned} \tag{A.16}$$

while the second condition suggests that

$$\begin{aligned} \left[B_\phi^{\rm core} - B_\phi^{\rm crust}\right]_{\rm cc} &= 0, \\ \left[\frac{1}{\varpi}\left(\frac{B}{\rho_{\rm p}^{\rm core}}f_{\rm sc}(u) - f_{\rm N}(u)\right)\right]_{\rm cc} &= 0, \\ f(u) \equiv f_{\rm sc}(u) &= \rho_{\rm p}^{\rm core}\frac{f_{\rm N}(u)}{\tilde{B}_{\rm cc}(u)}. \end{aligned} \tag{A.17}$$

A.1.4 Dimensionless density and enthalpy relation

Starting with (2.6) we have

$$\begin{aligned} P &= K\rho^{1+\frac{1}{N}}, \\ \frac{P}{\rho} &= K\rho^{\frac{1}{N}}. \end{aligned} \tag{A.18}$$

Differentiating both sides of (2.6), integrating after dividing by ρ and using the definition $H = \int \frac{dP}{\rho}$, we obtain

$$\begin{aligned} dP &= K\frac{N+1}{N}\rho^{\frac{1}{N}}, \\ \int \frac{dP}{\rho} &= K\frac{N+1}{N}\int \rho^{\frac{1}{N}-1}d\rho, \\ H &= K(N+1)\rho^{\frac{1}{N}} = (N+1)\frac{P}{\rho}. \end{aligned} \tag{A.19}$$

Inverting the aforementioned equation (using the left and middle parts) yields

$$\begin{aligned} H &= K(N+1)\rho^{\frac{1}{N}}, \\ \rho &= \left(\frac{H}{K(N+1)}\right)^{N}. \end{aligned} \tag{A.20}$$

Dividing this result by $\rho_{\max}$ we obtain

$$\begin{aligned} \hat{\rho} \equiv \frac{\rho}{\rho_{\max}} &= \left(\frac{H}{K(N+1)}\right)^{N}\rho_{\max}^{-1} \\ \hat{\rho} &= \left(\frac{H}{K(N+1)}\right)^{N}\left(\frac{H_{\max}}{K(N+1)}\right)^{-N} \\ \hat{\rho} &= \left(\frac{H}{H_{\max}}\right)^{N}. \end{aligned} \tag{A.21}$$

A.1.5 Dimensionless density and pressure relation

Dividing (2.6) by $P_{\max}$ we obtain

$$\begin{aligned} \frac{P}{P_{\max}} &= \frac{K\rho^{1+\frac{1}{N}}}{P_{\max}}, \\ \frac{P}{P_{\max}} &= \frac{K\rho^{1+\frac{1}{N}}}{K\rho_{\max}^{1+\frac{1}{N}}}, \\ \left(\frac{\rho}{\rho_{\max}}\right)^{1+\frac{1}{N}} &= \frac{P}{P_{\max}}, \\ P &= P_{\max}\hat{\rho}^{1+\frac{1}{N}}. \end{aligned} \tag{A.22}$$

A.1.6 Dimensionless density and chemical potential relation

In order to relate the density to the chemical potential we divide $\tilde{\mu}_{\mathrm{p}}$ with maximum value $\tilde{\mu}_{\mathrm{p}\,\max}$ and using (3.10a), (3.10b), (3.78) and (3.79) we obtain

$$
\begin{aligned}
\frac{\tilde{\mu}_{\mathrm{p}}}{\tilde{\mu}_{\mathrm{p\,max}}} &= \left(\frac{\rho_{\mathrm{p}}}{\rho_{\mathrm{p\,max}}}\right)^{\frac{1}{N_{\mathrm{p}}}}, \\
\frac{\rho_{\mathrm{p}}}{\rho_{\mathrm{p\,max}}} &= \left(\frac{\tilde{\mu}_{\mathrm{p}}}{\tilde{\mu}_{\mathrm{p\,max}}}\right)^{N_{\mathrm{p}}}, \\
\frac{\rho_{\mathrm{p}}}{\rho_{\mathrm{p\,max}}}\frac{\rho_{\mathrm{p\,max}}}{\rho_{\mathrm{max}}} &= \frac{\rho_{\mathrm{p\,max}}}{\rho_{\mathrm{max}}}\left(\frac{\tilde{\mu}_{\mathrm{p}}}{\tilde{\mu}_{\mathrm{p\,max}}}\right)^{N_{\mathrm{p}}}, \\
\frac{\rho_{\mathrm{p}}}{\rho_{\mathrm{max}}} &= \frac{\rho_{\mathrm{p\,max}}}{\rho_{\mathrm{max}}}\left(\frac{\tilde{\mu}_{\mathrm{p}}}{\tilde{\mu}_{\mathrm{p\,max}}}\right)^{N_{\mathrm{p}}}, \\
\hat{\rho}_{\mathrm{p}} &= x_{\mathrm{p}}(0)\left(\frac{\tilde{\mu_{\mathrm{p}}}}{\tilde{\mu}_{\mathrm{p\,max}}}\right)^{N_{\mathrm{p}}}.
\end{aligned}
\tag{A.23}
$$

A.2 Mathematical formulas

A.2.1 The Heavyside Step function

The heavyside step function is defined by

$$
\theta(x - x_0) = \begin{cases} 1\,, & x \geq x_0 \\ 0\,, & x < x_0 \end{cases}\,. \tag{A.24}
$$

A.2.2 Parity of Legendre polynomials

The parity of Legendre polynomials is even for n even and odd for odd n

$$
P_n(-\mu) = (-1)^n P_n(\mu). \tag{A.25}
$$

The parity of associate Legendre polynomials is

$$
P_n^m(-\mu) = (-1)^{m+n} P_n^m(\mu). \tag{A.26}
$$

A.2.3 Vector calculus identities

Here we show some of the vector calculus identities and definitions in cylindrical (ϖ, ϕ, z) and spherical polar (r, θ, ϕ) coordinates.

A.2.3.1 Gradient of a scalar

The gradient of a scalar f in cylindrical coordinates is given by

$$
\boldsymbol{\nabla} f = \frac{\partial f}{\partial \varpi}\mathbf{e}_{\varpi} + \frac{1}{\varpi}\frac{\partial f}{\partial \phi}\mathbf{e}_{\phi} + \frac{\partial f}{\partial z}\mathbf{e}_z, \tag{A.27}
$$

while in spherical polar coordinates by

$$\boldsymbol{\nabla} f = \frac{\partial f}{\partial r}\mathbf{e}_r + \frac{1}{r}\frac{\partial f}{\partial \theta}\mathbf{e}_\theta + \frac{1}{r\sin\theta}\frac{\partial f}{\partial \phi}\mathbf{e}_\phi. \tag{A.28}$$

A.2.3.2 Divergence of a vector

The divergence of a vector $\mathbf{A}$ is cylindrical coordinates is

$$\boldsymbol{\nabla}\cdot\mathbf{A} = \frac{1}{\varpi}\frac{\partial(\varpi A_\varpi)}{\partial \varpi} + \frac{1}{\varpi}\frac{\partial A_\phi}{\partial \phi} + \frac{\partial A_z}{\partial z}, \tag{A.29}$$

and

$$\boldsymbol{\nabla}\cdot\mathbf{A} = \frac{1}{r^2}\frac{\partial(r^2 A_r)}{\partial r} + \frac{1}{r\sin\theta}\frac{\partial(A_\theta \sin\theta)}{\partial \theta} + \frac{1}{r\sin\theta}\frac{\partial A_\phi}{\partial \phi}, \tag{A.30}$$

in spherical polar coordinates.

A.2.3.3 Curl of a vector

The curl of a vector $\mathbf{A}$ is

$$\boldsymbol{\nabla}\times\mathbf{A} = \left(\frac{1}{\varpi}\frac{\partial A_z}{\partial \phi} - \frac{\partial A_\phi}{\partial z}\right)\mathbf{e}_\varpi + \left(\frac{\partial A_\varpi}{\partial z} - \frac{\partial A_z}{\partial \varpi}\right)\mathbf{e}_\phi + \frac{1}{\varpi}\left(\frac{\partial(\varpi A_\phi)}{\partial \varpi} - \frac{\partial A_\varpi}{\partial \phi}\right)\mathbf{e}_z, \tag{A.31}$$

in cylindrical coordinates and

$$\boldsymbol{\nabla}\times\mathbf{A} = \frac{1}{r\sin\theta}\left(\frac{\partial(A_\phi\sin\theta)}{\partial\theta} - \frac{\partial A_\theta}{\partial\phi}\right)\mathbf{e}_r + \frac{1}{r}\left(\frac{1}{\sin\theta}\frac{\partial A_r}{\partial\phi} - \frac{\partial(rA_\phi)}{\partial r}\right)\mathbf{e}_\theta + \frac{1}{r}\left(\frac{\partial(rA_\theta)}{\partial r} - \frac{\partial A_r}{\partial\theta}\right)\mathbf{e}_\phi, \tag{A.32}$$

in spherical polar coordinates.

A.2.3.4 Laplacian of a scalar

The Laplacian operator of a scalar f is

$$\boldsymbol{\nabla}^2 f = \frac{1}{\varpi}\frac{\partial}{\partial\varpi}\left(\varpi\frac{\partial f}{\partial\varpi}\right) + \frac{1}{\varpi^2}\frac{\partial^2 f}{\partial\phi^2} + \frac{\partial^2 f}{\partial z^2}, \tag{A.33}$$

in cylindrical and coordinates and

$$\boldsymbol{\nabla}^2 f = \frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial f}{\partial r}\right) + \frac{1}{r^2\sin\theta}\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial f}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2 f}{\partial\phi^2}, \tag{A.34}$$

in spherical polar coordinates.

A.2.3.5 The $\Delta_\star$ operator

When deriving the Grad-Shafranov equation, we encounter the $\Delta_\star$ operator given by

$$\Delta_\star = \frac{\partial^2}{\partial \varpi^2} - \frac{1}{\varpi}\frac{\partial}{\partial \varpi} + \frac{\partial^2}{\partial z^2}. \tag{A.35}$$

We show that this operator can be written as a Laplacian operator. Let u be an arbitrary function. Then

$$\begin{aligned}
\frac{\varpi}{\sin\phi}\boldsymbol{\nabla}^2\left(\frac{u\sin\phi}{\varpi}\right) = \\
\frac{\varpi}{\sin\phi}\left[\frac{1}{\varpi}\frac{\partial}{\partial\varpi}\left(\varpi\frac{\partial}{\partial\varpi}\left(\frac{u\sin\phi}{\varpi}\right)\right) + \frac{1}{\varpi^2}\frac{\partial^2}{\partial\phi^2}\left(\frac{u\sin\phi}{\varpi}\right) + \frac{\partial^2}{\partial z^2}\left(\frac{u\sin\phi}{\varpi}\right)\right] = \\
\frac{\varpi}{\sin\phi}\left[\frac{1}{\varpi}\frac{\partial}{\partial\varpi}\left(\varpi\sin\phi\left(\frac{1}{\varpi}\frac{\partial u}{\partial\varpi} - u\frac{1}{\varpi^2}\right)\right) - \frac{1}{\varpi^3}\,u\,\sin\phi + \frac{\sin\phi}{\varpi}\frac{\partial^2 u}{\partial z^2}\right] = \\
\frac{\varpi}{\sin\phi}\left[\frac{\sin\phi}{\varpi}\left(\frac{\partial^2 u}{\partial\varpi^2} - \frac{\partial}{\partial\varpi}\left(\frac{u}{\varpi}\right)\right) - \frac{u}{\varpi^3}\sin\phi + \frac{\sin\phi}{\varpi}\frac{\partial^2 u}{\partial z^2}\right] = \\
\frac{\partial^2 u}{\partial\varpi^2} - \frac{1}{\varpi}\frac{\partial u}{\partial\varpi} + \frac{u}{\varpi^2} - \frac{u}{\varpi^2} + \frac{\partial^2 u}{\partial z^2} = \\
\frac{\partial^2}{\partial\varpi^2} - \frac{1}{\varpi}\frac{\partial}{\partial\varpi} + \frac{\partial^2}{\partial z^2} = \Delta_\star u.
\end{aligned} \tag{A.36}$$

A.2.3.6 Identities of gradients

Consider two arbitrary axisymmetric functions $A(\varpi, z)$ and $B(\varpi, z)$. Assuming that

$$\boldsymbol{\nabla} A = c\,\boldsymbol{\nabla} B, \tag{A.37}$$

holds, where c is a proportionallity constant, we show that $A = A(B)$ and hence

$$\boldsymbol{\nabla} A = \frac{dA}{dB}\boldsymbol{\nabla} B. \tag{A.38}$$

Starting with (A.37) we have

$$\begin{aligned}
\boldsymbol{\nabla} A = c\boldsymbol{\nabla} B, \\
\boldsymbol{\nabla} B \times \boldsymbol{\nabla} A = c\boldsymbol{\nabla} B \times \boldsymbol{\nabla} B, \\
\boldsymbol{\nabla} B \times \boldsymbol{\nabla} A = 0, \\
\left(-\frac{\partial A}{\partial\varpi}\frac{\partial B}{\partial z} + \frac{\partial A}{\partial z}\frac{\partial B}{\partial\varpi}\right)\mathbf{e}_\phi = 0, \\
\frac{\partial A}{\partial z}\frac{\partial B}{\partial\varpi} - \frac{\partial A}{\partial\varpi}\frac{\partial B}{\partial z} = 0, \\
\left|\frac{D(A,B)}{D(\varpi,z)}\right| = 0.
\end{aligned} \tag{A.39}$$

where $\frac{D(A,B)}{D(\varpi,z)}$ is the determinant of the Jacobian matrix. Thus, according to the implicit function theorem $A = A(B)$ and so we have

$$\boldsymbol{\nabla} A = \frac{dA}{dB} \boldsymbol{\nabla} B \tag{A.40}$$

A.2.3.7 θ and μ transformations

Instead of using θ we use $\mu = \cos\theta$. It follows that

$$d\mu = -\sin\theta \, d\theta \equiv -\sqrt{1-\mu^2} \, d\theta, \tag{A.41}$$

and hence the derivative operator $\frac{\partial}{\partial\theta}$ is

$$\frac{\partial}{\partial\theta} = -\sqrt{1-\mu^2} \frac{\partial}{\partial\mu}. \tag{A.42}$$

A.2.4 Solution of the Poisson equation

We solve the Poisson equation by inverting it using the appropriate Green's function. In general, for arbitrary functions Ψ and Φ we have

$$\begin{aligned} \boldsymbol{\nabla}^2 \Psi(\mathbf{r}) &= \Phi(\mathbf{r}), \\ \Psi(\mathbf{r}) &= -\frac{1}{4\pi} \int \frac{\Phi(\mathbf{r}')}{|\mathbf{r}' - \mathbf{r}|} dV \end{aligned} \tag{A.43}$$

One way of calculating the integral on the right side is by using the spherical harmonic expansion of $\frac{1}{|\mathbf{r}'-\mathbf{r}|}$. Here, $\mathbf{r}$ refers to an arbitrary observing point (r, θ, ϕ) while $\mathbf{r}'$ to an arbitrary source point (r', θ', ϕ').

In the general case, we have

$$\frac{1}{|\mathbf{r}' - \mathbf{r}|} = \begin{cases} \sum_{n=0}^{\infty} P_n(\cos\gamma) \frac{r'^n}{r^{n+1}}, & r \geq r' \\ \sum_{n=0}^{\infty} P_n(\cos\gamma) \frac{r^n}{r'^{n+1}}, & r < r' \end{cases}, \tag{A.44}$$

where P_n are the Legendre polynomials and γ is the angle between the source and the point of observation. The Legendre polynomials can be decomposed (through the addition theorem) into

$$\frac{1}{|\mathbf{r}' - \mathbf{r}|} = \begin{cases} \sum_{n=0}^{\infty} \sum_{m=-n}^{m=n} \frac{4\pi}{2n+1} \left[Y_n^m(\theta', \phi')\right]^\star Y_n^m(\theta, \phi) \frac{r'^n}{r^{n+1}}, & r \geq r' \\ \sum_{n=0}^{\infty} \sum_{m=-n}^{m=n} \frac{4\pi}{2n+1} \left[Y_n^m(\theta', \phi')\right]^\star Y_n^m(\theta, \phi) \frac{r^n}{r'^{n+1}}, & r < r' \end{cases}, \tag{A.45}$$

where $Y_n^m(\theta,\phi)$ are the spherical harmonics. The aforementioned equation in terms of the associated Legendre functions is written as

$$\frac{1}{|\mathbf{r}'-\mathbf{r}|}=\begin{cases}\sum_{n=0}^{\infty}\left(P_n(\cos\theta)P_n(\cos\theta')\right. \\ \qquad\left.+2\sum_{m=1}^{m=n}\frac{(n-m)!}{(n+m)!}P_n^m(\cos\theta)P_n^m(\cos\theta')\cos\left(m(\phi-\phi')\right)\frac{r'^n}{r^{n+1}}\right), & r\geq r' \\ \sum_{n=0}^{\infty}\left(P_n(\cos\theta)P_n(\cos\theta')\right. \\ \qquad\left.+2\sum_{m=1}^{m=n}\frac{(n-m)!}{(n+m)!}P_n^m(\cos\theta)P_n^m(\cos\theta')\cos\left(m(\phi-\phi')\right)\frac{r^n}{r'^{n+1}}\right), & r<r'\end{cases}. \tag{A.46}$$

Assuming specific symmetries for the source function $\Phi(\mathbf{r})$, (A.46) attains a simpler form. Here, we will examine the case of symmetry around the z axis (i.e. the source function being independent of ϕ) as well as symmetry around the z axis but with a multiplicative dependence on $\sin\phi$ (integrating $\Phi(r,\theta)\sin\phi$ as source). Substituting (A.46) into (A.43) and due to axisymmetry the integral with respect to ϕ can be separated from the r,θ integrals. The source function is also symmetric with respect to the equator. This is expressed through

$$\Phi(r,\cos\theta)=\Phi[r,\cos(\pi-\theta)], \tag{A.47}$$

while working with $\mu=\cos\theta$ instead of θ it is

$$\Phi(r,\mu)=\Phi(r,-\mu). \tag{A.48}$$

The ϕ part of the Green's function is $\cos\left[m(\phi-\phi')\right]$, $m=1..n$. Therefore

$$\int_0^{2\pi}\cos\left[m(\phi-\phi')\right]d\phi'=0,\ \ m=1..n\,, \tag{A.49}$$

for the purely axisymmetric case. Hence, the Green's function expansion is dependent only on r,θ ($m=0$) and can be simplified as follows

$$\frac{1}{|\mathbf{r}-\mathbf{r}'|}=\begin{cases}\sum_{n=0}^{\infty}P_n(\cos\theta)P_n(\cos\theta')\frac{r'^n}{r^{n+1}}, & r\geq r' \\ \sum_{n=0}^{\infty}P_n(\cos\theta)P_n(\cos\theta')\frac{r^n}{r'^{n+1}}, & r<r'\end{cases}. \tag{A.50}$$

The parity of the Legendre polynomials (A.2.2) along with the symmetry property (A.48) results in only even degree polynomials surviving the integration, while those of odd degree vanish. Therefore, the Green's function for an axisymmetric and equatorially symmetric source function is

$$\frac{1}{|\mathbf{r}-\mathbf{r}'|}=\begin{cases}\sum_{n=0}^{\infty}P_{2n}(\mu)P_{2n}(\mu')\frac{r'^{2n}}{r^{2n+1}}, & r\geq r' \\ \sum_{n=0}^{\infty}P_{2n}(\mu)P_{2n}(\mu')\frac{r^{2n}}{r'^{2n+1}}, & r<r'\end{cases}. \tag{A.51}$$

For the axisymmetric case with additional $\sin\phi'$ dependence we have

$$\int_0^{2\pi} \sin\phi' d\phi' = 0, \tag{A.52}$$

and

$$\int_0^{2\pi} \sin\phi' \cos\left[m(\phi-\phi')\right] d\phi' = \frac{2\sin\left[m(\phi-\pi)\right]\sin(m\pi)}{m^2-1} = \begin{cases} \pi\sin\phi, & m=1 \\ 0, & m=2..n \end{cases}, \tag{A.53}$$

where the first integral (A.52) refers to the $P_n(\cos\theta)P_n(\cos\theta')$ part of (A.46) while the second (A.53) to the rest of it. It is obvious, that integration eliminates Legendre polynomials (i.e. $P_n(\cos\theta)$) as well as all Legendre associate polynomials with order higher than one (i.e. $P_n^m(\cos\theta),\ m \geq 2$). According to the parity (A.2.2), first order associate Legendre polynomials are even if their degree is odd and odd when the degree is even. Since the source function $\Phi(\mathbf{r})$ is equatorially symmetric (i.e. even with respect to μ), only odd degree polynomials survive in the Green's function expansion and hence

$$\frac{1}{|\mathbf{r}'-\mathbf{r}|} = \begin{cases} 2\sum_{n=1}^{\infty} \frac{1}{2n(2n-1)} P_{2n-1}^1(\mu)P_{2n-1}^1(\mu')\cos\left(\phi-\phi'\right)\frac{r'^{2n-1}}{r^{2n}}, & r \geq r' \\ 2\sum_{n=1}^{\infty} \frac{1}{2n(2n-1)} P_{2n-1}^1(\mu)P_{2n-1}^1(\mu')\cos\left(\phi-\phi'\right)\frac{r^{2n-1}}{r'^{2n}}, & r < r' \end{cases}, \tag{A.54}$$

which is the Green's function polynomial expansion for an axisymmetric and equatorially symmetric source with additional $\sin\phi$ dependence.

A.3 Numerical schemes

A.3.1 Numerical integration

Simpson's formula for integrating a function $f(x)$ over the interval (x_0, x_n) using grid spacing h is given by

$$\int_{x_0}^{x_n} f(x)dx = \frac{h}{3}\sum_{k=0}^{n-2}\left[f(x_k) + 4f(x_{k+1}) + f(x_{k+2})\right] - \frac{x_n - x_0}{180}h^5 f^{(4)}(\xi), \tag{A.55}$$

where x_i is the ith grid point and $f^{(4)}(\xi)$ is the 4th derivative of $f(x)$ at some point $x_0 < \xi < x_n$. The last term $-\frac{x_n-x_0}{180}h^5 f^{(4)}(\xi)$ is the total error of the method which is of local 5th order and global 4th order.

A.3.2 Numerical differentiation

The four-point central differencing formula for differentiating a function $f(x)$ using grid spacing h is given by

$$f'(x_i) = \frac{1}{12h}\big[f(x_{i-2}) - 8f(x_{i-1}) + 8f(x_{i+1}) - f(x_{i+2})\big] + O(h^4), \tag{A.56}$$

where $O(h^4)$ denotes that the formula is of 4th order accuracy. Similarly, at the edges we use a forwards and backwards differencing scheme for the first two and last two grid points respectively

$$f'(x_i) = \frac{1}{h}\left[-\frac{25}{12}f(x_i) + 4f(x_{i+1}) - 3f(x_{i+2}) + \frac{4}{3}f(x_{i+3}) - \frac{1}{4}f(x_{i+4})\right] + O(h^4), \;\; i = 1, 2. \tag{A.57}$$

$$f'(x_i) = \frac{1}{h}\left[\frac{25}{12}f(x_i) - 4f(x_{i-1}) + 3f(x_{i-2}) - \frac{4}{3}f(x_{i-3}) + \frac{1}{4}f(x_{i-4})\right] + O(h^4), \;\; i = n, n-1. \tag{A.58}$$

A.3.3 Extrapolation

On the edges of the grid, division of small numbers may occur, producing highly inaccurate results. To prevent, this we resort to extrapolation from interior points, using a Lagrange polynomial of 2nd order

$$\begin{aligned} f(x_i) = {} & \frac{(x_i - x_{i+2})(x_i - x_{i+3})}{(x_{i+1} - x_{i+2})(x_{i+1} - x_{i+3})} f(x_{i+1}) + \frac{(x_i - x_{i+1})(x_i - x_{i+3})}{(x_{i+2} - x_{i+1})(x_{i+2} - x_{i+3})} f(x_{i+2}) \\ & + \frac{(x_i - x_{i+1})(x_i - x_{i+2})}{(x_{i+3} - x_{i+1})(x_{i+3} - x_{i+2})} f(x_{i+3}), \end{aligned} \tag{A.59}$$

where $i = 1$. To use the formula for points $i = n$ we change x_{i+k} into x_{i-k}.

A.3.4 Interpolation

In order to find the surface value of a quantity we use linear a interpolation formula and the fact that enthalpy H (or chemical potential $\tilde{\mu}_{\rm p}$) vanishes at the surface of the star. Assuming that the desired surface value is denoted with subscript s, while subscripts l and $l+1$ denote the last value inside the star and the first value outside the star, respectively, we have for a function F

$$F_s = F_l \left(\frac{F_{l+1} - F_l}{r_{l+1} - r_l}\right)(r_s - r_l)\,. \tag{A.60}$$

For the enthalpy, it holds (and in case of the model in Ch. 3 we use the proton chemical potential instead of the enthaply)

$$\begin{aligned} H_s = 0 &= H_l + \underbrace{\left(\frac{H_{l+1} - H_l}{r_{l+1} - r_l}\right)}_{\mathsf{a}} (r_s - r_l)\,, \\ 0 &= \mathsf{a}(r_s - r_l), \\ r_s &= -\frac{H_l}{\mathsf{a}} + r_l, \end{aligned} \tag{A.61}$$

where r_s is the distance to surface. Combining (A.60) and (A.61) we have

$$\begin{aligned} F_s &= F_l - \frac{H_l}{\mathsf{a}} \frac{F_{l+1} - F_l}{r_{l+1} - r_l}, \\ F_s &= F_l - \frac{H_l}{H_{l+1} - H_l}(F_{l+1} - F_l). \end{aligned} \tag{A.62}$$

Also, a 3rd order Lagrange polynomial is used in the case that a grid point attains infinite or not-a-number value (NaN)

$$\begin{aligned} f(x_i) = &\frac{(x_i - x_{i+2})(x_i - x_{i+3})(x_i - x_{i+4})}{(x_{i+1} - x_{i+2})(x_{i+1} - x_{i+3})(x_{i+1} - x_{i+4})} f(x_{i+1}) \\ &+ \frac{(x_i - x_{i+1})(x_i - x_{i+3})(x_i - x_{i+4})}{(x_{i+2} - x_{i+1})(x_{i+2} - x_{i+3})(x_{i+2} - x_{i+4})} f(x_{i+2}) \\ &+ \frac{(x_i - x_{i+1})(x_i - x_{i+2})(x_i - x_{i+4})}{(x_{i+3} - x_{i+1})(x_{i+3} - x_{i+2})(x_{i+3} - x_{i+4})} f(x_{i+3}) \\ &+ \frac{(x_i - x_{i+1})(x_i - x_{i+2})(x_i - x_{i+3})}{(x_{i+4} - x_{i+1})(x_{i+4} - x_{i+2})(x_{i+4} - x_{i+3})} f(x_{i+4}), \end{aligned} \tag{A.63}$$

where $i = 2..n-4$. This formula can be also written using the previous grid points, if we change x_{i+k} into x_{i-k} where $i = 5..n-1$.

A.3.5 Under-relaxation

When the iterative scheme used to evaluate u throughout the star does not converge, we resort to under-relaxation. Assuming that in the nth iteration we substitute u_n in the right side of (3.34), after integration we obtain $u_n^\star$. Then, the fully updated u (i.e. u_{n+1}) is given by

$$u_{n+1} = (1 - \omega)u_n + \omega u_n^\star, \tag{A.64}$$

where $\omega < 1$ is the under-relaxation parameter. If $\omega = 1$, we have the usual iterative scheme where $u_{n+1} = u_n^\star$.

A.3.6 Magnetic Flux

The magnetic flux $\Phi_{\rm B}$ is given by (3.97). To evaluate $\Phi_{\rm B}$ at the surface of the star we work as follows. The integrated surface is the surface of the star and since we examine non rotating equilibria it is constant and thus $r = r(\theta, \phi) = 1$. The surface element in spherical polar coordinates for the specific case is

$$d\mathbf{S} = r^2 \sin\theta d\theta d\phi \, \mathbf{e}_r. \tag{A.65}$$

Thus the flux one the surface is

$$\begin{aligned}
\Phi_{\rm B} &= \int\int_{\rm surface} \mathbf{B} \cdot d\mathbf{S} \\
&= \int_{\theta=0}^{\pi/2} \int_{\phi=0}^{2\pi} B_r \left(r\left(\theta,\phi\right),\theta\right) r\left(\theta,\phi\right)^2 \sin\theta \, d\theta d\phi \\
&= \int_{\theta=0}^{\pi/2} \int_{\phi=0}^{2\pi} B_r \left(1,\theta\right) \sin\theta \, d\theta d\phi \\
&= \int_{\mu=0}^{1} \int_{\phi=0}^{2\pi} B_r \left(1,\mu\right) \, d\mu d\phi \\
&= 2\pi \int_{\mu=0}^{1} B_r \left(1,\mu\right) \, d\mu.
\end{aligned} \tag{A.66}$$

Appendix B

Results

In this section we provide some additional results for our models.

B.1 First part models

Table B.1: Comparison between our results and Tomimura & Eriguchi (TE) [25], $\alpha = 200$, $\kappa_0 = 0.3$, $N = 1.5$ for a 401×401 grid.

Model	$r_{\rm p}/r_{\rm e}$	$\mathcal{E}_{\rm mag}/\lvert W\rvert$	$3\Pi/\lvert W\rvert$	$T/\lvert W\rvert$	$\lvert W\rvert$	Ω_0^2	C	M	Virial test
TE	0.806	0.0548	0.315	5.70E-04	3.52E-02	2.50E-04	-0.0657	0.709	4.61E-05
current	0.805	0.0548	0.315	6.27E-04	3.52E-02	2.75E-04	-0.0657	0.709	8.48E-06
TE	0.8	0.0548	0.314	2.09E-03	3.48E-02	9.09E-04	-0.0654	0.703	4.60E-05
current	0.8	0.0548	0.314	2.08E-03	3.47E-02	9.08E-04	-0.0654	0.703	8.51E-06
TE	0.75	0.0542	0.305	0.0161	3.00E-02	6.73E-03	-0.0633	0.647	4.73E-05
current	0.749	0.0542	0.304	0.0163	3.00E-02	6.81E-03	-0.0632	0.646	8.9E-06
TE	0.7	0.0532	0.295	0.0308	2.53E-02	1.23E-02	-0.0605	0.586	4.96E-05
current	0.699	0.0531	0.295	0.0312	2.52E-02	1.24E-02	-0.0604	0.584	9.49E-06
TE	0.65	0.0513	0.286	0.0460	2.05E-02	1.74E-02	-0.0567	0.519	5.28E-05
current	0.651	0.0513	0.286	0.0458	2.06E-02	1.73E-02	-0.067	0.520	1.01E-05
TE	0.6	0.0479	0.277	0.0609	1.54E-02	2.18E-02	-0.0513	0.440	5.68E-05
	0.6	0.0479	0.277	0.0609	1.54E-02	2.18E-02	-0.0513	0.440	1.09E-05
TE	0.55	0.0411	0.271	0.0725	9.92E-03	2.47E-02	-0.0431	0.340	6.37E-05
current	0.549	0.0410	0.271	0.0726	9.85E-03	2.47E-02	-0.0429	0.339	1.25E-05
TE	0.522	0.0346	0.273	0.0735	6.74E-03	2.49E-02	-0.0364	0.270	7.07E-05
current	0.523	0.0347	0.273	0.0735	6.79E-03	2.49E-02	-0.0365	0.272	1.09E-05

Table B.2: Comparison between our results and Tomimura & Eriguchi (TE) [25], $\alpha = 200$, $\kappa_0 = 0.35$, $N = 1.5$ for a 401×401 grid.

Model	$r_{\rm p}/r_{\rm e}$	$\mathcal{E}_{\rm mag}/\lvert W\rvert$	$3\Pi/\lvert W\rvert$	$T/\lvert W\rvert$	$\lvert W\rvert$	Ω_0^2	C	M	Virial test
TE	0.722	0.0846	0.304	1.24E-03	3.81E-02	5.33E-04	-0.0729	0.736	5.44E-05
current	0.720	0.0846	0.304	1.92E-03	3.97E-02	8.24E-04	-0.0728	0.734	9.40E-06
TE	0.7	0.0847	0.300	8.24E-03	3.60E-02	3.48E-03	-0.0721	0.713	5.49E-05
current	0.699	0.0848	0.299	8.65E-03	3.59E-02	3.65E-03	-0.0721	0.712	9.63E-06
TE	0.65	0.0847	0.289	0.0248	3.14E-02	9.99E-03	-0.0702	0.658	5.76E-05
current	0.651	0.0847	0.289	0.0245	3.14E-02	9.90E-03	-0.0702	0.659	1.01E-05
TE	0.6	0.0839	0.277	0.0426	2.65E-02	1.63E-02	-0.0673	0.598	6.06E-05
current	0.6	0.0839	0.277	0.0426	2.65E-02	1.63E-02	-0.0673	0.598	1.07E-05
TE	0.55	0.0811	0.265	0.0618	2.11E-02	2.22E-02	-0.0627	0.524	6.43E-05
current	0.549	0.0811	0.265	0.0620	2.10E-02	2.23E-02	-0.0627	0.522	1.15E-05
TE	0.5	0.0707	0.255	0.0816	1.33E-02	2.71E-02	-0.0523	0.403	6.99E-05
current	0.499	0.0701	0.255	0.0821	1.31E-02	2.72E-02	-0.0520	0.398	1.28E-05
TE	0.478	0.0519	0.261	0.0823	6.96E-03	2.69E-02	-0.0388	0.275	7.85E-05
current	0.477	0.0513	0.262	0.0820	6.82E-03	2.68E-02	-0.0384	0.272	7.59E-06

Table B.3: Comparison between our results and Tomimura & Eriguchi (TE) [25], $\alpha = 200$, $\kappa_0 = 0.41$, $N = 1.5$ for a 401×401 grid.

Model	$r_{\rm p}/r_{\rm e}$	$\mathcal{E}_{\rm mag}/\lvert W\rvert$	$3\Pi/\lvert W\rvert$	$T/\lvert W\rvert$	$\lvert W\rvert$	Ω_0^2	C	M	Virial test
TE	0.517	0.180	0.273	5.53E-04	4.71E-02	2.13E-04	-0.0950	0.823	8.44E-05
current	0.517	0.180	0.273	3.80E-04	4.72E-02	1.46E-04	-0.0950	0.828	1.26E-05
TE	0.5	0.186	0.270	2.04E-03	4.58E-02	7.60E-04	-0.0949	0.817	8.72E-05
current	0.499	0.187	0.270	2.02E-03	4.58E-02	7.50E-04	-0.0949	0.817	1.29E-05
TE	0.45	0.212	0.263	1.70E-04	4.15E-02	5.59E-05	-0.0931	0.782	9.77E-05
current	0.451	0.212	0.263	1.50E-04	4.16E-02	4.96E-05	-0.0932	0.783	1.41E-05

Appendix C

Source Code

C.1 Rotating magnetized normal matter neutron stars

```
1  /*****************************************************************/
2  /*                          NMAG11a.C                            */
3  /*                                                               */
4  /* Newtonian models of magnetized rotating polytropic stars.     */
5  /*                                                               */
6  /* Author: K.Palapanidis                                         */
7  /* (based on an earlier nonmagnetized version by N. Stergioulas, */
8  /*  1993)                                                        */
9  /* Date: Winter 2014                                             */
10 /*                                                               */
11 /* Usage:  nmag11a  -N n_index -r r_ratio                        */
12 /*                                                               */
13 /*****************************************************************/
14
15 #include <stdio.h>
16 #include <string.h>
17 #include <math.h>
18
19
20 #define KDIV 451                    /* grid points in mu-direction */
21 #define NDIV 451                    /* grid points in r-direction */
22 #define RMAX 1.06666666666666666    /* multiply r by 16.0/15.0 */
23 #define LMAX 16                     /* 1/2 of max. term in Legendre poly.*/
24 #define PI 3.141592653589793
25 #define Sqrt_4PI 3.5449077018110318
```

```
26
27    int  n_ra ,
28         counter=0,
29         point [KDIV+1],
30         max_count=0;                              /* grid position of r_a=1.0 */
31
32
33    double rho[KDIV+1][NDIV+1],             /* density */
34           mu[KDIV+1],                      /* grid points in mu-direction */
35           r[NDIV+1],                       /* grid points in r-direction */
36           theta[KDIV+1],
37           f2n[LMAX+1][NDIV+1][NDIV+1],     /* function f_2n */
38           p2n[LMAX+1][KDIV+1],             /* function p_2n */
39           phi[KDIV+1][NDIV+1],             /* gravitational potential */
40           n_index ,                        /* index N in polytropic EOS */
41           omega_02 ,                       /* omega_0^2 */
42           h_max ,                          /* H_max */
43           p_max ,                          /* maximum pressure */
44           h[KDIV+1][NDIV+1],               /* enthalpy H */
45           v ,                              /* volume */
46           m,                               /* mass */
47           mi ,                             /* moment of inertia */
48           am ,                             /* angular momentum */
49           ke ,                             /* kinetic energy */
50           w,                               /* gravitational energy */
51           pres[KDIV+1][NDIV+1],            /* pressure */
52           pint ,                           /* integral of pressure */
53           vt ,                             /* Virial test = | 2T+W+3Pi | / |W| */
54           omega_k2 ,                       /* omega_Kepler^2 */
55
56
57           Aphi[KDIV+1][NDIV+1],            /* Phi component of vector Potential A */
58           f2n_1[LMAX+1][NDIV+1][NDIV+1],   /* function f_2n-1 */
59           p1_2n_1[LMAX+1][KDIV+1],         /* Assoc Legendre P^1_2n-1 */
60           Emag,                             /* Magnetic Energy   */
61           Emagtor ,
62           k0=0.4*Sqrt_4PI ,               /* k(u)=k0 function*/
63           alpha_c=200.0/Sqrt_4PI ,    //4.5,//4.5,                    /* alpha constant
                 */
64           F[KDIV+1][NDIV+1],               /* Aphi density   */
65           zeta=1.0,
```

```
66            dAphi2[LMAX+1][NDIV+1],
67            dAphi1[NDIV+1][LMAX+1],
68            r_bound1[KDIV+1],
69            c,
70            B_pol_norm[KDIV+1][NDIV+1],
71            B_tor_norm[KDIV+1][NDIV+1],
72            x[KDIV+1][NDIV+1],
73            z[KDIV+1][NDIV+1],
74             df_du[KDIV+1][NDIV+1],
75            mag_f[KDIV+1][NDIV+1],
76            kfun[KDIV+1][NDIV+1];
77
78
79
80
81
82    // Heavyside step function
83    double unit_step(double x, double x0)
84
85    {
86            if(x>=x0) return 1.0;
87            else return 0.0;
88            }
89
90    /*************************************************************************/
91    /* A first guess for the density distribution is stored in the array     */
92    /* rho[i][j]. It corresponds to a uniform-density nonrotating sphere.    */
93    /*************************************************************************/
94    void guess_density(void)
95    {
96     int i,
97         j;
98
99      for(j=1;j<=NDIV;j++)
100      {
101       if(j<=n_ra) rho[1][j]=1.0;       /* First find rho for mu=0 */
102        else
103         rho[1][j]=0.0;
104
105       for(i=1;i<=KDIV;i++) rho[i][j]=rho[1][j];
106      }
```

```
107   }
108
109   /*************************************************************************/
110   /* A first guess for the density distribution is stored in the array      */
111   /* Aphi[i][j]. It corresponds to a uniform-density nonrotating sphere.    */
112   /*************************************************************************/
113   void guess_Aphi(void)
114   {
115    int i,
116        j;
117
118      for(j=1;j<=NDIV;j++)
119       {
120        for(i=1;i<=KDIV;i++) Aphi[i][j]=0.0;
121       }
122   }
123
124   /*************************************************************/
125   /* Create the grid points where everything is evaluated.     */
126   /* The points in the mu-direction are stored in the array mu[i].  */
127   /* The points in the r-direction are stored in the array r[j].    */
128   /*************************************************************/
129   void make_grid(void)
130   {
131     int i,
132         j;
133
134     for (i=1;i<=KDIV;i++) mu[i]=(i-1.0)/(KDIV-1.0);
135
136     for (j=1;j<=NDIV;j++) r[j]=RMAX*(j-1.0)/(NDIV-1.0);
137   }
138
139   /*************************************************************/
140   /* Returns the Legendre polynomial of degree n, evaluated at x.  */
141   /*************************************************************/
142   double legendre( int n, double x )
143   {
144     int i;
145
146     double p,          /* Legendre polynomial of order n */
147            p_1,        /*    "          "       "    "   n-1*/
```

```
148             p_2;        /*     "           "       "     "    n-2 */
149
150
151       p_2=1.0;
152       p_1=x;
153
154      if(n>=2)
155       { for(i=2;i<=n;i++)
156         {
157          p=(x*(2.0*i-1.0)*p_1 - (i-1.0)*p_2)/i;
158          p_2=p_1;
159          p_1=p;
160         }
161       return p;
162       } else
163         { if (n==1) return p_1;
164           else return p_2;
165         }
166     }
167
168     /*****************************************************************/
169     /* Returns the Associated Legendre polynomial P_n^m  m=1, evaluated at x.    */
170     /*****************************************************************/
171
172     double Assoc_legendre( int n, double x )
173     {
174       int i;
175
176       double p,         /*Assoc. Legendre polynomial P_l^1 */
177              p_1,       /*     "          "       "  P_(l-1)^1*/
178              p_2;       /*     "          "       "  P_(l-2)^1 */
179
180
181       p_2=-pow( (1.0-pow(x,2.0) ),0.5);
182       p_1=-3*x*pow( (1.0-pow(x,2.0) ),0.5);
183
184      if(n>=3)
185       { for(i=3;i<=n;i++)
186         {
187          p=( (2.0*i-1.0)/(i-1.0) ) * x * p_1 - ((i) /(i-1.0) ) *p_2;
188          p_2=p_1;
```

```
189         p_1=p;
190       }
191     return p;
192     } else
193       { if (n==1) return p_2;
194         else return p_1;
195       }
196   }
197
198   /*****************************************************************************/
199   /* Computing the radial component f_n(r,r')*r' of the polynomial expansion of 1/|r-r'|
        */
200   /*****************************************************************************/
201    /* n=degree of the Legendre polynomials */
202   double f_n(int n, double vec_r[NDIV+1],int k,int j)
203   {
204     double f;
205
206               if (k<j)  f=pow(vec_r[k],n+2.0)/pow(vec_r[j],n+1.0) ;
207                else
208                   { if( j==1) f=0;
209                      else f=pow(vec_r[j],n)/pow(vec_r[k],n-1.0);
210                   }
211               return f;
212
213      }
214
215   /*****************************************************************/
216   /* Since the grid points are fixed, we can compute the functions   */
217   /* f_n(r',r)*r'^2  and P_2n(mu) once at the beginning.             */
218   /*****************************************************************/
219   void comp_f_2n_p_2n(void)
220   {
221     int m,n,                  /* 2n=degree of the Legendre polynomials */
222         k,                  /* counter for r' */
223         j,                  /* counter for r */
224         i;                  /* counter for P_2n*/
225    double vec_r[NDIV+1];
226
227
228
```

```
229
230       for (n=0;n<=LMAX;n++)
231        {
232          for (k=1;k<=NDIV;k++)
233           {
234             for (j=1;j<=NDIV;j++)
235              {
236                f2n[n][k][j]=f_n(2*n,r,k,j);
237                f2n_1[n][k][j]=f_n(2*n-1,r,k,j);
238              }
239           }
240        }
241
242      for(i=1;i<=KDIV;i++)
243       {
244         for(n=0;n<=LMAX;n++) { p2n[n][i] = legendre(2*n,mu[i]);
245                                p1_2n_1[n][i] = Assoc_legendre( 2*n-1, mu[i]);
246                               }
247
248       }
249     }
250
251     /*************************************************************************/
252     /* Returns the maximum value in a KDIV x NDIV array.                     */
253     /*************************************************************************/
254     double max(double array[KDIV+1][NDIV+1])
255     {
256      int i,              /* counter */
257          j;              /* counter */
258      double max_val;     /* intermediate max. value */
259
260      max_val=array[1][1];
261
262      for(i=1;i<=KDIV;i++)
263       {
264         for(j=1;j<=NDIV;j++)
265          {
266            if(array[i][j]>max_val)   max_val=array[i][j];
267          }
268       }
269      return max_val;
```

```
270     }
271
272     /*******************************************************/
273     /* Find surface maximum for r*Aphi */
274     /*******************************************************/
275     double surf_max(void)
276     {
277     int i,
278         j,
279              j_b;
280
281
282     double maximum,
283              r_bound[KDIV+1],
284              Aphisur[KDIV+1],
285              max_check[KDIV+1],
286              alpha;
287
288
289
290           maximum=-1.0e+10;
291
292         /* Boundary */
293
294           for(i=1;i<=KDIV;i++)
295            {
296             j=1;
297             r_bound[i]=0.0;
298             while(h[i][j]>=0)
299              {
300               j_b=j;              /*  find find last point j_b inside star */
301               j++;
302              }
303
304             alpha= (NDIV-1)/RMAX*(h[i][j_b+1]-h[i][j_b]);    /* slope */
305             r_bound[i]=r[j_b]-h[i][j_b]/alpha;    /* linear interpolation */
306
307           Aphisur[i]=Aphi[i][j_b]-(h[i][j_b]/(h[i][j_b+1]-h[i][j_b]))*(Aphi[i][j_b+1]-Aphi[i][j_b]);
308             max_check[i]=r_bound[i]*pow(1.0-mu[i]*mu[i],0.5)*Aphisur[i];
309            }
310
```

```
311
312
313             for(i=1;i<KDIV;i++) {
314                if(maximum<max_check[i]){
315             maximum=max_check[i];
316
317             }
318             }
319
320             return maximum;
321             }
322
323
324   /*****************************************************************/
325   /* Find the density-like function for Aphi */
326   /*****************************************************************/
327   void Aphi_dens(void)
328   {
329
330       int i,
331       j;
332
333   double max;
334
335   max=surf_max();
336
337    for(i=1;i<=KDIV;i++)
338           {
339            for(j=1;j<=NDIV;j++)
340             {
341
342
343          df_du[i][j]=   alpha_c*pow( (r[j]*pow(1.0-mu[i]*mu[i],0.5)*Aphi[i][j]-max),zeta)*
344                             unit_step(r[j]*pow(1.0-mu[i]*mu[i],0.5)*Aphi[i][j],max);
345
346          mag_f[i][j]=   (alpha_c/(zeta+1.0) )*
347                          pow( (r[j]*pow(1.0-mu[i]*mu[i],0.5)*Aphi[i][j]-max),zeta+1.0)*
348                          unit_step(r[j]*pow(1.0-mu[i]*mu[i],0.5)*Aphi[i][j],max);
349
350          kfun[i][j]=    k0;
351
```

```
352         F[i][j]=(1.0/(4.0*PI) )* (1.0/(r[j]*pow(1.0-mu[i]*mu[i],0.5) ) ) *
353                         df_du[i][j]*mag_f[i][j] +
354                         kfun[i][j]*r[j]*pow(1.0-mu[i]*mu[i],0.5)*rho[i][j];
355
356
357             }
358          }
359                 for(i=1;i<=KDIV;i++){
360
361                 F[i][1]= ( (r[1]  -r[3])*(r[1]  -r[4])  ) * (F[i][2] )/
362                                                 ( (r[2]-r[3])*(r[2]-r[4])  ) +
363
364                                                 ( (r[1]  -r[2])*(r[1]  -r[4])  ) * (F[i][3]
                                                    )/
365                                                 ( (r[3]-r[2])*(r[3]-r[4])  ) +
366
367                                                 ( (r[1]  -r[2])*(r[1]  -r[3])  ) * (F[i][4]
                                                    )/
368                                                 ( (r[4]-r[2])*(r[4]-r[3])  );
369
370
371                                         }
372             for(j=1;j<=NDIV;j++){
373
374         F[KDIV][j]=    ( (mu[KDIV]  -mu[KDIV-2])*(mu[KDIV]  -mu[KDIV-3])  ) * (F[KDIV-1][j] )/
375                                                 (
                                                        (mu[KDIV-1]-mu[KDIV-2])*(mu[KDIV-1]-mu[KDIV-3])
                                                    ) +
376
377                                                 ( (mu[KDIV]  -mu[KDIV-1])*(mu[KDIV]
                                                    -mu[KDIV-3])  ) * (F[KDIV-2][j] )/
378                                                 (
                                                        (mu[KDIV-2]-mu[KDIV-1])*(mu[KDIV-2]-mu[KDIV-3])
                                                    ) +
379
380                                                 ( (mu[KDIV]  -mu[KDIV-1])*(mu[KDIV]
                                                    -mu[KDIV-2])  ) * (F[KDIV-3][j] )/
381                                                 (
                                                        (mu[KDIV-3]-mu[KDIV-1])*(mu[KDIV-3]-mu[KDIV-2])
                                                    );
382
```

```
383
384
385     }
386
387
388      }
389
390     /*************************************************************************/
391     /* Main iteration routine.                                               */
392     /*************************************************************************/
393     void iterate( int n_rb, double n_index )
394     {
395       int i,
396           j,
397           n,
398           k,
399           iter;                       /* counter */
400
401       double d1[NDIV+1][LMAX+1],   /* function D^(1)_{k,n} */
402              d2[LMAX+1][NDIV+1],   /* function D^(2)_{n,j} */
403              s=0.0,                /* term in sum */
404              sum=0.0,              /* intermediate sum */
405                                  /* constant C */
406              c_old=0.0,            /* C in previous cycle */
407              omega_02_old=0.0,     /* omega_0^2 in previous cycle */
408              h_max_old,            /* H_max in previous cycle */
409              dif1=1.0,             /* | h_max_old - h_max | */
410              dif2=1.0,             /* | omega_02_old - omega_02 | */
411              dif3=1.0;             /* | c_old - c | */
412
413
414       while( (dif1>1.0e-6) || (dif2>1.0e-6) || (dif3>1.0e-6) )
415
416     // for(iter=1;iter<70;iter++) // for specifies iterations
417        {
418             counter++;
419
420          /* Gravitational Potential */
421
422          for(k=1;k<=NDIV;k++)
423           {
```

```
424             for(n=0;n<=LMAX;n++)
425              {
426               for(i=1;i<=KDIV-2;i+=2)
427                {
428                 s= (1.0/(3.0*(KDIV-1.0)))*(p2n[n][i]*rho[i][k]
429                       + 4.0*p2n[n][i+1]*rho[i+1][k]
430                       + p2n[n][i+2]*rho[i+2][k]);
431                 sum+=s;
432                }
433                 d1[k][n]=sum;
434                 sum=0.0;
435              }
436            }
437
438
439          sum=0.0;
440          for(j=1;j<=NDIV;j++)
441           {
442            for(n=0;n<=LMAX;n++)
443             {
444              for(k=1;k<=NDIV-2;k+=2)
445               {
446                 s=RMAX/(3.0*(NDIV-1))*( f2n[n][k][j]*d1[k][n]
447                                + 4.0*f2n[n][k+1][j]*d1[k+1][n]
448                                + f2n[n][k+2][j]*d1[k+2][n]);
449                 sum+=s;
450               }
451                d2[n][j]=sum;
452                sum=0.0;
453             }
454          }
455
456
457           sum=0.0;
458           for(i=1;i<=KDIV;i++)
459            {
460             for(j=1;j<=NDIV;j++)
461              {
462               for(n=0;n<=LMAX;n++)
463                {
464                 s= -4.0*PI*d2[n][j]*p2n[n][i];
```

```
465
466                 sum+=s;
467                }
468                 phi[i][j]=sum;
469                 sum=0.0;
470              }
471            }
472
473
474        for(i=1;i<=KDIV;i++) phi[i][1]=phi[i][2]; /* Correct sing. at r=0. */
475                                                  /* Introduced error is   */
476                                                  /* negligible.           */
477
478         /* Vector Potential Aphi   */
479
480              Aphi_dens();
481
482              sum=0.0;
483        for(k=1;k<=NDIV;k++)
484         {
485          for(n=1;n<=LMAX;n++)
486           {
487            for(i=1;i<=KDIV-2;i+=2)
488             {
489       s= (1.0/(3.0*(KDIV-1.0)))*
490                    (p1_2n_1[n][i]*F[i][k]
491                   + 4.0*p1_2n_1[n][i+1]*F[i+1][k]
492                   + p1_2n_1[n][i+2]*F[i+2][k] );
493              sum+=s;
494             }
495              dAphi1[k][n]=sum;
496              sum=0.0;
497           }
498         }
499
500        sum=0.0;
501        for(j=1;j<=NDIV;j++)
502         {
503          for(n=1;n<=LMAX;n++)
504           {
505            for(k=1;k<=NDIV-2;k+=2)
```

```
506             {
507               s=RMAX/(3.0*(NDIV-1))*( f2n_1[n][k][j]*dAphi1[k][n]
508                                     + 4.0*f2n_1[n][k+1][j]*dAphi1[k+1][n]
509                                     + f2n_1[n][k+2][j]*dAphi1[k+2][n] );
510               sum+=s;
511             }
512              dAphi2[n][j]=sum;
513              sum=0.0;
514           }
515         }
516
517          sum=0.0;
518          for(i=1;i<=KDIV;i++)
519           {
520            for(j=1;j<=NDIV;j++)
521             {
522              for(n=1;n<=LMAX;n++)
523               {
524                s= 4.0*PI*dAphi2[n][j]*p1_2n_1[n][i]*(1.0/(2.0*n*(2.0*n-1.0)));
525
526                sum+=s;
527               }
528                Aphi[i][j]=sum;
529                sum=0.0;
530             }
531           }
532
533
534        for(i=1;i<=KDIV;i++) Aphi[i][1]=Aphi[i][2]; /* Correct sing. at r=0. */
535                                                    /* Introduced error is   */
536                                                    /* negligible.           */
537
538
539         /* omega_02 and c */
540
541         omega_02 = 2.0*(phi[1][n_ra]-phi[KDIV][n_rb]-k0*Aphi[1][n_ra]);
542
543         c=phi[1][n_ra]-0.5*omega_02-k0*Aphi[1][n_ra];
544
545         /* Enthalpy */
546
```

```
547         for(i=1;i<=KDIV;i++)
548          {
549           for(j=1;j<=NDIV;j++)   h[i][j]=c-phi[i][j]+0.5*omega_02*r[j]*r[j]
550                         *(1.0-mu[i]*mu[i])+r[j]*pow((1-mu[i]*mu[i]) ,0.5)*Aphi[i][j]*k0;
551          }
552
553         h_max = max(h);
554
555
556         /* New density */
557
558         for(i=1;i<=KDIV;i++)
559          {
560           for(j=1;j<=NDIV;j++)
561            {
562             if(h[i][j]>=0)
563              {
564               if(n_index==0.0) rho[i][j]=1.0;
565                else
566                 rho[i][j]=pow(h[i][j]/h_max ,n_index);
567              }
568              else rho[i][j]=0.0;
569            }
570          }
571
572        dif1=fabs(h_max_old - h_max);
573        dif2=fabs(omega_02_old - omega_02);
574        dif3=fabs(c_old - c);
575
576        h_max_old=h_max;
577        omega_02_old=omega_02;
578        c_old=c;
579
580       }
581    }
582
583
584
585    /*************************************************************************/
586    /* Compute various quantities.                                           */
587    /*************************************************************************/
```

```
588  void comp()
589  {
590   int i,
591       j,
592       j_b;                    /* last point inside star */
593
594   double s=0.0,               /* individual term in sum */
595          sum=0.0,             /* intermediate sum */
596
597          dv1[NDIV+1],         /* integrated quantity in volume */
598          dm1[KDIV+1],         /* integrated quantity in mass */
599          dmi1[KDIV+1],        /* integrated quantity in moment of inertia */
600          dw1[KDIV+1],         /* integrated quantity in potential energy */
601          dp1[NDIV+1],         /* integrated quantity in \Pi */
602          alpha,               /* slope of line in interpolation for boundary */
603
604          Aphiderr[KDIV+1][NDIV+1],  /* Aphi derivative with respect to r */
605          dmag1[KDIV+1],      /* integrated quantity in Magnetic Energy */
606          dmag2[KDIV+1], /* integrated quantity in Magnetic Energy */
607          Emag1=0.0,          /* part of sum in Magnetic energy */
608          Emag2=0.0;
609
610   /* Initialize variables */
611
612   m=0.0;
613   v=0.0;
614   mi=0.0;
615   am=0.0;
616   ke=0.0;
617   w=0.0;
618   pint=0.0;
619
620
621
622       /* Maximum pressure */
623
624       p_max=h_max/(1.0+n_index);
625
626
627       /* Boundary */
628
```

```
629         for(i=1;i<=KDIV;i++)
630          {
631           j=1;
632           r_bound1[i]=0.0;
633           while(h[i][j]>=0)
634            {
635             j_b=j;              /* last point j_b inside star */
636             j++;
637            }
638
639           alpha= (KDIV-1)/RMAX*(h[i][j_b+1]-h[i][j_b]);    /* slope */
640           r_bound1[i]=r[j_b]-h[i][j_b]/alpha;    /* linear interpolation */
641          }
642
643
644         /* CORRECT DENSITY OUTSIDE STAR */
645
646         for(i=1;i<=KDIV;i++)
647           for(j=1;j<=NDIV;j++) {
648                if(r[j]>r_bound1[i]) {
649                  rho[i][j]=0.0;
650                }
651           }
652
653
654         /* Volume */
655
656         for(i=1;i<=KDIV-2;i+=2)
657          {
658           s= (4.0/9.0)*PI*(mu[i+1]-mu[i])*(pow(r_bound1[i],3.0)
659                       +4.0*pow(r_bound1[i+1],3.0)+pow(r_bound1[i+2],3.0));
660           v+=s;
661          }
662
663
664        /* Mass */
665
666        if(n_index==0.0) m=v;
667         else
668          {
669           for(j=1;j<=NDIV;j++)
```

```
670             {
671              for(i=1;i<=KDIV-2;i+=2)
672               {
673                s=((mu[i+1]-mu[i])/3.0)*(rho[i][j]+4*rho[i+1][j]+rho[i+2][j]);
674                sum+=s;
675               }
676                dm1[j]=sum;
677                sum=0.0;
678             }
679
680            for(j=1;j<=NDIV-2;j+=2)
681             {
682              s=(4.0/3.0)*PI*(r[j+1]-r[j])*(r[j]*r[j]*dm1[j]
683                                            +4.0*r[j+1]*r[j+1]*dm1[j+1]
684                                            +r[j+2]*r[j+2]*dm1[j+2]);
685              m+=s;
686             }
687           }
688
689
690           /* Moment of inertia */
691
692           for(j=1;j<=NDIV;j++)
693            {
694             for(i=1;i<=KDIV-2;i+=2)
695              {
696               s=((mu[i+1]-mu[i])/3.0)*(  (1.0-mu[i]*mu[i])*rho[i][j]
697                                        +4.0*(1.0-mu[i+1]*mu[i+1])*rho[i+1][j]
698                                         +(1.0-mu[i+2]*mu[i+2])*rho[i+2][j]);
699               sum+=s;
700              }
701               dmi1[j]=sum;
702               sum=0.0;
703            }
704
705
706           for(j=1;j<=NDIV-2;j+=2)
707            {
708             s=(4.0/3.0)*PI*(r[j+1]-r[j])*(pow(r[j],4.0)*dmi1[j]
709                                           +4.0*pow(r[j+1],4.0)*dmi1[j+1]
710                                           +pow(r[j+2],4.0)*dmi1[j+2]);
```

```
711             mi+=s;
712           }
713
714
715         /* Angular momentum */
716
717         am=mi*pow(omega_02,0.5);
718
719
720         /* Kinetic energy */
721
722         ke=0.5*mi*omega_02;
723
724
725         /* Gravitational energy */
726
727         for(j=1;j<=KDIV;j++)
728           {
729             for(i=1;i<=KDIV-2;i+=2)
730               {
731                 s=((mu[i+1]-mu[i])/3.0)*(rho[i][j]*phi[i][j]
732                                          +4.0*rho[i+1][j]*phi[i+1][j]
733                                           +rho[i+2][j]*phi[i+2][j]);
734                 sum+=s;
735               }
736                 dw1[j]=sum;
737                 sum=0.0;
738           }
739
740
741         for(j=1;j<=NDIV-2;j+=2)
742           {
743             s=(2.0/3.0)*PI*(r[j+1]-r[j])*(r[j]*r[j]*dw1[j]
744                                 +4.0*r[j+1]*r[j+1]*dw1[j+1]
745                                 +r[j+2]*r[j+2]*dw1[j+2]);
746             w+=s;
747           }
748
749
750       /* pressure */
751
```

```
752         if(n_index==0.0)
753          {
754           for(j=1;j<=NDIV;j++)
755            {
756             for(i=1;i<=KDIV;i++)
757              {
758               if(h[i][j]>0.0) pres[i][j]=h[i][j];
759                else
760                 pres[i][j]=0.0;
761              }
762            }
763          }
764          else
765           {
766            for(j=1;j<=NDIV;j++)
767             {
768              for(i=1;i<=KDIV;i++)
769               {
770                pres[i][j]=pow(rho[i][j],1.0+1.0/n_index)*p_max;
771               }
772             }
773           }
774
775
776           /* Integral of pressure */
777
778           for(j=1;j<=NDIV;j++)
779            {
780             for(i=1;i<=KDIV-2;i+=2)
781              {
782               s=((mu[i+1]-mu[i])/3.0)*(pres[i][j]+4*pres[i+1][j]+pres[i+2][j]);
783               sum+=s;
784              }
785               dp1[j]=sum;
786               sum=0.0;
787            }
788
789
790           for(j=1;j<=NDIV-2;j+=2)
791            {
792             s=(4.0/3.0)*PI*(r[j+1]-r[j])*(r[j]*r[j]*dp1[j]
```

```
793                         + 4.0*r[j+1]*r[j+1]*dp1[j+1] + r[j+2]*r[j+2]*dp1[j+2]);
794          pint+=s;
795         }
796
797
798
799
800
801
802        /* Magnetic Energy */
803
804                 /* Aphi der r */
805
806                         for(i=1;i<=KDIV;i++)
807                    {
808                   for(j=3;j<=NDIV-2;j++)
809                      {
810        Aphiderr[i][j]=  ((NDIV-1)/(RMAX))*(1.0/12.0)*(Aphi[i][j-2]-8.0*Aphi[i][j-1]
811                     + 8.0*Aphi[i][j+1] - Aphi[i][j+2]);
812                                          }
813
814
815        Aphiderr[i][1]=((NDIV-1)/(RMAX))*((-25.0/12.0)*Aphi[i][1]+4.0*Aphi[i][2]-3.0*Aphi[i][3]
816                       +(4.0/3.0)*Aphi[i][4]-(1.0/4.0)*Aphi[i][5] );
817
818        Aphiderr[i][2]=((NDIV-1)/(RMAX))*((-25.0/12.0)*Aphi[i][2]+4.0*Aphi[i][3]-3.0*Aphi[i][4]
819                        +(4.0/3.0)*Aphi[i][5]-(1.0/4.0)*Aphi[i][6]);
820
821        Aphiderr[i][NDIV-1]=((NDIV-1)/(RMAX))*((25.0/12.0)*Aphi[i][KDIV-1]
822                             -4.0*Aphi[i][NDIV-2]+3.0*Aphi[i][NDIV-3]
823                             -(4.0/3.0)*Aphi[i][NDIV-4]+(1.0/4.0)*Aphi[i][NDIV-5]);
824
825        Aphiderr[i][NDIV]=((NDIV-1)/(RMAX))*((25.0/12.0)*Aphi[i][NDIV]
826                           -4.0*Aphi[i][NDIV-1]+3.0*Aphi[i][NDIV-2]
827                           -(4.0/3.0)*Aphi[i][NDIV-3]+(1.0/4.0)*Aphi[i][NDIV-4]);
828                                          }
829                sum=0.0;
830        for(j=1;j<=NDIV;j++)
831         {
832          for(i=1;i<=KDIV-2;i+=2)
833            {
```

```
834             s=((mu[i+1]-mu[i])/3.0)*(pow((1-mu[i]*mu[i]),0.5)*rho[i][j]*Aphi[i][j]
835                     +4.0*pow((1-mu[i+1]*mu[i+1]),0.5)*rho[i+1][j]*Aphi[i+1][j]
836                     +pow((1-mu[i+2]*mu[i+2]),0.5)*rho[i+2][j]*Aphi[i+2][j]);
837             sum+=s;
838            }
839             dmag1[j]=sum;
840             sum=0.0;
841           }
842
843
844         for(j=1;j<=NDIV-2;j+=2)
845           {
846            s=(4.0/3.0)*PI*(r[j+1]-r[j])*(pow(r[j],3.0)*dmag1[j]
847                     +4.0*pow(r[j+1],3.0)*dmag1[j+1]
848                     +pow(r[j+2],3.0)*dmag1[j+2]);
849            Emag1+=s;
850           }
851         sum=0.0;
852                 /*----------*/
853           for(j=1;j<=NDIV;j++)
854           {
855            for(i=1;i<=KDIV-2;i+=2)
856             {
857              s=((mu[i+1]-mu[i])/3.0)*(pow((1-mu[i]*mu[i]),0.5)*rho[i][j]*Aphiderr[i][j]
858                     +4.0*pow((1-mu[i+1]*mu[i+1]),0.5)*rho[i+1][j]*Aphiderr[i+1][j]
859                     +pow((1-mu[i+2]*mu[i+2]),0.5)*rho[i+2][j]*Aphiderr[i+2][j]);
860              sum+=s;
861             }
862              dmag2[j]=sum;
863              sum=0.0;
864           }
865
866
867         for(j=1;j<=NDIV-2;j+=2)
868           {
869            s=(4.0/3.0)*PI*(r[j+1]-r[j])*(pow(r[j],4.0)*dmag2[j]
870            +4.0*pow(r[j+1],4.0)*dmag2[j+1]
871            +pow(r[j+2],4.0)*dmag2[j+2]);
872            Emag2+=s;
873           }
874
```

```
875
876         Emag=(Emag1+Emag2)*k0;
877
878
879       /* Keplerian angular velocity */
880
881         omega_k2 = -((NDIV-1)/(12.0*RMAX))*(h[1][n_ra-2]-8.0*h[1][n_ra-1]
882                                           + 8.0*h[1][n_ra+1] - h[1][n_ra+2])
883                    + omega_02
884                    +k0*(Aphi[1][n_ra]+Aphiderr[1][n_ra] );
885
886
887
888
889
890
891         /* Virial test */
892
893         vt=fabs(2*ke+w+3*pint+Emag)/fabs(w);
894
895
896
897
898
899     }
900
901
902     void compute_B(void)
903     {
904
905     int i,
906         j,
907         k;
908
909     double  Aphiderr[KDIV+1][NDIV+1],
910             Aphidermu[KDIV+1][NDIV+1],
911                  Br[KDIV+1][NDIV+1],
912               Btheta[KDIV+1][NDIV+1],
913                 Bphi[KDIV+1][NDIV+1],
914                 max,
915                 s,
```

```
916                   sum,
917                  Emtor[KDIV+1];
918
919
920   max=surf_max();
921
922
923
924       /* Aphi der r */
925
926           for(i=1;i<=KDIV;i++)
927               {
928                for(j=3;j<=NDIV-2;j++)
929                 {
930         Aphiderr[i][j]=  ((NDIV-1)/(RMAX))*(1.0/12.0)*(Aphi[i][j-2]-8.0*Aphi[i][j-1]
931                             + 8.0*Aphi[i][j+1] - Aphi[i][j+2]);
932                 }
933
934
935         Aphiderr[i][1]=((NDIV-1)/(RMAX))*((-25.0/12.0)*Aphi[i][1]+4.0*Aphi[i][2]-3.0*Aphi[i][3]
936                        +(4.0/3.0)*Aphi[i][4]-(1.0/4.0)*Aphi[i][5]  );
937
938         Aphiderr[i][2]=((NDIV-1)/(RMAX))*((-25.0/12.0)*Aphi[i][2]+4.0*Aphi[i][3]-3.0*Aphi[i][4]
939                        +(4.0/3.0)*Aphi[i][5]-(1.0/4.0)*Aphi[i][6]);
940
941         Aphiderr[i][NDIV-1]=((NDIV-1)/(RMAX))*((25.0/12.0)*Aphi[i][NDIV-1]
942                               -4.0*Aphi[i][NDIV-2]+3.0*Aphi[i][NDIV-3]
943                              -(4.0/3.0)*Aphi[i][NDIV-4]+(1.0/4.0)*Aphi[i][NDIV-5]);
944
945         Aphiderr[i][NDIV]=((NDIV-1)/(RMAX))*((25.0/12.0)*Aphi[i][NDIV]
946                            -4.0*Aphi[i][NDIV-1]+3.0*Aphi[i][NDIV-2]
947                            -(4.0/3.0)*Aphi[i][NDIV-3]+(1.0/4.0)*Aphi[i][NDIV-4]);
948                               }
949
950     /* Aphi der mu */
951
952           for(j=1;j<=NDIV;j++)
953               {
954                for(i=3;i<=KDIV-2;i++)
955                   {
956         Aphidermu[i][j]=  ((KDIV-1)/(1.0))*(1.0/12.0)*(Aphi[i-2][j]-8.0*Aphi[i-1][j]
```

```
957                                 + 8.0*Aphi[i+1][j] - Aphi[i+2][j]);
958                                                 }
959
960
961         Aphidermu[1][j]=((KDIV-1)/(1.0))*((-25.0/12.0)*Aphi[1][j]+4.0*Aphi[2][j]-3.0*Aphi[3][j]
962                             +(4.0/3.0)*Aphi[4][j]-(1.0/4.0)*Aphi[5][j] );
963
964         Aphidermu[2][j]=((KDIV-1)/(1.0))*((-25.0/12.0)*Aphi[2][j]+4.0*Aphi[3][j]-3.0*Aphi[4][j]
965                           +(4.0/3.0)*Aphi[5][j]-(1.0/4.0)*Aphi[6][j]);
966
967         Aphidermu[KDIV-1][j]=((KDIV-1)/(1.0))*((25.0/12.0)*Aphi[KDIV-1][j]
968                                 -4.0*Aphi[KDIV-2][j]+3.0*Aphi[KDIV-3][j]
969                                -(4.0/3.0)*Aphi[KDIV-4][j]+(1.0/4.0)*Aphi[KDIV-5][j]);
970
971         Aphidermu[KDIV][j]=((KDIV-1)/(1.0))*((25.0/12.0)*Aphi[KDIV][j]
972                                 -4.0*Aphi[KDIV-1][j]+3.0*Aphi[KDIV-2][j]
973                                 -(4.0/3.0)*Aphi[KDIV-3][j]+(1.0/4.0)*Aphi[KDIV-4][j]);
974                         }
975
976
977
978     for(j=1;j<=NDIV;j++)
979             {
980              for(i=1;i<=KDIV;i++)
981               {
982       Br[i][j]= ( (mu[i]*Aphi[i][j])/(r[j]*pow(1.0-mu[i]*mu[i],0.5)) )
983                   -(pow(1.0-mu[i]*mu[i],0.5)/r[j])*Aphidermu[i][j];
984       Btheta[i][j]=-Aphi[i][j]/r[j] -Aphiderr[i][j];
985       Bphi[i][j]=mag_f[i][j]/(r[j]*pow(1.0-mu[i]*mu[i],0.5));
986
987               }
988             }
989
990
991      /* Br extrapolation */
992      for(i=1;i<=KDIV;i++){
993
994                 Br[i][1]= ( (r[1] -r[3])*(r[1] -r[4]) ) * (Br[i][2] )/
995                           ( (r[2]-r[3])*(r[2]-r[4]) ) +
996
997                           ( (r[1] -r[2])*(r[1] -r[4]) ) * (Br[i][3] )/
```

```
998                                 ( (r[3]-r[2])*(r[3]-r[4]) ) +
999
1000                                ( (r[1] -r[2])*(r[1] -r[3]) ) * (Br[i][4] )/
1001                                ( (r[4]-r[2])*(r[4]-r[3]) );
1002
1003                                    }
1004            for(j=1;j<=NDIV;j++){
1005
1006      Br[KDIV-4][j]=    ( (mu[KDIV-4] -mu[KDIV-2-4])*(mu[KDIV-4] -mu[KDIV-3-4]) ) *
             (Br[KDIV-1-4][j] )/
1007                        ( (mu[KDIV-1-4]-mu[KDIV-2-4])*(mu[KDIV-1-4]-mu[KDIV-3-4]) ) +
1008
1009                        ( (mu[KDIV-4] -mu[KDIV-1-4])*(mu[KDIV-4] -mu[KDIV-3-4]) ) *
                               (Br[KDIV-2-4][j] )/
1010                        ( (mu[KDIV-2-4]-mu[KDIV-1-4])*(mu[KDIV-2-4]-mu[KDIV-3-4]) ) +
1011
1012                        ( (mu[KDIV-4] -mu[KDIV-1-4])*(mu[KDIV-4] -mu[KDIV-2-4]) ) *
                               (Br[KDIV-3-4][j] )/
1013                        ( (mu[KDIV-3-4]-mu[KDIV-1-4])*(mu[KDIV-3-4]-mu[KDIV-2-4]) );
1014
1015      Br[KDIV-3][j]=Br[KDIV-4][j];
1016      Br[KDIV-2][j]= Br[KDIV-4][j];
1017      Br[KDIV-1][j]= Br[KDIV-4][j];
1018      Br[KDIV][j]= Br[KDIV-4][j];
1019
1020
1021
1022                                    }
1023
1024    /* Btheta extrapolation */
1025     for(i=1;i<=KDIV;i++){
1026
1027                 Btheta[i][1]=( (r[1] -r[3])*(r[1] -r[4]) ) * (Btheta[i][2] )/
1028                                ( (r[2]-r[3])*(r[2]-r[4]) ) +
1029
1030                               ( (r[1] -r[2])*(r[1] -r[4]) ) * (Btheta[i][3] )/
1031                                ( (r[3]-r[2])*(r[3]-r[4]) ) +
1032
1033                               ( (r[1] -r[2])*(r[1] -r[3]) ) * (Btheta[i][4] )/
1034                                ( (r[4]-r[2])*(r[4]-r[3]) );
1035
```

```
1036
1037                                    }
1038      for(j=1;j<=NDIV;j++){
1039
1040          Btheta[KDIV-4][j]=     ( (mu[KDIV-4]   -mu[KDIV-2-4])*(mu[KDIV-4]   -mu[KDIV-3-4])   ) *
                 (Btheta[KDIV-1-4][j] )/
1041                              ( (mu[KDIV-1-4]-mu[KDIV-2-4])*(mu[KDIV-1-4]-mu[KDIV-3-4])   ) +
1042
1043                              ( (mu[KDIV-4]   -mu[KDIV-1-4])*(mu[KDIV-4]   -mu[KDIV-3-4])   ) *
                                    (Btheta[KDIV-2-4][j] )/
1044                              ( (mu[KDIV-2-4]-mu[KDIV-1-4])*(mu[KDIV-2-4]-mu[KDIV-3-4])   ) +
1045
1046                              ( (mu[KDIV-4]   -mu[KDIV-1-4])*(mu[KDIV-4]   -mu[KDIV-2-4])   ) *
                                    (Btheta[KDIV-3-4][j] )/
1047                              ( (mu[KDIV-3-4]-mu[KDIV-1-4])*(mu[KDIV-3-4]-mu[KDIV-2-4])   );
1048
1049        Btheta[KDIV-3][j]=Btheta[KDIV-4][j];
1050        Btheta[KDIV-2][j]= Btheta[KDIV-4][j];
1051        Btheta[KDIV-1][j]= Btheta[KDIV-4][j];
1052        Btheta[KDIV][j]= Btheta[KDIV-4][j];
1053
1054
1055
1056                                  }
1057
1058
1059     /* Bphi extrapolation */
1060      for(i=1;i<=KDIV;i++){
1061
1062                  Bphi[i][1]=( (r[1]-r[3])*(r[1]-r[4])*(r[1]-r[5])   ) * (Bphi[i][2] )/
1063                             ( (r[2]-r[3])*(r[2]-r[4])*(r[2]-r[5]) ) +
1064
1065                            ( (r[1]-r[2])*(r[1]-r[4])*(r[1]-r[5])   ) * (Bphi[i][3] )/
1066                             ( (r[3]-r[2])*(r[3]-r[4])*(r[3]-r[5]) ) +
1067
1068                             ( (r[1]-r[2])*(r[1]-r[3])*(r[1]-r[5])   ) * (Bphi[i][4] )/
1069                             ( (r[4]-r[2])*(r[4]-r[3])*(r[4]-r[5]) ) +
1070
1071                             ( (r[1]-r[2])*(r[1]-r[3])*(r[1]-r[4])   ) * (Bphi[i][5] )/
1072                             ( (r[5]-r[2])*(r[5]-r[3])*(r[5]-r[4]) ) ;
1073
```

```
1074                                    }
1075              for(j=1;j<=NDIV;j++){
1076
1077        Bphi[KDIV][j]= (  (mu[KDIV]-mu[KDIV-2])*(mu[KDIV]-mu[KDIV-3])*(mu[KDIV]-mu[KDIV-4])  )
1078                              *(Bphi[KDIV-1][j]  )/
1079                        (
                                (mu[KDIV-1]-mu[KDIV-2])*(mu[KDIV-1]-mu[KDIV-3])*(mu[KDIV-1]-mu[KDIV-4])
                                )+
1080
1081                        (  (mu[KDIV]-mu[KDIV-1])*(mu[KDIV]-mu[KDIV-3])*(mu[KDIV]-mu[KDIV-4])  )
1082                         *(Bphi[KDIV-2][j]  )/
1083                        (
                                (mu[KDIV-2]-mu[KDIV-1])*(mu[KDIV-2]-mu[KDIV-3])*(mu[KDIV-2]-mu[KDIV-4])
                                )+
1084
1085                        (  (mu[KDIV]-mu[KDIV-1])*(mu[KDIV]-mu[KDIV-2])*(mu[KDIV]-mu[KDIV-4])  )
1086                         *(Bphi[KDIV-3][j]  )/
1087                        (
                                (mu[KDIV-3]-mu[KDIV-1])*(mu[KDIV-3]-mu[KDIV-2])*(mu[KDIV-3]-mu[KDIV-4])
                                )+
1088
1089                        (  (mu[KDIV]-mu[KDIV-1])*(mu[KDIV]-mu[KDIV-2])*(mu[KDIV]-mu[KDIV-3])  )
1090                         *(Bphi[KDIV-4][j]  )/
1091                        (
                                (mu[KDIV-4]-mu[KDIV-1])*(mu[KDIV-4]-mu[KDIV-2])*(mu[KDIV-4]-mu[KDIV-3])
                                );
1092
1093                                    }
1094
1095
1096
1097    for(j=1;j<=NDIV;j++)
1098            {
1099             for(i=1;i<=KDIV;i++)
1100              {
1101              B_pol_norm[i][j]= pow( (Br[i][j]*Br[i][j]+Btheta[i][j]*Btheta[i][j])  ,0.5);
1102              B_tor_norm[i][j]=fabs(Bphi[i][j]);
1103              }
1104            }
1105
1106    // Magnetic toroidal energy
```

```
1107        s=0.0;
1108        sum=0.0;
1109        Emagtor=0.0;
1110    for(j=1;j<=NDIV;j++)
1111            {
1112             for(i=1;i<=KDIV-2;i+=2)
1113              {
1114               s=((mu[i+1]-mu[i])/3.0)*(pow(B_tor_norm[i][j],2.0)
1115               +4.0*pow(B_tor_norm[i+1][j],2.0)
1116               +pow(B_tor_norm[i+2][j],2.0) );
1117               sum+=s;
1118              }
1119               Emtor[j]=sum;
1120               sum=0.0;
1121            }
1122
1123           for(j=1;j<=NDIV-2;j+=2)
1124            {
1125             s=(4.0/3.0)*PI*(r[j+1]-r[j])*(r[j]*r[j]*Emtor[j]
1126                                          +4.0*r[j+1]*r[j+1]*Emtor[j+1]
1127                                          +r[j+2]*r[j+2]*Emtor[j+2]);
1128             Emagtor+=s;
1129            }
1130            Emagtor=Emagtor/(8.0*PI);
1131
1132    }
1133
1134    void compute_xz(void)
1135
1136    { int i,
1137        j;
1138
1139
1140    for(j=1;j<=NDIV;j++)
1141            {
1142             for(i=1;i<=KDIV;i++)
1143              {
1144               x[i][j]=r[j]*pow(1.0-mu[i]*mu[i],0.5);
1145               z[i][j]=r[j]*mu[i];
1146              }
1147            }
```

```
1148
1149
1150
1151    }
1152
1153    void compute_theta(void)
1154    {
1155
1156    int i;
1157
1158    for(i=1;i<=KDIV;i++)
1159            {
1160              theta[i]=acos(mu[i]);
1161
1162    }
1163
1164
1165    }
1166
1167
1168    /*************************************************************************/
1169    /* Print header for table of results                                     */
1170    /*************************************************************************/
1171    void print_header(void)
1172    {
1173       printf("POLYTROPES (rigid rotation)\n");
1174       printf("-------------------------------------------------------------------------");
1175       printf("----------------------\n");
1176       printf("N=%2.1f                          \n",n_index);
1177       printf("-------------------------------------------------------------------------");
1178       printf("----------------------\n");
1179       printf("    r_B    | O_0^2    | O_K^2    |     M     |    V      |     J     | T/|W|    |");
1180       printf(" Virial \n |    Pmax | 3U/|W|     | Emag/|W|       |");
1181       printf("-------------------------------------------------------------------------");
1182       printf("-------------------------\n");
1183
1184    }
1185
1186
1187    void print_header2(void)
1188
```

```
1189    {
1190
1191
1192        printf("\n\n");
1193        printf("N=%2.1f                          \n",n_index);
1194        printf("-------------------------------------------------------------------------");
1195        printf("----------------------\n");
1196        printf("    r_B    | Emag/|W|| U/|W|  |  T/|W|  |    W     | Omega^2 |     M     |");
1197        printf(" Virial \n k0    |     c     |Omega_02/Omega_K2");
1198        printf("-------------------------------------------------------------------------");
1199        printf("----------------------\n");
1200
1201
1202
1203          }
1204
1205
1206    /*************************************************************************/
1207    main(int argc, char **argv)
1208    {int i,                  /* counter */
1209         n_rb;               /* grid position of pole */
1210
1211     double r_ratio;         /* axes ratio */
1212
1213
1214        /* MAKE GRID */
1215
1216        make_grid();
1217
1218
1219        /* DEFAULT VALUES */
1220
1221        n_index=1.0;
1222        r_ratio=1.0;
1223
1224
1225        /* READ OPTIONS */
1226
1227        for(i=1;i<argc;i++)
1228            if(argv[i][0]=='-'){
1229              switch(argv[i][1]){
```

```
1230                    case 'N':
1231                                sscanf(argv[i+1],"%lf",&n_index);
1232                                break;
1233
1234                    case 'r':
1235                                sscanf(argv[i+1],"%lf",&r_ratio);
1236                                break;
1237              }
1238           }
1239
1240
1241        /* GRID POSITION OF EQUATOR */
1242
1243        n_ra=(NDIV-1)/RMAX+1;
1244
1245
1246        /* GRID POSITION OF POLE */
1247
1248        n_rb=r_ratio*n_ra;
1249
1250
1251        printf("%d %d\n",n_ra,n_rb);
1252
1253        /* COMPUTE SPHERICAL EQUILIBRIUM MODEL */
1254
1255        guess_density();
1256        guess_Aphi();
1257
1258        /* COMPUTE NECESSARY FUNCTIONS FOR INTEGRALS */
1259
1260        comp_f_2n_p_2n();
1261
1262
1263        /* PRINT HEADER */
1264
1265      FILE *fres;
1266         fres = fopen("res.txt", "w");
1267
1268    int numb,numb1,ii;
1269    char ch;
1270        /* MAIN LOOP */
```

```
1271
1272      iterate(n_rb, n_index);
1273      comp();
1274     compute_B();
1275
1276       if(r_ratio !=1.0) {
1277         print_header();
1278
1279    printf("%3.2e  %3.2e  %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %d\n",
1280                  r[n_rb], omega_02/(4.0*PI), omega_k2/(4.0*PI), m, v, am, ke/fabs(w),
1281                 vt, p_max,pint/fabs(w),Emag/fabs(w),counter);
1282
1283         printf("\n\n");
1284
1285          print_header2();
1286
1287          printf("%3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e  %3.2e \n",
1288                   r[n_rb],Emag/fabs(w),pint/fabs(w),ke/fabs(w),fabs(w)/(4.0*PI),
1289                  omega_02/(4.0*PI),m,vt,k0/(pow(4.0*PI,0.5)),c/(4.0*PI),omega_02/omega_k2);
1290
1291
1292         fprintf(fres,"%3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e
              \n",
1293                   r[n_rb],Emag/fabs(w),pint/fabs(w),ke/fabs(w),fabs(w)/(4.0*PI),
1294                        omega_02/(4.0*PI),m,vt,k0/(pow(4.0*PI,0.5)),c/(4.0*PI),omega_02/omega_k2);
1295
1296        printf("iterations      %d",counter);
1297
1298        printf("\n\n");
1299
1300        printf("   Emag_tor/Emag %f    ",Emagtor/Emag);
1301        printf("\n\n");
1302
1303          }
1304       else {
1305       printf("%3.2e   ----      ----    %3.2e %3.2e   ----      ----    %3.2e %3.2e %3.2e
            %3.2e \n",
1306                   r[n_rb], m, v, am, vt, p_max,Emag/fabs(w));
1307
1308              printf("\n\n");
1309
```

```
1310          print_header2();
1311
1312          printf("%3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e \n",
1313                  r[n_rb],Emag/fabs(w),n_index*pint/fabs(w),ke/fabs(w),fabs(w),omega_02,m,vt);
1314
1315          fprintf(fres,"%3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e %3.2e \n",
1316                  r[n_rb],Emag/fabs(w),pint/fabs(w),ke/fabs(w),fabs(w)/(4.0*PI),
1317                  omega_02/(4.0*PI),m,vt,k0/(pow(4.0*PI,0.5)),c/(4.0*PI));
1318
1319
1320        printf("\n");
1321        printf("iterations      %d",counter);
1322        printf("\n\n");
1323
1324
1325                  }
1326
1327     compute_B();
1328    compute_xz();
1329    compute_theta();
1330    //fclose(fres);
1331
1332    /* Export in files */
1333    FILE *fr,*fmu,*frho,*fh,*fAphi,*frbound,*fphi,*fBpol,*fBtor,*fx,*fz,*ftheta;
1334
1335          fr = fopen("r.txt", "w");
1336         fmu = fopen("mu.txt", "w");
1337         frho = fopen("rho.txt", "w");
1338         fh = fopen("h.txt", "w");
1339         fAphi = fopen("Aphi.txt", "w");
1340         frbound = fopen("rbound.txt", "w");
1341         fphi=fopen("phi.txt","w");
1342         fBpol=fopen("Bpol.txt","w");
1343         fBtor=fopen("Btor.txt","w");
1344         fx=fopen("x.txt","w");
1345         fz=fopen("z.txt","w");
1346         ftheta=fopen("theta.txt","w");
1347
1348         for(numb=1;numb<=KDIV;numb++){
1349           for(numb1=1;numb1<=NDIV;numb1++){
1350
```

```
1351            fprintf(frho," %f ",rho[numb][numb1]);
1352            fprintf(fh," %f ",h[numb][numb1]);
1353            fprintf(fAphi," %f ",Aphi[numb][numb1]);
1354            fprintf(fphi," %f ",phi[numb][numb1]);
1355            fprintf(fBpol," %f ",B_pol_norm[numb][numb1]);
1356            fprintf(fBtor," %f ",B_tor_norm[numb][numb1]);
1357            fprintf(fx," %f ",x[numb][numb1]);
1358            fprintf(fz," %f ",z[numb][numb1]);
1359
1360
1361               } fprintf(frho,"\n");
1362                 fprintf(fh,"\n");
1363                 fprintf(fAphi,"\n");
1364                 fprintf(fphi,"\n");
1365                 fprintf(fBpol,"\n");
1366                 fprintf(fBtor,"\n");
1367                 fprintf(fx,"\n");
1368                 fprintf(fz,"\n");
1369    }
1370
1371      for(numb=1;numb<=NDIV;numb++) fprintf(fr," %f \n",r[numb]);
1372      for(numb=1;numb<=NDIV;numb++) fprintf(frbound," %f \n",r_bound1[numb]);
1373       for(numb=1;numb<=KDIV;numb++) fprintf(fmu," %f \n",mu[numb]);
1374       for(numb=1;numb<=KDIV;numb++) fprintf(ftheta," %f \n",theta[numb]);
1375
1376    fclose(fr);
1377    fclose(fmu);
1378    fclose(frho);
1379    fclose(fh);
1380    fclose(fAphi);
1381    fclose(frbound);
1382    fclose(fphi);
1383    fclose(fBpol);
1384    fclose(fBtor);
1385    fclose(fx);
1386    fclose(fz);
1387    fclose(ftheta);
1388    }
```

Listing C.1: Source code for rotating single fluid magnetized neutron stars (normal MHD)

C.2 Rotating magnetized superconductive neutron stars

```
1  /*****************************************************************/
2  /*                          NSCONT231a.C                         */
3  /*                                                               */
4  /* Newtonian models of SC MHD two fluid                          */
5  /*          rotating polytropic neutron stars.                   */
6  /*                                                               */
7  /* Author : K.Palapanidis                                        */
8  /* (based on an earlier nonmagnetized version by N. Stergioulas, */
9  /*  1993)                                                        */
10 /*                                                               */
11 /* Date   : July 2014                                            */
12 /*****************************************************************/
13
14 //***************************************************************/
15 // Wherever CC is mentioned it is assumed that the values are    */
16 // from the core side of the boudary (last point in core)        */
17 // except for those situations where we                          */
18 // interpolate to find a closer to the exact value.              */
19 /*                                                               */
20 /* Usage:  nscont231a  -N Np_index -r r_ratio                    */
21 /*****************************************************************/
22
23
24 #include <stdio.h>
25 #include <string.h>
26 #include <math.h>
27
28
29 #define KDIV 511                    /* grid points in mu-direction */
30 #define NDIV 511                  /* grid points in r-direction */
31 //#define RMAX 3.0       // greater area integration
32 #define RMAX 1.06666666666666666     /* multiply r by 16.0/15.0 */
33 //#define LMAX 16          // greater area integration better accuracy
34 #define LMAX 16               /* 1/2 of max. term in Legendre poly.*/
35 #define PI 3.141592653589793
36 #define Sqrt_4PI 3.5449077018110318
37
38  FILE *fdata,
39
```

```
40          *fconv,
41          *fconvcc,
42          *fconvQ,
43          *fmagcharge,
44          *fFlux;
45
46    int n_ra,                    /* grid position of r_a=1.0 */
47        cc_ra,                // Grid position of neutron equatiorial radius  n_n_ra=0.9*n_ra
48        counter=0,
49        point[KDIV+1],       /* tester*/
50
51        last_r_p[KDIV+1],            // last point j inside protons (star) (KDIV in total)
52       last_mu_p[NDIV+1],            // last point i inside protons (star) (NDIV in total)
53        last_r_n[KDIV+1],           // last point j inside neutrons (KDIV in total)
54       last_mu_n[NDIV+1],            // last point i inside neutrons (NDIV in total)
55       last_r_cc[KDIV+1],           // last point j inside core (KDIV in total)
56      last_mu_cc[NDIV+1],          // last point i inside core  (NDIV in total)
57
58        max_count=0,
59        counter1=0,
60        counter2=0;
61
62
63
64    double      eps1=0.0,
65               mu[KDIV+1],                    /* grid points in mu-direction */
66                r[NDIV+1],                     /* grid points in r-direction */
67           varpi[KDIV+1][NDIV+1],           //   r*sin(theta)=r*(1-mu^2)^(1/2)
68
69             f2n[LMAX+1][NDIV+1][NDIV+1],  /* function f_2n */
70             p2n[LMAX+1][KDIV+1],          /* function p_2n */
71           f2n_1[LMAX+1][NDIV+1][NDIV+1],  /* function f_2n-1 */
72         p1_2n_1[LMAX+1][KDIV+1],          /* Assoc Legendre P^1_2n-1 */
73
74           rho_p[KDIV+1][NDIV+1],          /*proton density */
75           rho_n[KDIV+1][NDIV+1],          /*proton density */
76             rho[KDIV+1][NDIV+1],
77             phi[KDIV+1][NDIV+1],          /* gravitational potential */
78               u[KDIV+1][NDIV+1],         /* streamfunction u */
79      u_new_star[KDIV+1][NDIV+1],          // intermediate value for u
80         check_u[KDIV+1][NDIV+1],
```

```
81              F[KDIV+1][NDIV+1],          /* streamfunction integrated value*/
82          chem_p[KDIV+1][NDIV+1],         /*chemical potential for protons */
83          chem_n[KDIV+1][NDIV+1],         /*chemical potential for neutrons */
84          p_ener[KDIV+1][NDIV+1],         /*EOS for protons */
85          n_ener[KDIV+1][NDIV+1],        /* EOS for neutrons */
86          Pi_fun[KDIV+1][NDIV+1],         /* Pi function of Magnetic field */
87
88        mag_f_N[KDIV+1][NDIV+1],        /* Magnetic superconductivity function y(u) */
89            M_N[KDIV+1][NDIV+1],
90        df_N_du[KDIV+1][NDIV+1],
91        dM_N_du[KDIV+1][NDIV+1],
92          mag_f[KDIV+1][NDIV+1],
93              y[KDIV+1][NDIV+1],
94           M_SC[KDIV+1][NDIV+1],
95          df_du[KDIV+1][NDIV+1],
96          dy_du[KDIV+1][NDIV+1],
97            Bcc[KDIV+1],
98       dB_du_cc[KDIV+1],                 // Bcc derivative with respect to u
99          rcc_0[KDIV+1][NDIV+1],    // Crust-core boundary values with no interpolation.
                Derived r[last_r_cc[i]]
100             M[KDIV+1][NDIV+1],     // Piece wise M function for the integral equations of
                    motion
101         dB_du[KDIV+1][NDIV+1],
102       checkQ[KDIV+1][NDIV+1],
103          divB[KDIV+1][NDIV+1],
104     BFlux_surf,    // B flux at the surface of the star
105
106
107       r_bound[KDIV+1],               // Star Surface
108     r_neutron[KDIV+1],        // Neutron surface
109           r_cc[KDIV+1],              // Crust-core boundary surface
110
111
112
113         Np_index,                       /* index N in polytropic EOS for protons */
114         Nn_index,                       /* index N in polytropic EOS for protons */
115         x_p_0,                        // rho_p_max  (in paper  X_p(0)
116         chem_p_max,                          /* Proton chemical potential max */
117         chem_n_max,                          /* Neutron chemical potential max */
118
119         omega_02,                       /* omega_0^2 */
```

```
120         C_p,                            // proton integration constant
121         C_dif,                          // difference Euler integration constant
122
123         undrlx_c,                       // underrelaxation parameter omega
124
125         v,                              /* volume */
126         m,                              /* mass */
127         mi,                             /* moment of inertia */
128         am,                             /* angular momentum */
129         ke,                             /* kinetic energy */
130         w,                              /* gravitational energy */
131         Pi_p,                             /* proton internal energy */
132         Pi_n,                             // neutron internal energy
133         Emag,                             /* Magnetic Energy   */
134         Emagtor=0.0,                          // Toroidal Magnetic energy
135         magratio,
136         magcharge,
137
138         vt,                             /* Virial test = | 2T+W+3(U_n+U_p)+E_mag | / |W| */
139         omega_k2,                       /* omega_Kepler^2 */
140
141
142         k0,               /* k(u)=k0 function*/
143         alpha_c,        /* alpha constant */
144         zeta,                       // magnetic function constant
145         h_c,                           // superconductivity constant
146        eps,
147        conv=0.0,                // u convergence parameter
148      conv_cc=0.0,
149      convQ=0.0,
150      convFlux=0.0,
151      BFlux_surf_check=0.0,
152
153        grad_u_r[KDIV+1][NDIV+1],
154   grad_u_theta[KDIV+1][NDIV+1],
155    grad_u_norm[KDIV+1][NDIV+1],
156
157      grad_Pi_r[KDIV+1][NDIV+1],
158  grad_Pi_theta[KDIV+1][NDIV+1],
159
160             Br[KDIV+1][NDIV+1],
```

```
            Btheta[KDIV+1][NDIV+1],
              Bphi[KDIV+1][NDIV+1],
       B_pol_norm[KDIV+1][NDIV+1],
       B_tor_norm[KDIV+1][NDIV+1],
                 B[KDIV+1][NDIV+1];


//&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
//*****************************************************************************/
//                          Functions
//*****************************************************************************/

//************************************************************ Unit Step function
double unit_step(double x, double x0)
{
     if(x>=x0) return 1.0;
     else return 0.0;
}

//************************************************************ Grid initialization
void make_grid(void)
{
  int i,          /* counter in mu-direction */
      j;          /* counter in r-direction */

  for (i=1;i<=KDIV;i++) mu[i]=(i-1.0)/(KDIV-1.0);
  for (j=1;j<=NDIV;j++) r[j]=RMAX*(j-1.0)/(NDIV-1.0);

   for(i=1;i<=KDIV;i++)
       {
       for(j=1;j<=NDIV;j++)
         {
          varpi[i][j]= r[j]*pow(1.0-mu[i]*mu[i],0.5);
         }
       }

}

//************************************************************ Adding densities
// rho=rho_p+rho_n
void total_density(void)
```

```
202   {
203     int i,
204         j;
205
206      for(i=1;i<=KDIV;i++)
207           {
208           for(j=1;j<=NDIV;j++)
209             {
210              rho[i][j]= rho_p[i][j]+rho_n[i][j];
211             }
212           }
213   }
214
215
216   //*************************************************************** Densities
          initialization
217   void guess_density(void)
218   {
219    int i,
220        j;
221
222
223     for(j=1;j<=NDIV;j++)                  // Proton density. evaluated until surface of star
224      {
225       if(j<=n_ra) rho_p[1][j]=1.0;        /* First find rho for mu=0 */
226        else
227         rho_p[1][j]=0.0;
228
229       for(i=1;i<=KDIV;i++) rho_p[i][j]=rho_p[1][j];
230      }
231
232     //Evaluate spherical cc throught 0.9r_a compute rho_n
233
234     for(j=1;j<=NDIV;j++)                  // Neutron density. evaluated until cc boundary
235      {
236       if(j<=cc_ra) rho_n[1][j]=1.0;
237        else
238         rho_n[1][j]=0.0;
239
240       for(i=1;i<=KDIV;i++) rho_n[i][j]=rho_n[1][j];
241      }
```

```
242
243   }
244
245   //********************************************************** Initialization of
          chemical potentials
246   //The chemical potentials are initialized so that we have an
247   //initial boundary for the neutrons and protons
248   void guess_last_points(void){
249
250     int i,
251         j;
252
253
254      for(j=1;j<=NDIV;j++)                  // Proton density. evaluated until surface of star
255      {
256       if(j<=n_ra) chem_p[1][j]=1.0;        /* First find rho for mu=0 */
257        else
258         chem_p[1][j]=-1.0;
259
260       for(i=1;i<=KDIV;i++) chem_p[i][j]=chem_p[1][j];
261      }
262
263     //Evaluate spherical neutron surface throught 0.9r_a compute rho_n
264
265     for(j=1;j<=NDIV;j++)                  // Neutron density. evaluated until neutron surface
266      {
267       if(j<=cc_ra) chem_n[1][j]=1.0;
268        else
269         chem_n[1][j]=-1.0;
270
271       for(i=1;i<=KDIV;i++) chem_n[i][j]=chem_n[1][j];
272      }
273
274      // we initialize the cc boundary to be the same as the neutrons boundary
275      // later it can deviate from the neutron surface
276
277
278      for(i=1;i<=KDIV;i++)
279      {
280         last_r_cc[i]=cc_ra;
281      }
```

```
282
283       for(j=1;j<=NDIV;j++)
284        {
285         if(j<=cc_ra) last_mu_cc[j]=KDIV;
286         else last_mu_cc[j]=0;
287        }
288
289         for(i=1;i<=KDIV;i++)
290        {
291            last_r_p[i]=n_ra;
292        }
293
294       for(j=1;j<=NDIV;j++)
295        {
296         if(j<=n_ra) last_mu_p[j]=KDIV;
297         else last_mu_p[j]=0;
298        }
299
300
301
302   }
303
304   /*************************************************************************/
305   /* A first guess for the density distribution is stored in the array      */
306   /* Aphi[i][j]. It corresponds to a uniform-density nonrotating sphere.    */
307   /*************************************************************************/
308   void guess_u(void)
309   {
310    int i,                              /* counter */
311        j;                              /* counter */
312
313     for(j=1;j<=KDIV;j++)
314      {
315       for(i=1;i<=NDIV;i++) {
316         u[i][j]=1.0;
317        check_u[i][j]=1.0;
318       }
319      }
320
321
322
```

```
323   }
324
325
326   /*****************************************************************/
327   // Returns the Legendre polynomial of degree n, evaluated at x.
328   /*****************************************************************/
329   double legendre( int n, double x )
330   {
331     int i;            /* counter */
332
333     double p,         /* Legendre polynomial of order n */
334            p_1,       /*    "          "       "     "   n-1*/
335            p_2;       /*    "          "       "     "   n-2 */
336
337
338     p_2=1.0;
339     p_1=x;
340
341    if(n>=2)
342     { for(i=2;i<=n;i++)
343       {
344        p=(x*(2.0*i-1.0)*p_1 - (i-1.0)*p_2)/i;
345        p_2=p_1;
346        p_1=p;
347       }
348     return p;
349     } else
350       { if (n==1) return p_1;
351         else return p_2;
352       }
353   }
354
355   /*****************************************************************/
356   // Returns the Associated Legendre polynomial P_n^m  m=1, evaluated at x.
357   /*****************************************************************/
358
359   double Assoc_legendre( int n, double x )
360   {
361     int i;            /* counter */
362
363     double p,         /*Assoc. Legendre polynomial P_l^1 */
```

```
364             p_1,        /*     "           "       "   P_(l-1)^1*/
365             p_2;        /*     "           "       "   P_(l-2)^1 */
366
367
368     p_2=-pow( (1.0-pow(x,2.0) ),0.5);
369     p_1=-3*x*pow( (1.0-pow(x,2.0) ),0.5);
370
371    if(n>=3)
372     { for(i=3;i<=n;i++)
373       {
374        p=( (2.0*i-1.0)/(i-1.0) ) * x * p_1 - ((i) /(i-1.0) ) *p_2;
375        p_2=p_1;
376        p_1=p;
377       }
378     return p;
379     } else
380       { if (n==1) return p_2;
381         else return p_1;
382       }
383   }
384
385   /*****************************************************************************/
386   // Computing the radial component f_n(r,r')*r' of the polynomial expansion of 1/|r-r'|
387   /*****************************************************************************/
388
389   double f_n(int n, double vec_r[NDIV+1],int k,int j)  /* n=degree of the Legendre
          polynomials */
390   {
391                   /* k counter for r' */
392       /* j counter for r */
393
394     double f;
395
396               if (k<j)  f=pow(vec_r[k],n+2.0)/pow(vec_r[j],n+1.0) ;
397                else
398                   { if( j==1) f=0;
399                      else f=pow(vec_r[j],n)/pow(vec_r[k],n-1.0);
400                   }
401               return f;
402
403
```

```
404       }
405
406     /*****************************************************************/
407     // Since the grid points are fixed, we can compute the functions
408     /* f_n(r',r)*r'^2  and P_2n(mu) once at the beginning.           */
409     /*****************************************************************/
410     void comp_f_2n_p_2n(void)
411     {
412
413       int m,n,                    /* 2n=degree of the Legendre polynomials */
414           k,                    /* counter for r' */
415           j,                    /* counter for r */
416           i;                    /* counter for P_2n*/
417      double vec_r[NDIV+1];
418                   /* temporary storage of f_2n */
419
420     //for(m=1;m<=NDIV;m++){ vec_r[m]=r[m];
421     //   printf("   aaa %f",r[m]);
422     //}
423
424       for (n=0;n<=LMAX;n++)
425        {
426          for (k=1;k<=NDIV;k++)
427           {
428             for (j=1;j<=NDIV;j++)
429              {
430                f2n[n][k][j]=f_n(2*n,r,k,j);
431                f2n_1[n][k][j]=f_n(2*n-1,r,k,j);
432              }
433           }
434        }
435
436      for(i=1;i<=KDIV;i++)
437       {
438         for(n=0;n<=LMAX;n++) { p2n[n][i] = legendre(2*n,mu[i]);
439                                p1_2n_1[n][i] = Assoc_legendre( 2*n-1, mu[i]);
440                               }
441
442       }
443     }
444
```

```
/*********************************************************************/
// Returns the maximum value in a KDIV x NDIV array.
/*********************************************************************/
double max(double array[KDIV+1][NDIV+1])
{
 int i,
     j;
 double max_val;      /* intermediate max. value */

 max_val=array[1][1];

 for(i=1;i<=KDIV;i++)
  {
   for(j=1;j<=NDIV;j++)
    {
     if(array[i][j]>max_val)   max_val=array[i][j];
    }
  }
 return max_val;
}


//&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

//******************************************** Calculate r derivative in star

double deriv_r(double array[KDIV+1][NDIV+1], int i0, int j0){

    int i,
        j,
 i_last,
 j_last,
 i_cc,
 j_cc;

   double deriv;

  i_last = last_mu_p[j0];
  j_last =  last_r_p[i0];
    i_cc = last_mu_cc[j0];
    j_cc = last_r_cc[i0];
```

```
486
487     //#########################################################################################
488         if((j0<=j_cc)&&(i0<=i_cc)) // Point inside the core
489
490           {
491
492
493      if(j_cc==1) // If the point is the only one in that mu then the derivative is equal
             to the previous mu derivative. (j remains the same)
494                {
495               deriv=deriv_r(array,i0-1,j0);
496
497        }
498
499       else if (j_cc<6)  // If the last point in star is less than 6 points from the core
             then use less accurate (less points) formulas
500          {
501
502          if ((j0==1)||(j0==2)) {
503                        deriv=((NDIV-1)/(RMAX))*((-1.0)*array[i0][j0]+1.0*array[i0][j0+1] );
504                   }
505
506        else if ((j0==j_cc-1)||(j0==j_cc)){
507
508                          deriv=((NDIV-1)/(RMAX))*((1.0)*array[i0][j0]-1.0*array[i0][j0-1] );
509                 }
510
511        else     {
512                        deriv=((NDIV-1)/(RMAX))*(-0.5*array[i0][j0-1]+0.5*array[i0][j0+1] );
513               }
514
515          }
516
517
518
519       else  //If the last point is more than 6 points then use the normal formulas
520       {
521
522        if ((j0==1)||(j0==2))
523                           {       deriv=((NDIV-1)/(RMAX))*((-25.0/12.0)*array[i0][j0]
524                                        +4.0*array[i0][j0+1]-3.0*array[i0][j0+2]
```

```
525                             +(4.0/3.0)*array[i0][j0+3]-(1.0/4.0)*array[i0][j0+4] );
526                                }
527
528       else if ((j0==j_cc-1)||(j0==j_cc))
529                             {     deriv=((NDIV-1)/(RMAX))*((25.0/12.0)*array[i0][j0]
530                                  -4.0*array[i0][j0-1]+3.0*array[i0][j0-2]
531                     -(4.0/3.0)*array[i0][j0-3]+(1.0/4.0)*array[i0][j0-4]);
532        }
533
534
535        else  {     deriv= ((NDIV-1)/(RMAX))*(1.0/12.0)*(array[i0][j0-2]-8.0*array[i0][j0-1]
536                               + 8.0*array[i0][j0+1] - array[i0][j0+2]);
537                    }
538
539        }
540
541        }
542
543        //*****************************************************************
544
545       else  if(chem_p[i0][j0]>=0.0) // Point inside the crust
546        {
547
548
549        if((j_last-j_cc)==1) // If the point is the only one in that mu then the derivative
               is equal to the previous mu derivative. (j remains the same)
550                  {
551           if(i0>1)
552
553                deriv=deriv_r(array,i0-1,j0);
554                 else deriv=deriv_r(array,i0,j0-1);
555           }
556
557        else if ((j_last-j_cc)<6)  // If the last point in star is less than 6 points from
               the core then use less accurate (less points) formulas
558           {
559
560          if ((j0==j_cc+1)||(j0==j_cc+2)) {
561                       deriv=((NDIV-1)/(RMAX))*((-1.0)*array[i0][j0]+1.0*array[i0][j0+1] );
562                    }
563
```

```
564         else if ((j0==j_last-1)||(j0==j_last)){
565
566                         deriv=((NDIV-1)/(RMAX))*((1.0)*array[i0][j0]-1.0*array[i0][j0-1] );
567                 }
568
569         else     {
570                         deriv=((NDIV-1)/(RMAX))*(-0.5*array[i0][j0-1]+0.5*array[i0][j0+1] );
571                 }
572
573           }
574
575
576       else  //If the last point is more than 6 points then use the normal formulas
577       {
578
579        if ((j0==j_cc+1)||(j0==j_cc+2))
580                           {       deriv=((NDIV-1)/(RMAX))*((-25.0/12.0)*array[i0][j0]
581                                 +4.0*array[i0][j0+1]-3.0*array[i0][j0+2]
582                               +(4.0/3.0)*array[i0][j0+3]-(1.0/4.0)*array[i0][j0+4] );
583                                 }
584
585       else if ((j0==j_last-1)||(j0==j_last))
586                           {       deriv=((NDIV-1)/(RMAX))*((25.0/12.0)*array[i0][j0]
587                                 -4.0*array[i0][j0-1]+3.0*array[i0][j0-2]
588                     -(4.0/3.0)*array[i0][j0-3]+(1.0/4.0)*array[i0][j0-4]);
589        }
590
591
592        else  {     deriv= ((NDIV-1)/(RMAX))*(1.0/12.0)*(array[i0][j0-2]-8.0*array[i0][j0-1]
593                                   + 8.0*array[i0][j0+1] - array[i0][j0+2]);
594                     }
595
596        }
597
598        }
599       //#############################################################################
600        else  // Point outside the star
601           {
602
603       if( (NDIV-j_last)==1) // If the point is the only one in that mu then the derivative
```

```
604                                 // is equal to the previous mu derivative. (j remains the
                                        same)
605                 {
606               if(i0>1)
607                 deriv=deriv_r(array,i0-1,j0);
608               // printf("Last point in r direction is j=1");
609       //    printf("r %d",counter1++);
610             else deriv=deriv_r(array,i0,j0-1);
611                   }
612
613       else if ( (NDIV-j_last)<6)  // If the last point in star is less than 6 points from
              the outer
614                                 //     boundary then use less accurate (less points)
                                        formulas
615         {
616
617         if ( (j0==(j_last+1)) || (j0==(j_last+2)) ) {
618                       deriv=((NDIV-1)/(RMAX))*((-1.0)*array[i0][j0]+1.0*array[i0][j0+1] );
619                   }
620
621       else if ((j0==NDIV-1)||(j0==NDIV)){
622
623                         deriv=((NDIV-1)/(RMAX))*((1.0)*array[i0][j0]-1.0*array[i0][j0-1] );
624                 }
625
626       else     {
627                       deriv=((NDIV-1)/(RMAX))*(-0.5*array[i0][j0-1]+0.5*array[i0][j0+1] );
628             }
629
630         }
631
632
633
634      else  //If the last point is more than 6 points then use the normal formulas
635      {
636
637       if ( (j0==(j_last+1)) || (j0==(j_last+2)) )
638                         {       deriv=((NDIV-1)/(RMAX))*((-25.0/12.0)*array[i0][j0]
639                                   +4.0*array[i0][j0+1]-3.0*array[i0][j0+2]
640                             +(4.0/3.0)*array[i0][j0+3]-(1.0/4.0)*array[i0][j0+4] );
641                                }
```

```
642
643       else if ((j0==NDIV-1)||(j0==NDIV))
644                                 {     deriv=((NDIV-1)/(RMAX))*((25.0/12.0)*array[i0][j0]
645                                     -4.0*array[i0][j0-1]+3.0*array[i0][j0-2]
646                         -(4.0/3.0)*array[i0][j0-3]+(1.0/4.0)*array[i0][j0-4]);
647       }
648
649
650       else  {     deriv= ((NDIV-1)/(RMAX))*(1.0/12.0)*(array[i0][j0-2]-8.0*array[i0][j0-1]
651                                     + 8.0*array[i0][j0+1] - array[i0][j0+2]);
652                     }
653
654       }
655
656
657           }
658
659   if (fabs(deriv)<=eps1) deriv=0.0;
660
661   return deriv ;
662
663   }
664
665   double deriv_mu(double array[KDIV+1][NDIV+1], int i0, int j0){
666
667           int i,
668            j,
669    i_last,
670    j_last,
671    i_cc,
672    j_cc;
673
674       double deriv;
675
676     i_last = last_mu_p[j0];
677     j_last =  last_r_p[i0];
678       i_cc = last_mu_cc[j0];
679       j_cc = last_r_cc[i0];
680
681
682       //#########################################################################################
```

```
683
684         if((j0<=j_cc)&&(i0<=i_cc)) // Point inside the core
685         {
686
687        if(i_cc==1) // If the point is the only one in that r then the derivative is equal
               to the previous r derivative. (i remains the same)
688                 {
689           if (j0>1)
690              deriv=deriv_mu(array,i0,j0-1);
691
692          else deriv=deriv_mu(array,i0-1,j0);
693                   }
694
695        else if (i_cc<6)  // If the last point in star is less than 6 points from the
               boundary then use less accurate (less points) formulas
696          {
697
698          if ((i0==1)||(i0==2)) {
699                          deriv=((KDIV-1)/(1.0))*((-1.0)*array[i0][j0]+1.0*array[i0+1][j0] );
700                    }
701
702         else if ((i0==(i_cc-1))||(i0==i_cc)){
703                             deriv=((KDIV-1)/(1.0))*((1.0)*array[i0][j0]-1.0*array[i0-1][j0]
                                    );
704                           }
705
706         else     {
707                         deriv=((KDIV-1)/(1.0))*(-0.5*array[i0-1][j0]+0.5*array[i0+1][j0] );
708               }
709
710          }
711
712
713
714       else  //If the last point is more than 6 points then use the normal formulas
715       {
716
717         if ((i0==1)||(i0==2))
718                             {       deriv=((KDIV-1)/(1.0))*((-25.0/12.0)*array[i0][j0]
719                                        +4.0*array[i0+1][j0]-3.0*array[i0+2][j0]
720                                       +(4.0/3.0)*array[i0+3][j0]-(1.0/4.0)*array[i0+4][j0] );
```

```
721                                 }
722
723       else if ((i0==i_cc-1)||(i0==i_cc))
724                                 {    deriv=((KDIV-1)/(1.0))*((25.0/12.0)*array[i0][j0]
725                                        -4.0*array[i0-1][j0]+3.0*array[i0-2][j0]
726                                 -(4.0/3.0)*array[i0-3][j0]+(1.0/4.0)*array[i0-4][j0]) ;
727        }
728
729
730        else  {      deriv=((KDIV-1)/(1.0))*(1.0/12.0)*(array[i0-2][j0]-8.0*array[i0-1][j0]
731                                    + 8.0*array[i0+1][j0] - array[i0+2][j0]);
732                    }
733
734       }
735
736
737        }
738
739
740         else   if(chem_p[i0][j0]>=0.0) // Point inside the star
741       {
742
743
744       if((i_last-i_cc)==1) // If the point is the only one in that r then the derivative
745                            // is equal to the previous r derivative. (i remains the same)
746                  {
747          if (j0>1)
748                deriv=deriv_mu(array,i0,j0-1);
749                 //  printf("mu %d",counter2++);
750         else deriv=deriv_mu(array,i0-1,j0);
751                    }
752
753       else if ((i_last-i_cc)<6)  // If the last point in star is less than 6 points
754                                  // from the boundary then use less accurate (less points)
                                       formulas
755          {
756
757          if ((i0==(i_cc+1))||(i0==(i_cc+2))) {
758                          deriv=((KDIV-1)/(1.0))*((-1.0)*array[i0][j0]+1.0*array[i0+1][j0] );
759                     }
760
```

```
761       else if ((i0==i_last-1)||(i0==i_last)){
762                           deriv=((KDIV-1)/(1.0))*((1.0)*array[i0][j0]-1.0*array[i0-1][j0]
                                ;
763                         }
764
765       else     {
766                       deriv=((KDIV-1)/(1.0))*(-0.5*array[i0-1][j0]+0.5*array[i0+1][j0] );
767             }
768
769         }
770
771
772
773     else  //If the last point is more than 6 points then use the normal formulas
774     {
775
776      if ((i0==(i_cc+1))||(i0==(i_cc+2)))
777                         {       deriv=((KDIV-1)/(1.0))*((-25.0/12.0)*array[i0][j0]
778                                    +4.0*array[i0+1][j0]-3.0*array[i0+2][j0]
779                                   +(4.0/3.0)*array[i0+3][j0]-(1.0/4.0)*array[i0+4][j0] );
780                               }
781
782     else if ((i0==(i_last-1))||(i0==i_last))
783                           {    deriv=((KDIV-1)/(1.0))*((25.0/12.0)*array[i0][j0]
784                                    -4.0*array[i0-1][j0]+3.0*array[i0-2][j0]
785                            -(4.0/3.0)*array[i0-3][j0]+(1.0/4.0)*array[i0-4][j0]) ;
786      }
787
788
789      else  {      deriv=((KDIV-1)/(1.0))*(1.0/12.0)*(array[i0-2][j0]-8.0*array[i0-1][j0]
790                                  + 8.0*array[i0+1][j0] - array[i0+2][j0]);
791                  }
792
793      }
794
795      }
796     ////////////////////////////////////////////////////////////////////////////////
797      else  // Point outside the star
798         {
799
800     if( (KDIV-i_last)==1) // If the point is the only one in that mu then
```

```
801                                  //the derivative is equal to the previous mu derivative.
                                        (j remains the same)
802                 {
803                deriv=deriv_mu(array,i0,j0-1);
804                    }
805
806       else if ( (KDIV-i_last)<6)  // If the last point in star is less than 6 points
807                                    // from the outer boundary then use less accurate (less
                                        points) formulas
808          {
809
810          if ( (i0==(i_last+1)) || (i0==(i_last+2)) ) {
811                        deriv=((KDIV-1)/(1.0))*((-1.0)*array[i0][j0]+1.0*array[i0+1][j0] );
812                   }
813
814          else if ((i0==KDIV-1)||(i0==KDIV)){
815
816                           deriv=((KDIV-1)/(1.0))*((1.0)*array[i0][j0]-1.0*array[i0-1][j0] );
817                  }
818
819          else     {
820                         deriv=((KDIV-1)/(1.0))*(-0.5*array[i0-1][j0]+0.5*array[i0+1][j0] );
821                }
822
823          }
824
825
826
827       else  //If the last point is more than 6 points then use the normal formulas
828       {
829
830          if ( (i0==(i_last+1)) || (i0==(i_last+2)) )
831                           {       deriv=((KDIV-1)/(1.0))*((-25.0/12.0)*array[i0][j0]
832                                    +4.0*array[i0+1][j0]-3.0*array[i0+2][j0]
833                                     +(4.0/3.0)*array[i0+3][j0]-(1.0/4.0)*array[i0+4][j0] );
834                                 }
835
836          else if ((i0==KDIV-1)||(i0==KDIV))
837                             {      deriv=((KDIV-1)/(1.0))*((25.0/12.0)*array[i0][j0]
838                                     -4.0*array[i0-1][j0]+3.0*array[i0-2][j0]
839                              -(4.0/3.0)*array[i0-3][j0]+(1.0/4.0)*array[i0-4][j0]) ;
```

```
840       }
841
842
843       else  {     deriv=((KDIV-1)/(1.0))*(1.0/12.0)*(array[i0-2][j0]-8.0*array[i0-1][j0]
844                                 + 8.0*array[i0+1][j0] - array[i0+2][j0]);
845                   }
846
847       }
848
849
850           }
851
852     if (fabs(deriv)<=eps1) deriv=0.0;
853     return deriv ;
854
855     }
856
857
858     double interpolate(double array[KDIV+1][NDIV+1],int i0, int j0){ // higher order core
          crust out
859
860
861
862       int  i,
863            j,
864      i_last, //refering to the last point inside surface of the star
865      j_last,
866
867      i_cc, // refering to the last point inside cc
868      j_cc;
869
870       double interp;
871
872       j_last=last_r_p[i0];
873       i_last=last_mu_p[j0];
874
875         j_cc=last_r_cc[i0];
876         i_cc=last_mu_cc[j0];
877
878
879
```

```
880     //#########################################################################################
881      if ((j0<=j_cc)&&(i0<=i_cc)){ //Point inside core
882
883          // The corners of the grid (1,1) (KDIV,1)
884          if  ((i0==1)&&(j0==1)) interp=array[i0][j0+1];
885
886          else if ( (i0==KDIV)&&(j0==1) ) interp=array[i0-1][j0];
887
888           // Calculate values for j=1. mu axis. centre of the star
889          else if (j0==1){
890
891                if (j_cc==1) // If the point is the only one in that mu then the
892                                    // value is equal to the previous mu value. (j remains
                                        the same)
893                                      {
894           interp=array[i0-1][j0];
895
896                                  }
897                else if(j_cc==2) {
898                           interp=array[i0][j0+1];
899                              }
900
901                     else    {  // Use 3 points formula
902
903                  interp=( (r[j0]  -r[j0+2])*(r[j0]  -r[j0+3])  ) * (array[i0][j0+1] )/
904                                                ( (r[j0+1]-r[j0+2])*(r[j0+1]-r[j0+3])  ) +
905
906                                                ( (r[j0]  -r[j0+1])*(r[j0]  -r[j0+3])  ) *
                                                    (array[i0][j0+2] )/
907                                                ( (r[j0+2]-r[j0+1])*(r[j0+2]-r[j0+3])  ) +
908
909                                                ( (r[j0]  -r[j0+1])*(r[j0]  -r[j0+2])  ) *
                                                    (array[i0][j0+3] )/
910                                                ( (r[j0+3]-r[j0+1])*(r[j0+3]-r[j0+2])  );
911
912                                   }
913                           }
914
915            // Calculate values for i=1. r axis. equatorial plane
916           else if (i0==1) {
917
```

```
918             if(i_cc==1) // If the point is the only one in that r then the value is
                    equal to the previous r value. (i remains the same)
919                                 {
920                     interp=array[i0][j0-1];
921
922                                 }
923                 else if (i_cc==2) interp=array[i0+1][j0];  //if the last point is 2
                        points from the r axis then give it the next value.
924
925
926                 else    {  // Use 2 points formula
927
928                                 interp=      ( (mu[i0]  -mu[i0+2])*(mu[i0]  -mu[i0+3])
                                        ) * (array[i0+1][j0] )/
929                                             ( (mu[i0+1]-mu[i0+2])*(mu[i0+1]-mu[i0+3])  )
                                                +
930
931                                             ( (mu[i0]  -mu[i0+1])*(mu[i0]  -mu[i0+3])  )
                                                * (array[i0+2][j0] )/
932                                             ( (mu[i0+2]-mu[i0+1])*(mu[i0+2]-mu[i0+3])  )
                                                +
933
934                                             ( (mu[i0]  -mu[i0+1])*(mu[i0]  -mu[i0+2])  )
                                                * (array[i0+3][j0] )/
935                                             ( (mu[i0+3]-mu[i0+1])*(mu[i0+3]-mu[i0+2])  );
936
937                                               }
938                         }
939
940          // Calculate values for i=KDIV. r (polar) axis. No search for other
                possibilities is needed.
941             else if (i0==KDIV)   {
942
943                             interp=      ( (mu[i0]  -mu[i0-2])*(mu[i0]  -mu[i0-3])  ) *
                                    (array[i0-1][j0] )/
944                                             ( (mu[i0-1]-mu[i0-2])*(mu[i0-1]-mu[i0-3])  )
                                                +
945
946                                             ( (mu[i0]  -mu[i0-1])*(mu[i0]  -mu[i0-3])  )
                                                * (array[i0-2][j0] )/
```

```
947                                                 ( (mu[i0-2]-mu[i0-1])*(mu[i0-2]-mu[i0-3]) )
                                                      +
948
949                                                 ( (mu[i0] -mu[i0-1])*(mu[i0] -mu[i0-2]) )
                                                      * (array[i0-3][j0] )/
950                                                 ( (mu[i0-3]-mu[i0-1])*(mu[i0-3]-mu[i0-2]) );
951                             }
952
953           // Calculate values on the cc boundary from the inside.
954
955
956           // Calculate for the other points
957          else    {
958              if(j_cc==1) // If the point is the only one in that mu then the value is
                     equal to the previous mu value. (j remains the same)
959                                 {
960         interp=array[i0-1][j0];
961
962                             }
963                  else if (j_cc==2) interp=array[i0][j0-1];  //if the last point is 2
                         points from the mu axis then give it the next value.
964
965
966                  else    {  // Use 2 points formula
967
968              interp=( (r[j0] -r[j0+2])*(r[j0] -r[j0+3])*(r[j0] -r[j0+4]) ) *
                     (array[i0][j0+1] )/
969                      ( (r[j0+1]-r[j0+2])*(r[j0+1]-r[j0+3])*(r[j0+1]-r[j0+4]) ) +
970
971                      ( (r[j0] -r[j0+1])*(r[j0] -r[j0+3])*(r[j0] -r[j0+4]) ) *
                         (array[i0][j0+2] )/
972                      ( (r[j0+2]-r[j0+1])*(r[j0+2]-r[j0+3])*(r[j0+2]-r[j0+4]) ) +
973
974                      ( (r[j0] -r[j0+1])*(r[j0] -r[j0+2])*(r[j0] -r[j0+4]) ) *
                         (array[i0][j0+3] )/
975                      ( (r[j0+3]-r[j0+1])*(r[j0+3]-r[j0+2])*(r[j0+3]-r[j0+4]) ) +
976
977                      ( (r[j0] -r[j0+1])*(r[j0] -r[j0+2])*(r[j0] -r[j0+3]) ) *
                         (array[i0][j0+4] )/
978                      ( (r[j0+4]-r[j0+1])*(r[j0+4]-r[j0+2])*(r[j0+4]-r[j0+3]) );
979
```

```
980                                                     }
981                     }
982
983
984     }
985
986     else if(chem_p[i0][j0]>=0.0) // Point inside the crust
987      {
988           // The corners of the crust area
989          if  ((i0==1)&&(j0==n_ra)) interp=array[i0][j0-1];
990
991          else if ((i0==KDIV)&&(j0==(j_cc+1)) ) interp=array[i0][j0-1];
992
993           // Calculate values for i=1. r axis. equatorial plane
994          else if (i0==1) {
995
996                if(i_last==1) // If the point is the only one in that r then
997                                    // the value is equal to the previous r value. (i
                                        remains the same)
998                                      {
999                         interp=array[i0][j0-1];
1000
1001                          }
1002                    else if (i_last==2) interp=array[i0+1][j0];  //if the last point is
                            2 points from the r axis then give it the next value.
1003
1004
1005                    else    {  // Use 2 points formula
1006
1007                              interp=       ( (mu[i0]  -mu[i0+2])*(mu[i0]  -mu[i0+3])  )
                                      * (array[i0+1][j0] )/
1008                                          ( (mu[i0+1]-mu[i0+2])*(mu[i0+1]-mu[i0+3])  )
                                              +
1009
1010                                          ( (mu[i0]  -mu[i0+1])*(mu[i0]  -mu[i0+3])  )
                                              * (array[i0+2][j0] )/
1011                                          ( (mu[i0+2]-mu[i0+1])*(mu[i0+2]-mu[i0+3])  )
                                              +
1012
1013                                          ( (mu[i0]  -mu[i0+1])*(mu[i0]  -mu[i0+2])  )
                                              * (array[i0+3][j0] )/
```

```
1014                                        ( (mu[i0+3]-mu[i0+1])*(mu[i0+3]-mu[i0+2]) );
1015
1016                                          }
1017                                  }
1018
1019
1020            // Calculate values for i=KDIV. r (polar) axis.
1021          else if (i0==KDIV)   {
1022
1023                    if((KDIV-i_cc)==1) // If the point is the only one in that
1024                                       // r then the value is equal to the previous r
                                              value. (i remains the same)
1025                                     {
1026            interp=array[i0][j0-1];
1027                                       //   printf("r %d",counter1++);
1028
1029                           }
1030                    else if ((KDIV-i_cc)==2) interp=array[i0-1][j0];  //if the last
                          point is 2 points from the r axis then give it the previous value.
1031
1032
1033                    else    {  // Use 2 points formula
1034
1035                               interp=      ( (mu[i0]  -mu[i0-2])*(mu[i0]  -mu[i0-3]) )
                                      * (array[i0-1][j0] )/
1036                                           ( (mu[i0-1]-mu[i0-2])*(mu[i0-1]-mu[i0-3]) )
                                              +
1037
1038                                           ( (mu[i0]  -mu[i0-1])*(mu[i0]  -mu[i0-3]) )
                                              * (array[i0-2][j0] )/
1039                                           ( (mu[i0-2]-mu[i0-1])*(mu[i0-2]-mu[i0-3]) )
                                              +
1040
1041                                           ( (mu[i0]  -mu[i0-1])*(mu[i0]  -mu[i0-2]) )
                                              * (array[i0-3][j0] )/
1042                                           ( (mu[i0-3]-mu[i0-1])*(mu[i0-3]-mu[i0-2]) );
1043
1044                              }
1045                                }
1046
1047
```

```
1048
1049          else if (j0==j_last){ // Calculate values on the surface from the inside.
1050
1051                              if((j_last-j_cc)==1) // If the point is the only one in
                                        that mu then the value is equal to the previous mu
                                        value. (j remains the same)
1052                                  {
1053                                      interp=array[i0-1][j0];
1054
1055                             }
1056                                  else if ((j_last-j_cc)==2) interp=array[i0][j0-1];  //if
                                        the last point is 2 points from the mu axis then give it
                                        the previous value.
1057
1058                                  else    {  // Use 2 points formula
1059
1060                  interp=         ( (r[j0]  -r[j0-2])*(r[j0]  -r[j0-3])  ) * (array[i0][j0-1]
                                    )/
1061                                                  ( (r[j0-1]-r[j0-2])*(r[j0-1]-r[j0-3])  ) +
1062
1063                                                  ( (r[j0]  -r[j0-1])*(r[j0]  -r[j0-3])  ) *
                                                        (array[i0][j0-2] )/
1064                                                  ( (r[j0-2]-r[j0-1])*(r[j0-2]-r[j0-3])  ) +
1065
1066                                                  ( (r[j0]  -r[j0-1])*(r[j0]  -r[j0-2])  ) *
                                                        (array[i0][j0-3] )/
1067                                                  ( (r[j0-3]-r[j0-1])*(r[j0-3]-r[j0-2])  );
1068
1069                                                    }
1070                               }
1071
1072
1073          else    { // Calculate for the other points
1074                 if((j_last-j_cc)==1) // If the point is the only one in that mu then the
                          value is equal to the previous mu value. (j remains the same)
1075                                      {
1076           interp=array[i0-1][j0];
1077
1078                             }
1079                       else if ((j_last-j_cc)==2) interp=array[i0][j0+1];  //if the last
                              point is 2 points from the mu axis then give it the next value.
```

```
1080
1081
1082                    else    {  // Use 2 points formula
1083
1084            interp=      ( (r[j0]  -r[j0+2])*(r[j0]  -r[j0+3])*(r[j0]  -r[j0+4])  ) *
                    (array[i0][j0+1] )/
1085                         ( (r[j0+1]-r[j0+2])*(r[j0+1]-r[j0+3])*(r[j0+1]-r[j0+4])  ) +
1086
1087                         ( (r[j0]  -r[j0+1])*(r[j0]  -r[j0+3])*(r[j0]  -r[j0+4])  ) *
                             (array[i0][j0+2] )/
1088                         ( (r[j0+2]-r[j0+1])*(r[j0+2]-r[j0+3])*(r[j0+2]-r[j0+4])  ) +
1089
1090                         ( (r[j0]  -r[j0+1])*(r[j0]  -r[j0+2])*(r[j0]  -r[j0+4])  ) *
                             (array[i0][j0+3] )/
1091                         ( (r[j0+3]-r[j0+1])*(r[j0+3]-r[j0+2])*(r[j0+3]-r[j0+4])  ) +
1092
1093                         ( (r[j0]  -r[j0+1])*(r[j0]  -r[j0+2])*(r[j0]  -r[j0+3])  ) *
                             (array[i0][j0+4] )/
1094                         ( (r[j0+4]-r[j0+1])*(r[j0+4]-r[j0+2])*(r[j0+4]-r[j0+3])  );
1095
1096                                              }
1097         }
1098
1099
1100      }
1101     ////////////////////////////////////////////////////////////////////////////////
1102       else  // Point outside the star
1103          {
1104
1105           if  ((i0==1)&&(j0==NDIV)) interp=array[i0][j0-1]; // The corners of the grid
                  (1,NDIV) (KDIV,NDIV)
1106
1107           else if ((i0==KDIV)&&(j0==NDIV)) interp=array[i0-1][j0];
1108
1109           else if (j0==NDIV){ // Calculate values for j=NDIV. far mu axis.
1110
1111                 if((NDIV-j_last)==1) // If the point is the only one in that mu then the
                        value is equal to the previous mu value. (j remains the same)
1112                                      {
1113                                     interp=array[i0-1][j0];
1114
```

```
1115
1116                        }
1117                    else if ((NDIV-j_last)==2) interp=array[i0][j0-1];  //if the last
                            point is 2 points from the far mu axis then give it the previous
                            value.
1118
1119
1120                    else    {  // Use 2 points formula
1121
1122            interp=     ( (r[j0]  -r[j0-2])*(r[j0]  -r[j0-3])  ) * (array[i0][j0-1] )/
1123                                ( (r[j0-1]-r[j0-2])*(r[j0-1]-r[j0-3])  ) +
1124
1125                                ( (r[j0]  -r[j0-1])*(r[j0]  -r[j0-3])  ) *
                                      (array[i0][j0-2] )/
1126                                ( (r[j0-2]-r[j0-1])*(r[j0-2]-r[j0-3])  ) +
1127
1128                                ( (r[j0]  -r[j0-1])*(r[j0]  -r[j0-2])  ) *
                                      (array[i0][j0-3] )/
1129                                ( (r[j0-3]-r[j0-1])*(r[j0-3]-r[j0-2])  );
1130
1131                                  }
1132
1133                            }
1134
1135
1136        else if (i0==1) { // Calculate values for i=1. r axis. equatorial plane
1137
1138                    // Use 2 points formula
1139
1140                        interp=     ( (mu[i0]  -mu[i0+2])*(mu[i0]  -mu[i0+3])  )
                                * (array[i0+1][j0] )/
1141                                ( (mu[i0+1]-mu[i0+2])*(mu[i0+1]-mu[i0+3])  )
                                      +
1142
1143                                ( (mu[i0]  -mu[i0+1])*(mu[i0]  -mu[i0+3])  )
                                      * (array[i0+2][j0] )/
1144                                ( (mu[i0+2]-mu[i0+1])*(mu[i0+2]-mu[i0+3])  )
                                      +
1145
1146                                ( (mu[i0]  -mu[i0+1])*(mu[i0]  -mu[i0+2])  )
                                      * (array[i0+3][j0] )/
```

```
1147                                                    ( (mu[i0+3]-mu[i0+1])*(mu[i0+3]-mu[i0+2]) );
1148                                    }
1149
1150
1151
1152            else if (i0==KDIV)    {
1153                        // Calculate values for i=KDIV. r axis.
1154
1155                    if((KDIV-i_last)==1) // If the point is the only one in that r then the
                           value is equal to the previous r value. (i remains the same)
1156                                        {
1157              interp=array[i0][j0-1];
1158
1159                                 }
1160                        else if ((KDIV-i_last)==2) interp=array[i0-1][j0];  //if the last
                               point is 2 points from the r axis then give it the previous value.
1161
1162
1163                        else     {  // Use 2 points formula
1164
1165                                   interp=       ( (mu[i0]  -mu[i0-2])*(mu[i0]  -mu[i0-3]) )
                                          * (array[i0-1][j0] )/
1166                                                  ( (mu[i0-1]-mu[i0-2])*(mu[i0-1]-mu[i0-3]) )
                                                      +
1167
1168                                                  ( (mu[i0]  -mu[i0-1])*(mu[i0]  -mu[i0-3]) )
                                                        * (array[i0-2][j0] )/
1169                                                  ( (mu[i0-2]-mu[i0-1])*(mu[i0-2]-mu[i0-3]) )
                                                      +
1170
1171                                                  ( (mu[i0]  -mu[i0-1])*(mu[i0]  -mu[i0-2]) )
                                                        * (array[i0-3][j0] )/
1172                                                  ( (mu[i0-3]-mu[i0-1])*(mu[i0-3]-mu[i0-2]) );
1173
1174                                                    }
1175                                }
1176
1177              else if ((j0-j_last)==1){ // Calculate values on the boundary from the outside.
1178
1179                                    if((NDIV-j_last)==1) // If the point is the only one in
                                           that mu then the value is equal to the previous mu
```

```
                                value. (j remains the same)
1180                              {
1181          interp=array[i0-1][j0];
1182
1183
1184                        }
1185                           else if ((NDIV-j_last)==2) interp=array[i0][j0-1];  //if
                                  the last point is 2 points from the far mu axis then
                                  give it the next value.
1186
1187
1188                           else    {  // Use 2 points formula
1189
1190                 interp=       ( (r[j0]  -r[j0+2])*(r[j0]  -r[j0+3])  ) *
                        (array[i0][j0+1] )/
1191                                        ( (r[j0+1]-r[j0+2])*(r[j0+1]-r[j0+3])  ) +
1192
1193                                        ( (r[j0]  -r[j0+1])*(r[j0]  -r[j0+3])  ) *
                                              (array[i0][j0+2] )/
1194                                        ( (r[j0+2]-r[j0+1])*(r[j0+2]-r[j0+3])  ) +
1195
1196                                        ( (r[j0]  -r[j0+1])*(r[j0]  -r[j0+2])  ) *
                                              (array[i0][j0+3] )/
1197                                        ( (r[j0+3]-r[j0+1])*(r[j0+3]-r[j0+2])  );
1198
1199                                          }
1200
1201                         }
1202
1203          else    { // Calculate for the other points
1204              if((NDIV-j_last)==1) // If the point is the only one in that mu then the
                     value is equal to the previous mu value. (j remains the same)
1205                              {
1206          interp=array[i0-1][j0];
1207
1208                           }
1209                    else if ((NDIV-j_last)==2) interp=array[i0][j0-1];  //if the last
                           point is 2 points from the mu axis then give it the next value.
1210
1211
1212                    else    {  // Use 2 points formula
```

```
1213
1214                    interp=       ( (r[j0]   -r[j0-2])*(r[j0]   -r[j0-3])*(r[j0]   -r[j0-4])   )
                           * (array[i0][j0-1] )/
1215                                  ( (r[j0-1]-r[j0-2])*(r[j0-1]-r[j0-3])*(r[j0-1]-r[j0-4])   )
                                        +
1216
1217                                  ( (r[j0]   -r[j0-1])*(r[j0]   -r[j0-3])*(r[j0]   -r[j0-4])   )
                                        * (array[i0][j0-2] )/
1218                                  ( (r[j0-2]-r[j0-1])*(r[j0-2]-r[j0-3])*(r[j0-2]-r[j0-4])   )
                                        +
1219
1220                                  ( (r[j0]   -r[j0-1])*(r[j0]   -r[j0-2])*(r[j0]   -r[j0-4])   )
                                        * (array[i0][j0-3] )/
1221                                  ( (r[j0-3]-r[j0-1])*(r[j0-3]-r[j0-2])*(r[j0-3]-r[j0-4])   )
                                        +
1222
1223                                  ( (r[j0]   -r[j0-1])*(r[j0]   -r[j0-2])*(r[j0]   -r[j0-3])   )
                                        * (array[i0][j0-4] )/
1224                                  ( (r[j0-4]-r[j0-1])*(r[j0-4]-r[j0-2])*(r[j0-4]-r[j0-3])   );
1225
1226                                                    }
1227                        }
1228          }
1229
1230
1231    return interp ;
1232
1233    }
1234
1235    void fix_grid(double array[KDIV+1][NDIV+1]){ //fixing the points close to r=0
1236
1237     int i,j,j0;
1238     double value;
1239
1240       j0=10;
1241
1242      for(i=1;i<=KDIV;i++){
1243
1244          value=interpolate(array,i,j0);
1245
1246           for(j=1;j<=j0;j++) array[i][j]=value;
```

```
   }


}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

//************************************************************************ Star Surface
void star_surface(void)
{

 int   i,
       j,
       j_b;

double alpha;

      // Last r in protons
      for(i=1;i<=KDIV;i++)
        {
        j_b=1;
        for(j=1;j<=NDIV;j++)
   {
  if(chem_p[i][j]>=0.0) j_b=j;
   }
        alpha= (NDIV-1)/RMAX*(chem_p[i][j_b+1]-chem_p[i][j_b]);    /* slope */
        r_bound[i]=r[j_b]-chem_p[i][j_b]/alpha;    /* linear interpolation */
          }

}

//************************************************************************ Neutron
    Surface
void neutron_surface(void)
{

 int   i,
       j,
       j_b;

```

```
1287    double alpha;
1288
1289       // Last r in neutrons
1290          for(i=1;i<=KDIV;i++)
1291            {
1292            j_b=1;
1293            for(j=1;j<=NDIV;j++)
1294      {
1295     if(chem_n[i][j]>=0.0) j_b=j;
1296      }
1297
1298            alpha= (NDIV-1)/RMAX*(chem_n[i][j_b+1]-chem_n[i][j_b]);    /* slope */
1299            r_neutron[i]=r[j_b]-chem_n[i][j_b]/alpha;    /* linear interpolation */
1300              }
1301
1302    }
1303
1304    void cc_surface(void)
1305    {
1306      int   i,
1307            j,
1308            j_b;
1309
1310    double alpha;
1311
1312           for(i=1;i<=KDIV;i++)
1313            {
1314            j_b=0;
1315            for(j=1;j<=NDIV;j++)
1316      {
1317     if(rho_p[i][j]>=rho_p[1][cc_ra]) j_b=j;
1318      }
1319             alpha= (NDIV-1)/RMAX*(rho_p[i][j_b+1]-rho_p[i][j_b]);
1320              r_cc[i]=r[j_b]+(-rho_p[i][j_b]+rho_p[1][cc_ra])/alpha;
1321            }
1322    }
1323
1324    void compute_last_points(void){
1325
1326      int   i,
1327            j,
```

```
1328            j_b,
1329            i_b;
1330
1331        // Last r in protons
1332          for(i=1;i<=KDIV;i++)
1333            {
1334            j_b=0;
1335            for(j=1;j<=NDIV;j++)
1336       {
1337      if(chem_p[i][j]>=0.0) j_b=j;
1338       }
1339            last_r_p[i]=j_b;
1340            }
1341
1342        // Last r in neutrons
1343          for(i=1;i<=KDIV;i++)
1344            {
1345            j_b=0;
1346            for(j=1;j<=NDIV;j++)
1347       {
1348      if(chem_n[i][j]>=0.0) j_b=j;
1349       }
1350            last_r_n[i]=j_b;
1351            }
1352
1353       // Last r point in core  (CC boundary from the core side)
1354
1355          for(i=1;i<=KDIV;i++)
1356            {
1357            j_b=0;
1358            for(j=1;j<=NDIV;j++)
1359       {
1360      if(rho_p[i][j]>=rho_p[1][cc_ra]) j_b=j;
1361       }
1362            last_r_cc[i]=j_b;
1363            }
1364
1365
1366      //*************************************************************************
1367       // Last mu in protons
1368            for(j=1;j<=NDIV;j++)
```

```
1369              {
1370        i_b=0;
1371              for(i=1;i<=KDIV;i++)
1372       {
1373               if(chem_p[i][j]>=0.0) i_b=i;
1374      }
1375              last_mu_p[j]=i_b;
1376      }
1377
1378       // Last mu in neutrons
1379            for(j=1;j<=NDIV;j++)
1380              {
1381        i_b=0;
1382              for(i=1;i<=KDIV;i++)
1383       {
1384               if(chem_n[i][j]>=0.0) i_b=i;
1385      }
1386              last_mu_n[j]=i_b;
1387      }
1388
1389        // Last mu point in core (CC boundary from the core side)
1390            for(j=1;j<=NDIV;j++)
1391              {
1392        i_b=0;
1393              for(i=1;i<=KDIV;i++)
1394       {
1395               if(rho_p[i][j]>=rho_p[1][cc_ra]) i_b=i;
1396      }
1397              last_mu_cc[j]=i_b;
1398      }
1399
1400       if (counter==1) {
1401       for(i=1;i<=KDIV;i++)
1402       {
1403          last_r_cc[i]=cc_ra;
1404       }
1405
1406      for(j=1;j<=NDIV;j++)
1407       {
1408        if(j<=cc_ra) last_mu_cc[j]=KDIV;
1409        else last_mu_cc[j]=0;
```

```
      }

        for(i=1;i<=KDIV;i++)
      {
          last_r_p[i]=n_ra;
      }

    for(j=1;j<=NDIV;j++)
      {
        if(j<=n_ra) last_mu_p[j]=KDIV;
        else last_mu_p[j]=0;
      }
      }


}

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

//********************* CC boundary for B with polynomial approxiamation
void compute_Bcc(void)
{
  int i,
      j;

  double c0,c1,c2,ucc_eq,Bcc_eq,Bcc_pol,Bcc_mid,ucc_mid;


  counter1=0;
        Bcc_eq = B[1][last_r_cc[1]];
        ucc_eq = u[1][last_r_cc[1]];

       Bcc_mid = B[(KDIV+1)/2][last_r_cc[(KDIV+1)/2]];
       ucc_mid = u[(KDIV+1)/2][last_r_cc[(KDIV+1)/2]];

       Bcc_pol = B[KDIV][last_r_cc[KDIV]];

        c0=Bcc_pol;
 c1=(Bcc_eq-c0)/ucc_eq;
 c2=(Bcc_mid-c0-c1*ucc_mid)/ (ucc_mid*(ucc_mid-ucc_eq ) );

```

```
1451
1452          for(i=1;i<=KDIV;i++)
1453           {
1454       if(counter==1){
1455                  Bcc[i] = 0.0;
1456           dB_du_cc[i] = 0.0;
1457           }
1458        else {
1459                  Bcc[i] =
                          c0+c1*u[i][last_r_cc[i]]+c2*u[i][last_r_cc[i]]*(u[i][last_r_cc[i]]-ucc_eq);
1460           dB_du_cc[i] = c1+c2*(2.0*u[i][last_r_cc[i]]-ucc_eq );
1461        }
1462
1463           if((isnan(Bcc[i])==1)||(isinf(Bcc[i])==1)) counter1++;
1464           }
1465       printf("Bcc    %d\n",counter1);
1466     }
1467     //********************************************************************** Calculate
          Grad of u
1468     void compute_grad_u(void)
1469     {
1470        int  i,
1471             j;
1472
1473           counter1=0;
1474         for(i=1;i<=KDIV;i++)
1475           {
1476           for(j=1;j<=NDIV;j++)
1477              {
1478             grad_u_r[i][j]=    deriv_r(u,i,j);
1479             grad_u_theta[i][j]=  -pow(1.0-mu[i]*mu[i],0.5)*deriv_mu(u,i,j)/r[j];
1480
1481
1482        }
1483           }
1484
1485           fix_grid(grad_u_r);
1486           fix_grid(grad_u_theta);
1487
1488        //     counter1=0;
1489           // Grad_u_r extrapolation
```

```
1490
1491
1492            for(i=1;i<=KDIV;i++){
1493
1494                    grad_u_r[i][1]=interpolate(grad_u_r,i,1);
1495                    grad_u_r[i][NDIV]=interpolate(grad_u_r,i,NDIV);
1496
1497                                    }
1498
1499            for(j=1;j<=NDIV;j++){
1500
1501                    grad_u_r[KDIV][j]=interpolate(grad_u_r,KDIV,j);
1502                    grad_u_r[1][j]=interpolate(grad_u_r,1,j);
1503                                    }
1504
1505            for(i=1;i<=KDIV;i++){
1506           for(j=1;j<=NDIV;j++){
1507            if( (isnan(grad_u_r[i][j])==1)||(isinf(grad_u_r[i][j])==1)  ) {
1508
1509               if(
                   (isnan(interpolate(grad_u_r,i,j))==0)&&(isinf(interpolate(grad_u_r,i,j))==0)
                   )
1510                  grad_u_r[i][j]=interpolate(grad_u_r,i,j);
1511
1512            else {
1513        grad_u_r[i][j]=0.0;
1514        counter1++;
1515        }
1516           }
1517            }
1518          }
1519
1520
1521           // Grad_u_theta extrapolation
1522
1523            for(i=1;i<=KDIV;i++){
1524
1525                    grad_u_theta[i][1]=interpolate(grad_u_theta,i,1);
1526                    grad_u_theta[i][NDIV]=interpolate(grad_u_theta,i,NDIV);
1527                                    }
1528
```

```
1529            for(j=1;j<=NDIV;j++){
1530
1531                    grad_u_theta[KDIV][j]=interpolate(grad_u_theta,KDIV,j);
1532                    grad_u_theta[1][j]=interpolate(grad_u_theta,1,j);
1533                              }
1534
1535            for(i=1;i<=KDIV;i++){
1536           for(j=1;j<=NDIV;j++){
1537
1538
1539
1540            if( (isnan(grad_u_theta[i][j])==1)||(isinf(grad_u_theta[i][j])==1) ) {
1541
1542           if(
                   (isnan(interpolate(grad_u_theta,i,j))==0)&&(isinf(interpolate(grad_u_theta,i,j))==0)
                   )
1543        grad_u_r[i][j]=interpolate(grad_u_theta,i,j);
1544
1545             else {
1546        grad_u_theta[i][j]=0.0;
1547                              }
1548          }
1549           }
1550         }
1551
1552
1553
1554        for(i=1;i<=KDIV;i++)
1555           {
1556           for(j=1;j<=NDIV;j++)
1557              {
1558        grad_u_norm[i][j]=  pow( ( pow(grad_u_r[i][j],2.0) + pow(grad_u_theta[i][j],2.0)
               ),0.5);
1559       }
1560           }
1561
1562           printf("grad u  fixed %d  \n",counter1);
1563
1564    }
1565
```

```
1566  //************************************************************************** Calculate
          B field norm
1567  void compute_B_field(void)
1568  {
1569  int     i,
1570          j;
1571
1572
1573    counter1=0;
1574        for(i=1;i<=KDIV;i++)
1575         {
1576         for(j=1;j<=NDIV;j++)
1577            {
1578         if((j<=last_r_cc[i])&&(i<=last_mu_cc[j])){
1579
1580            B[i][j]= rho_p[i][j] * Pi_fun[i][j];
1581
1582         }
1583         else if(chem_p[i][j]>=0)  { // If the point is inside crust use the formula with
              Bphi
1584            B[i][j]=(1.0/varpi[i][j])*pow(grad_u_norm[i][j]*grad_u_norm[i][j]
1585                        +mag_f_N[i][j]*mag_f_N[i][j] ,0.5);
1586         }
1587         else {  B[i][j]=(1.0/varpi[i][j])*grad_u_norm[i][j];
1588
1589         }
1590     }
1591         }
1592
1593         fix_grid(B);
1594
1595       //Extrapolation
1596
1597        for(i=1;i<=KDIV;i++){
1598
1599                    B[i][1]=interpolate(B,i,1);
1600                    B[i][NDIV]=interpolate(B,i,NDIV);
1601                                }
1602
1603          for(j=1;j<=NDIV;j++){
1604
```

```
1605                    B[KDIV][j]=interpolate(B,KDIV,j);
1606                    B[1][j]=interpolate(B,1,j);
1607                                }
1608
1609            for(i=1;i<=KDIV;i++){
1610          for(j=1;j<=NDIV;j++){
1611
1612
1613           if( (isnan(B[i][j])==1)||(isinf(B[i][j])==1)  ) {
1614
1615                 if( (isnan(interpolate(B,i,j))==0)&&(isinf(interpolate(B,i,j))==0) )
1616            B[i][j]=interpolate(B,i,j);
1617
1618           else {  B[i][j]=0.0;
1619       counter1++;
1620
1621           }
1622         }
1623          }
1624        }
1625
1626    printf("B  %d \n",counter1);
1627
1628     }
1629
1630
1631   //************************************************************************ Calculate B
          field components
1632   void compute_B_field_components(void)
1633   {
1634
1635    int i,
1636        j;
1637
1638       for(i=1;i<=KDIV;i++)
1639          {
1640            for(j=1;j<=NDIV;j++)
1641            {
1642            Br[i][j]= - (1.0/ (r[j]*r[j]) )*deriv_mu(u,i,j);
1643            Btheta[i][j]= -(1.0/varpi[i][j] )*deriv_r(u,i,j);
1644
```

```
1645      if((j<=last_r_cc[i])&&(i<=last_mu_cc[j]))
              Bphi[i][j]=(mag_f[i][j]*B[i][j])/(varpi[i][j]*rho_p[i][j]);
1646
1647      else if(chem_p[i][j]>=0.0) Bphi[i][j]=mag_f_N[i][j]/varpi[i][j];
1648
1649      else Bphi[i][j]=0.0;
1650            }
1651          }
1652
1653          fix_grid(Br);
1654          fix_grid(Btheta);
1655          fix_grid(Bphi);
1656
1657
1658            /* Br extrapolation */
1659
1660      for(i=1;i<=KDIV;i++){
1661
1662                    Br[i][1]=interpolate(Br,i,1);
1663                    Br[i][NDIV]=interpolate(Br,i,NDIV);
1664                                  }
1665
1666           for(j=1;j<=NDIV;j++){
1667
1668                    Br[KDIV][j]=interpolate(Br,KDIV,j);
1669                    Br[1][j]=interpolate(Br,1,j);
1670                               }
1671
1672           for(i=1;i<=KDIV;i++){
1673          for(j=1;j<=NDIV;j++){
1674
1675           if( (isnan(Br[i][j])==1)||(isinf(Br[i][j])==1) ) {
1676
1677                 if( (isnan(interpolate(Br,i,j))==0)&&(isinf(interpolate(Br,i,j))==0) )
                          Br[i][j]=interpolate(Br,i,j);
1678
1679           else Br[i][j]=0.0;
1680
1681        }
1682          }
1683        }
```

```
1684
1685
1686
1687    /* Btheta extrapolation */
1688
1689            for(i=1;i<=KDIV;i++){
1690
1691                    Btheta[i][1]=interpolate(Btheta,i,1);
1692                    Btheta[i][NDIV]=interpolate(Btheta,i,NDIV);
1693                                    }
1694
1695            for(j=1;j<=NDIV;j++){
1696
1697                    Btheta[KDIV][j]=interpolate(Btheta,KDIV,j);
1698                    Btheta[1][j]=interpolate(Btheta,1,j);
1699                                    }
1700
1701            for(i=1;i<=KDIV;i++){
1702           for(j=1;j<=NDIV;j++){
1703
1704            if( (isnan(Btheta[i][j])==1)||(isinf(Btheta[i][j])==1)  ) {
1705
1706                  if(
                        (isnan(interpolate(Btheta,i,j))==0)&&(isinf(interpolate(Btheta,i,j))==0)
                        ) B[i][j]=interpolate(Btheta,i,j);
1707
1708            else Btheta[i][j]=0.0;
1709          }
1710           }
1711         }
1712
1713
1714    /* Bphi extrapolation */
1715
1716
1717           for(i=1;i<=KDIV;i++){
1718
1719                    Bphi[i][1]=interpolate(Bphi,i,1);
1720                    Bphi[i][NDIV]=interpolate(Bphi,i,NDIV);
1721                                    }
1722
```

```
1723            for(j=1;j<=NDIV;j++){
1724
1725                    Bphi[KDIV][j]=interpolate(Bphi,KDIV,j);
1726                    Bphi[1][j]=interpolate(Bphi,1,j);
1727                                }
1728
1729            for(i=1;i<=KDIV;i++){
1730           for(j=1;j<=NDIV;j++){
1731
1732            if( (isnan(Bphi[i][j])==1)||(isinf(Bphi[i][j])==1) ) {
1733
1734                  if( (isnan(interpolate(Bphi,i,j))==0)&&(isinf(interpolate(Bphi,i,j))==0)
                        ) Bphi[i][j]=interpolate(Bphi,i,j);
1735
1736            else Bphi[i][j]=0.0;
1737          }
1738           }
1739         }
1740
1741
1742    for(j=1;j<=NDIV;j++)
1743            {
1744            for(i=1;i<=KDIV;i++)
1745              {
1746              B_pol_norm[i][j]= pow( (Br[i][j]*Br[i][j]+Btheta[i][j]*Btheta[i][j]) ,0.5);
1747              B_tor_norm[i][j]=fabs(Bphi[i][j]);
1748              }
1749            }
1750    }
1751
1752    //********************************************************************** Calculate Pi
        fun
1753    void  compute_Pi_fun(void)
1754    {
1755      int  i,
1756           j;
1757
1758
1759           counter1=0;
1760           for(i=1;i<=KDIV;i++)
1761            {
```

```
1762          for(j=1;j<=NDIV;j++)
1763             {
1764        if((j<=last_r_cc[i])&&(i<=last_mu_cc[j]))
1765
1766          Pi_fun[i][j]=grad_u_norm[i][j]/pow(varpi[i][j]*varpi[i][j]*rho_p[i][j]*rho_p[i][j]
1767                      -mag_f[i][j]*mag_f[i][j],0.5);
1768
1769        else Pi_fun[i][j]=0.0;
1770             }
1771          }
1772
1773          fix_grid(Pi_fun);
1774
1775        // Extrapolation
1776          for(i=1;i<=KDIV;i++){
1777
1778                    Pi_fun[i][1]=interpolate(Pi_fun,i,1);
1779                    Pi_fun[i][NDIV]=interpolate(Pi_fun,i,NDIV);
1780                                  }
1781
1782          for(j=1;j<=NDIV;j++){
1783
1784                    Pi_fun[KDIV][j]=interpolate(Pi_fun,KDIV,j);
1785                    Pi_fun[1][j]=interpolate(Pi_fun,1,j);
1786                               }
1787
1788          for(i=1;i<=KDIV;i++){
1789          for(j=1;j<=NDIV;j++){
1790
1791
1792      if( (isnan(Pi_fun[i][j])==1)||(isinf(Pi_fun[i][j])==1)  ) {
1793
1794                  if(
                        (isnan(interpolate(Pi_fun,i,j))==0)&&(isinf(interpolate(Pi_fun,i,j))==0)
                        )
1795                Pi_fun[i][j]=interpolate(Pi_fun,i,j);
1796
1797           else {
1798        Pi_fun[i][j]=0.0;
1799           counter1++;
1800
```

```
1801            }
1802          }
1803           }
1804         }
1805        printf("Pi_fun %d  \n",counter1);
1806       //    counter1=0;
1807    }
1808
1809    //*************************************************************** Calculate Grad
           od PI_fun
1810    void compute_grad_PI(void)
1811    {
1812        int   i,
1813              j;
1814
1815
1816         counter1=0;
1817        for(i=1;i<=KDIV;i++)
1818           {
1819           for(j=1;j<=NDIV;j++)
1820              {
1821                     grad_Pi_r[i][j]=    deriv_r(Pi_fun,i,j);
1822                grad_Pi_theta[i][j]=    -pow(1.0-mu[i]*mu[i],0.5)*deriv_mu(Pi_fun,i,j)/r[j];
1823       }
1824           }
1825
1826          fix_grid(grad_Pi_r);
1827          fix_grid(grad_Pi_theta);
1828
1829          // Grad_Pi_r extrapolation
1830
1831
1832          for(i=1;i<=KDIV;i++){
1833
1834                    grad_Pi_r[i][1]=interpolate(grad_Pi_r,i,1);
1835                    grad_Pi_r[i][NDIV]=interpolate(grad_Pi_r,i,NDIV);
1836                                }
1837
1838             for(j=1;j<=NDIV;j++){
1839
1840                    grad_Pi_r[KDIV][j]=interpolate(grad_Pi_r,KDIV,j);
```

```
1841                    grad_Pi_r[1][j]=interpolate(grad_Pi_r,1,j);
1842                                }
1843
1844          for(i=1;i<=KDIV;i++){
1845         for(j=1;j<=NDIV;j++){
1846
1847
1848          if( (isnan(grad_Pi_r[i][j])==1)||(isinf(grad_Pi_r[i][j])==1)  ) {
1849
1850             if(
                  (isnan(interpolate(grad_Pi_r,i,j))==0)&&(isinf(interpolate(grad_Pi_r,i,j))==0)
                  )
1851               grad_Pi_r[i][j]=interpolate(grad_Pi_r,i,j);
1852
1853          else {grad_Pi_r[i][j]=0.0;
1854          printf("grad_Pi_r  %d %d\n",i,j);
1855          }
1856        }
1857         }
1858       }
1859
1860          // Grad_Pi_theta extrapolation
1861
1862        for(i=1;i<=KDIV;i++){
1863
1864                  grad_Pi_theta[i][1]=interpolate(grad_Pi_theta,i,1);
1865                  grad_Pi_theta[i][NDIV]=interpolate(grad_Pi_theta,i,NDIV);
1866                              }
1867
1868          for(j=1;j<=NDIV;j++){
1869
1870                 grad_Pi_theta[KDIV][j]=interpolate(grad_Pi_theta,KDIV,j);
1871                  grad_Pi_theta[1][j]=interpolate(grad_Pi_theta,1,j);
1872                              }
1873
1874          for(i=1;i<=KDIV;i++){
1875         for(j=1;j<=NDIV;j++){
1876
1877      if( (isnan(grad_Pi_theta[i][j])==1)||(isinf(grad_Pi_theta[i][j])==1)  ) {
1878
```

```
1879        if(
                (isnan(interpolate(grad_Pi_theta,i,j))==0)&&(isinf(interpolate(grad_Pi_theta,i,j))==0)
                )
1880                    grad_Pi_theta[i][j]=interpolate(grad_Pi_theta,i,j);
1881
1882                  else {grad_Pi_theta[i][j]=0.0;
1883                  }
1884                }
1885
1886                 }
1887               }
1888
1889                printf("Grad_Pi_r  %d  \n",counter1);
1890
1891    }
1892
1893    //********************************************************************** Compute the
           magnetic functions
1894    void compute_magnetic_functions(void)
1895    {
1896
1897      int i,
1898          j;
1899      double max1,test1;
1900
1901        max1= u[1][n_ra]; // Superconductivity
1902        counter1=0;
1903     for(i=1;i<=KDIV;i++)
1904           {
1905            for(j=1;j<=NDIV;j++)
1906             {
1907
1908
1909               //Normal MHD
1910                   mag_f_N[i][j] = (alpha_c )*pow(
                           u[i][j]-max1,zeta+1.0)*unit_step(u[i][j],max1);
1911                       M_N[i][j] = k0*u[i][j];
1912               //Derivatives
1913                   df_N_du[i][j] = alpha_c* (zeta+1.0) * pow( u[i][j]-max1
                           ,zeta)*unit_step(u[i][j],max1) ;
1914                   dM_N_du[i][j] = k0;
```

```
1915
1916            // SC MHD
1917
1918                 mag_f[i][j] = rho_p[i][last_r_cc[i]]*mag_f_N[i][j]/Bcc[i];
1919            y[i][j] = Bcc[i]+ (4.0*PI/h_c) *
                    (rho_p[i][last_r_cc[i]]/rho_p[i][last_r_cc[i]+1]) * M_N[i][j];
1920
1921      M_SC[i][j]= ( h_c/(4.0*PI) ) *( y[i][j] - rho_p[i][j]*Pi_fun[i][j] );
1922
1923            //Derivatives
1924                 df_du[i][j] =
                        rho_p[i][last_r_cc[i]]*(df_N_du[i][j]*Bcc[i]-mag_f_N[i][j]*
                        dB_du_cc[i])/(Bcc[i]*Bcc[i]);
1925          dy_du[i][j] = dB_du_cc[i]+ (4.0*PI/h_c) *
                 (rho_p[i][last_r_cc[i]]/rho_p[i][last_r_cc[i]+1]) * dM_N_du[i][j];
1926
1927    //          Fix mag_f and df_du =0 for the first iteration
1928          if (counter==1){
1929        mag_f[i][j] = 0.0;
1930        df_du[i][j] = 0.0;
1931          }
1932
1933
1934           // Piecewise M function
1935          if((j<=last_r_cc[i])&&(i<=last_mu_cc[j])) M[i][j]=M_SC[i][j];
1936          else
1937         M[i][j]= M_N[i][j];
1938
1939
1940               test1= mag_f[i][j];
1941          if (((isnan(test1)==1)||(isinf(test1)==1))) {
1942      counter1++;
1943          }
1944
1945     }
1946               }
1947          printf("Mag functions  %d \n",counter1);
1948
1949    }
1950    /*****************************************************************/
1951    /* Find the density-like function for Aphi */
```

```
1952  /*****************************************************************/
1953  void compute_u_density(void)
1954  {
1955
1956      int i,
1957          j;
1958
1959      double Pi_max;
1960
1961
1962        if (counter==1) Pi_max=1.0;
1963        else
1964   Pi_max=fabs(max(Pi_fun));
1965  //       Pi_max=1.0;
1966     //   printf("%f \n",Pi_max);
1967
1968      for(i=1;i<=KDIV;i++)
1969         {
1970          for(j=1;j<=NDIV;j++)
1971           {
1972
1973     if(counter==1) F[i][j]=0.00001;
1974
1975     else if((j<=last_r_cc[i])&&(i<=last_mu_cc[j]))
1976
1977     // SC MHD F(u)
1978
1979                F[i][j]=( -(
                       grad_Pi_r[i][j]*grad_u_r[i][j]+grad_Pi_theta[i][j]*grad_u_theta[i][j]
                       )/ ( Pi_fun[i][j]*varpi[i][j] )+
1980                            varpi[i][j]*rho_p[i][j]*Pi_fun[i][j]*dy_du[i][j]+
1981                  Pi_fun[i][j]*Pi_fun[i][j]*mag_f[i][j]*df_du[i][j]/varpi[i][j] );
1982
1983
1984            else if(chem_p[i][j]>=0.0){
1985            // Normal MHD
1986               F[i][j]= (1.0/(r[j]*pow(1.0-mu[i]*mu[i],0.5) )
                       )*df_N_du[i][j]*mag_f_N[i][j] +
1987                            4.0*PI*dM_N_du[i][j]*r[j]*pow(1.0-mu[i]*mu[i],0.5)*rho_p[i][j];
1988
1989                            }
```

```
1990
1991        else   F[i][j]=0.0;
1992
1993       F[i][j]=F[i][j]/Pi_max;   //divide by Pi_fun_max as instructed
1994
1995      }
1996           }
1997             //Extrapolation
1998       counter1=0;
1999
2000
2001          fix_grid(F);
2002
2003          for(i=1;i<=KDIV;i++){
2004
2005                       F[i][1]=interpolate(F,i,1);
2006                       F[i][NDIV]=interpolate(F,i,NDIV);
2007      F[i][last_r_cc[i]]=interpolate(F,i,last_r_cc[i]);
2008                                    }
2009
2010           for(j=1;j<=NDIV;j++){
2011
2012                    F[KDIV][j]=interpolate(F,KDIV,j);
2013                     F[1][j]=interpolate(F,1,j);
2014                                }
2015
2016           for(i=1;i<=KDIV;i++){
2017          for(j=1;j<=NDIV;j++){
2018
2019           if( (isnan(F[i][j])==1)||(isinf(F[i][j])==1)   ) {
2020
2021                  if( (isnan(interpolate(F,i,j))==0)&&(isinf(interpolate(F,i,j))==0) )
                          F[i][j]=interpolate(F,i,j);
2022
2023           else {F[i][j]=0.0;
2024       counter1++;
2025            }
2026         }
2027           }
2028         }
2029
```

```
2030        printf("F  %d \n",counter1);
2031     }
2032
2033
2034     //******************************************** Compute gravitational potential
2035     void compute_PHI(void){
2036
2037
2038        int i,
2039            j,
2040            n,
2041            k;
2042
2043
2044        double d1[NDIV+1][LMAX+1],   /* function D^(1)_{k,n} */
2045               d2[LMAX+1][NDIV+1],   /* function D^(2)_{n,j} */
2046               s=0.0,                /* term in sum */
2047               sum=0.0;              /* intermediate sum */
2048
2049
2050         /* Gravitational Potential */
2051
2052         for(k=1;k<=NDIV;k++)
2053          {
2054           for(n=0;n<=LMAX;n++)
2055            {
2056             for(i=1;i<=KDIV-2;i+=2)
2057              {
2058               s= (1.0/(3.0*(KDIV-1.0)))*(p2n[n][i]*rho[i][k]
2059                     + 4.0*p2n[n][i+1]*rho[i+1][k]
2060                     + p2n[n][i+2]*rho[i+2][k]);
2061               sum+=s;
2062              }
2063               d1[k][n]=sum;
2064               sum=0.0;
2065     }
2066          }
2067
2068         sum=0.0;
2069         for(j=1;j<=NDIV;j++)
2070          {
```

```
2071            for(n=0;n<=LMAX;n++)
2072             {
2073              for(k=1;k<=NDIV-2;k+=2)
2074               {
2075                 s=RMAX/(3.0*(NDIV-1))*( f2n[n][k][j]*d1[k][n]
2076                                 + 4.0*f2n[n][k+1][j]*d1[k+1][n]
2077                                 + f2n[n][k+2][j]*d1[k+2][n]);
2078                 sum+=s;
2079       }
2080        d2[n][j]=sum;
2081                 sum=0.0;
2082     }
2083          }
2084
2085           sum=0.0;
2086           for(i=1;i<=KDIV;i++)
2087            {
2088             for(j=1;j<=NDIV;j++)
2089              {
2090               for(n=0;n<=LMAX;n++)
2091                {
2092                 s= -4.0*PI*d2[n][j]*p2n[n][i];
2093
2094                 sum+=s;
2095                }
2096                 phi[i][j]=sum;
2097                 sum=0.0;
2098              }
2099            }
2100
2101          for(i=1;i<=KDIV;i++) phi[i][1]=phi[i][2]; /* Correct sing. at r=0. */
2102                                                    /* Introduced error is   */
2103                                                    /* negligible.           */
2104    }
2105
2106
2107
2108    //*************************************************************** Compute u from
          poisson equation
2109    void compute_u_pot(void){
2110
```

```
2111        int   i,
2112              j,
2113              n,
2114              k;
2115
2116       double s=0.0,                  /* term in sum */
2117              sum=0.0,                /* intermediate sum */
2118       Pi_max,
2119
2120       du1[NDIV+1][LMAX+1], // similar to d1 and d1 functions (see above) for integrating u
2121       du2[LMAX+1][NDIV+1];
2122
2123
2124      //   if (max(Pi_fun)>1.0)  Pi_max=max(Pi_fun);
2125             //    else Pi_max=10.0;
2126
2127       /* compute u_new* from u (u_old) integrating poisson equation  */
2128             if (counter==1) Pi_max=1.0;
2129      else
2130      Pi_max=fabs(max(Pi_fun));
2131      //Pi_max=1.0;
2132          sum=0.0;
2133          for(k=1;k<=NDIV;k++)
2134           {
2135            for(n=1;n<=LMAX;n++)
2136            {
2137             for(i=1;i<=KDIV-2;i+=2)
2138              {
2139       s= (1.0/(3.0*(KDIV-1.0)))*
2140                        (p1_2n_1[n][i]*F[i][k]
2141                       + 4.0*p1_2n_1[n][i+1]*F[i+1][k]
2142                       + p1_2n_1[n][i+2]*F[i+2][k] );
2143                sum+=s;
2144               }
2145                du1[k][n]=sum;
2146                sum=0.0;
2147      }
2148           }
2149
2150          sum=0.0;
2151          for(j=1;j<=NDIV;j++)
```

```
2152          {
2153           for(n=1;n<=LMAX;n++)
2154            {
2155             for(k=1;k<=NDIV-2;k+=2)
2156              {
2157                s=RMAX/(3.0*(NDIV-1))*( f2n_1[n][k][j]*du1[k][n]
2158                                      + 4.0*f2n_1[n][k+1][j]*du1[k+1][n]
2159                                      + f2n_1[n][k+2][j]*du1[k+2][n] );
2160                sum+=s;
2161      }
2162       du2[n][j]=sum;
2163                sum=0.0;
2164    }
2165         }
2166
2167
2168          sum=0.0;
2169          for(i=1;i<=KDIV;i++)
2170           {
2171            for(j=1;j<=NDIV;j++)
2172             {
2173              for(n=1;n<=LMAX;n++)
2174               {
2175                s= 4.0*PI*du2[n][j]*p1_2n_1[n][i]*(1.0/(2.0*n*(2.0*n-1.0)));
2176
2177                sum+=s;
2178               }
2179               u_new_star[i][j]=( 1.0/(4.0*PI) )*varpi[i][j]*sum*Pi_max;  // Myltiply the
                       result with Pi_fun_max
2180            //       u[i][j]=( 1.0/(4.0*PI) )*sum*Pi_max;
2181               sum=0.0;
2182             }
2183           }
2184
2185
2186
2187
2188
2189      for(i=1;i<=KDIV;i++) u_new_star[i][1]=u_new_star[i][2]; /* Correct sing. at r=0. */
2190                                                              /* Introduced error is   */
2191                                                              /* negligible.           */
```

```
2192
2193    }
2194
2195
2196    //*********************************************************** Find the toroidal
            magnetic energy
2197    void compute_toroidal_magnetic_energy(void)
2198    {
2199
2200     int i,
2201         j;
2202
2203     double s=0.0,
2204            sum=0.0,
2205     //          Emagtor1=0.0,
2206     Emtor[NDIV+1];
2207
2208     for(j=1;j<=NDIV;j++)
2209            {
2210             for(i=1;i<=KDIV-2;i+=2)
2211              {
2212               s=((mu[i+1]-mu[i])/3.0)*(pow(B_tor_norm[i][j],2.0)+
                       4.0*pow(B_tor_norm[i+1][j],2.0)+ pow(B_tor_norm[i+2][j],2.0) );
2213               sum+=s;
2214              }
2215               Emtor[j]=sum;
2216               sum=0.0;
2217            }
2218
2219           for(j=1;j<=NDIV-2;j+=2)
2220            {
2221             s=(4.0/3.0)*PI*(r[j+1]-r[j])*(r[j]*r[j]*Emtor[j]
2222                +4.0*r[j+1]*r[j+1]*Emtor[j+1]
2223                                                +r[j+2]*r[j+2]*Emtor[j+2]);
2224             Emagtor+=s;
2225            }
2226            Emagtor=Emagtor/(8.0*PI);
2227
2228
2229    }
2230
```

```
2231
2232
2233  //*************************************************************** Employ
          underrelaxation
2234  void underrelaxation(void){
2235
2236    int i,
2237        j;
2238
2239    for(i=1;i<=KDIV;i++)
2240          {
2241           for(j=1;j<=NDIV;j++)
2242            {
2243      u[i][j]=(1.0-undrlx_c)*u[i][j]+undrlx_c*u_new_star[i][j];
2244    }
2245          }
2246     for(i=1;i<=KDIV;i++) u[i][1]=u[i][2];
2247  }
2248
2249
2250  void u_converge()
2251  {
2252
2253    int i,
2254        j;
2255   double deltau[KDIV+1][NDIV+1],
2256          deltau_cc[KDIV+1],
2257   deltaQ[KDIV+1][NDIV+1];
2258
2259      conv=0.0;
2260      conv_cc=0.0;
2261      convQ=0.0;
2262      convFlux=0.0;
2263        for(i=1;i<=KDIV;i++)
2264              {
2265               for(j=1;j<=NDIV;j++)
2266                {
2267    deltau[i][j]= (u[i][j]-check_u[i][j])*(u[i][j]-check_u[i][j]);
2268   // deltau[i][j]= fabs( (u[i][j]-check_u[i][j])/u[i][j] );
2269    conv=conv+deltau[i][j];
2270
```

```
2271      deltaQ[i][j]=(chem_p[i][j]-checkQ[i][j])*(chem_p[i][j]-checkQ[i][j]);
2272      convQ=convQ+deltaQ[i][j];
2273            }
2274         deltau_cc[i]= (u[i][last_r_cc[i]+1]-check_u[i][last_r_cc[i]+1])
2275                       *(u[i][last_r_cc[i]+1]-check_u[i][last_r_cc[i]+1]);
2276         conv_cc=conv_cc+deltau_cc[i];
2277         }
2278
2279          conv=conv/(1.0*NDIV*KDIV);
2280          conv=pow(conv,0.5);
2281
2282          conv_cc=conv_cc/(1.0*KDIV);
2283          conv_cc=pow(conv_cc,0.5);
2284
2285          convQ=convQ/(1.0*NDIV*KDIV);
2286          convQ=pow(convQ,0.5);
2287
2288          convFlux=(BFlux_surf-BFlux_surf_check)*(BFlux_surf-BFlux_surf_check);
2289          convFlux=pow(convFlux,0.5);
2290
2291          BFlux_surf_check=BFlux_surf;
2292
2293             for(i=1;i<=KDIV;i++)
2294                 {
2295                   for(j=1;j<=NDIV;j++)
2296                    {
2297      check_u[i][j]=u[i][j];
2298      checkQ[i][j]=chem_p[i][j];
2299
2300            }
2301         }
2302
2303       fprintf(fconv," %2.16f    \n",conv);
2304       fprintf(fconvcc," %2.16f    \n",conv_cc);
2305       fprintf(fconvQ," %2.16f    \n",convQ);
2306       fprintf(fFlux," %2.16f   \n",convFlux);
2307    }
2308
2309    void div_B(void) {
2310
2311       int i,
```

```
2312            j;
2313
2314        double       dudmu[KDIV+1][NDIV+1],
2315                       dudr[KDIV+1][NDIV+1],
2316                 d2udmudr[KDIV+1][NDIV+1],
2317                 d2udrdmu[KDIV+1][NDIV+1],
2318         dBcharge[NDIV+1],
2319         sum=0.0,
2320         s=0.0;
2321
2322         magcharge=0.0;
2323
2324         for(j=1;j<=NDIV;j++)
2325              {
2326               for(i=1;i<=KDIV;i++)
2327                {
2328                    dudmu[i][j]=deriv_mu(u,i,j);
2329                     dudr[i][j]=deriv_r(u,i,j);
2330        }
2331             }
2332
2333         for(j=1;j<=NDIV;j++)
2334              {
2335               for(i=1;i<=KDIV;i++)
2336                {
2337                    d2udmudr[i][j]=deriv_r(dudmu,i,j);
2338                    d2udrdmu[i][j]=deriv_mu(dudr,i,j);
2339
2340           divB[i][j]= d2udrdmu[i][j]-d2udmudr[i][j];
2341        }
2342             }
2343
2344
2345        // Total numerical magnetic charge
2346
2347         sum=0.0;
2348           for(j=1;j<=NDIV;j++)
2349              {
2350               for(i=1;i<=KDIV-2;i+=2)
2351                {
2352                 s=((mu[i+1]-mu[i])/3.0)*(divB[i][j]+4*divB[i+1][j]+divB[i+2][j]);
```

```
                sum+=s;
               }
                dBcharge[j]=sum;
                sum=0.0;
            }

          for(j=1;j<=NDIV-2;j+=2)
            {
              s=(4.0/3.0)*PI*(r[j+1]-r[j])*(dBcharge[j]
                 +4.0*dBcharge[j+1]
                                                   +dBcharge[j+2]);
              magcharge+=s;
            }
          magcharge= magcharge/(4.0*PI);      // divB= 4 PI rho charge magnetic

          fprintf(fmagcharge,"%2.15f   \n",magcharge);
}


//************************************************************** calculate the Phi =
    \int \int B dot dS
// the flux on the half hemisphere in order to compare with div B.
void compute_B_flux(void) {

   int i,
       j;

   double s=0.0;

   BFlux_surf=0.0;

   for(i=1;i<=KDIV-2;i+=2)
        {
         s= (2.0/3.0)*PI*(mu[i+1]-mu[i])*(Br[i][n_ra]+4.0*Br[i+1][n_ra]+Br[i+2][n_ra]);
         BFlux_surf=BFlux_surf+s;
        }

}

//&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
```

```
2392  //*********************************************************************** Main iteration
          function
2393  void iterate (int n_rb, double Np_index, double Nn_index) {
2394
2395      int i,
2396          j,
2397          n,
2398          k,
2399          iter;
2400
2401    double chem_n_max_old,        // chemical potential values in previous cycle
2402    chem_p_max_old,
2403           omega_02_old=0.0,     /* omega_0^2 in previous cycle */
2404           C_p_old=0.0,          // integration constants in previous cycle
2405    C_dif_old=0.0,
2406
2407    test11,test12,test13,
2408
2409    dif1=1.0,             /* | chem_n_max_old - chem_n_max | */
2410           dif2=1.0,             /* |  Not Checkedchem_p_max_old - chem_p_max | */
2411           dif3=1.0,             /* | omega_02_old-  omega_02     | */
2412           dif4=1.0,             /* | C_p_old -    C_p | */
2413           dif5=1.0;             /* | C_dif_old - C_dif |  */
2414
2415    while( (dif1>eps) || (dif2>eps) || (dif3>eps) || (dif4>eps) || (dif5>eps ) )
2416
2417     {
2418          counter++;
2419
2420          printf(" %d    %2.9f    %2.9f    %2.9f     %2.9f     %2.9f
                 \n",counter,dif1,dif2,dif3,dif4,dif5);
2421
2422   compute_last_points();
2423   // Find Gravitational Potential
2424   total_density();
2425   compute_PHI();
2426
2427   // Find new u.
2428
2429   compute_grad_u();
2430    compute_Pi_fun();
```

```
2431    compute_B_field();
2432          compute_grad_PI();
2433    compute_Bcc();
2434    compute_magnetic_functions();
2435
2436    compute_u_density();
2437    compute_u_pot();
2438
2439    underrelaxation();
2440    compute_B_field_components();
2441     compute_B_flux();
2442    // Evaluate integration constants ////////////////////////
2443
2444        //  Protons
2445
2446         omega_02 = 2.0*(phi[1][n_ra]-M[1][n_ra]+M[KDIV][n_rb]-phi[KDIV][n_rb]);
2447             C_p = phi[1][n_ra]-M[1][n_ra]-omega_02/2.0;
2448
2449    //           omega_02 = 2.0*(phi[1][n_ra]-phi[KDIV][n_rb]);
2450     //         C_p = phi[1][n_ra]-omega_02/2.0;    // magnetic field and matter do not
            interact
2451
2452        // Chemical potential for protons
2453         for(i=1;i<=KDIV;i++)
2454                   {
2455                   for(j=1;j<=NDIV;j++)
2456               {
2457                 chem_p[i][j]=-phi[i][j]+varpi[i][j]*varpi[i][j]*omega_02/2.0 + M[i][j]+C_p;
2458      //         chem_p[i][j]=-phi[i][j]+varpi[i][j]*varpi[i][j]*omega_02/2.0 +C_p;
2459                                                    // magnetic field and matter do not
                                                           interact
2460        }
2461            }
2462
2463         // Neutrons
2464
2465             C_dif = chem_p[1][cc_ra]-M[1][cc_ra];
2466    // C_dif = chem_p[1][cc_ra]; // magnetic field and matter do not interact
2467
2468        // Chemical potential for neutrons
2469
```

```
2470          for(i=1;i<=KDIV;i++)
2471                  {
2472                  for(j=1;j<=NDIV;j++)
2473              {
2474          chem_n[i][j]=chem_p[i][j] - M[i][j] - C_dif ;
2475      //      chem_n[i][j]=chem_p[i][j] - C_dif ;    // magnetic field and matter do not
              interact
2476              }
2477            }
2478
2479            ////////////////////////////////////////////////////////////////
2480
2481              // New densities
2482
2483            chem_p_max = max(chem_p);
2484      chem_n_max = max(chem_n);
2485
2486      // Protons
2487       for(i=1;i<=KDIV;i++)
2488                {
2489                 for(j=1;j<=NDIV;j++)
2490                  {
2491                   if(chem_p[i][j]>=0)
2492                     {
2493                      if(Np_index==0.0) rho_p[i][j]=1.0;
2494                      else
2495                          rho_p[i][j]= x_p_0*pow( (chem_p[i][j]/chem_p_max) ,Np_index );
2496             }
2497                   else rho_p[i][j]=0.0;
2498          }
2499                }
2500
2501              // Neutrons
2502                for(i=1;i<=KDIV;i++)
2503                 {
2504                  for(j=1;j<=NDIV;j++)
2505                   {
2506                    if(chem_n[i][j]>=0)
2507                      {
2508                       if(Nn_index==0.0) rho_n[i][j]=1.0;
2509                       else
```

```
2510                          rho_n[i][j]= (1.0-x_p_0)*pow( (chem_n[i][j]/chem_n_max) ,Nn_index
                                  );
2511              }
2512                   else rho_n[i][j]=0.0;
2513           }
2514                }
2515
2516           ////////////////////////////////////////////////////////////////
2517
2518          dif1=fabs( chem_n_max_old - chem_n_max);
2519          dif2=fabs( chem_p_max_old - chem_p_max);
2520          dif3=fabs( omega_02_old - omega_02 );
2521          dif4=fabs(C_p_old - C_p);
2522          dif5=fabs(C_dif_old - C_dif);
2523
2524        chem_n_max_old = chem_n_max;
2525        chem_p_max_old = chem_p_max;
2526          omega_02_old = omega_02;
2527               C_p_old = C_p;
2528             C_dif_old = C_dif;
2529
2530      u_converge();
2531
2532       // Divergence of B field  (divB)
2533
2534            div_B();
2535
2536        } // end while
2537    } // end of function
2538
2539
2540    // iteration of the magnetic field decoupled from the matter
2541    void iterate2 (int iternumb)
2542    {
2543      int iter;
2544
2545           for(iter=1;iter<=iternumb;iter++)
2546       {
2547     printf("%d ",iter);
2548     compute_grad_u();
2549      compute_Pi_fun();
```

```
2550      compute_B_field();
2551             compute_grad_PI();
2552      compute_Bcc();
2553      compute_magnetic_functions();
2554      compute_u_density();
2555      compute_u_pot();
2556      underrelaxation();
2557      compute_B_field_components();
2558       compute_B_flux();
2559      u_converge();
2560            // Divergence of B field  (divB)
2561
2562             div_B();
2563        }
2564
2565    }
2566
2567    // iteration of the magnetic field coupled with matter (same as iteration() but for a
          specific
2568    // number of iterations
2569    void iterate3(int n_rb, double Np_index, double Nn_index, int repet) {
2570
2571        int i,
2572            j,
2573            n,
2574            k,
2575            iter;
2576
2577      double chem_n_max_old,        // chemical potential values in previous cycle
2578      chem_p_max_old,
2579             omega_02_old=0.0,     /* omega_0^2 in previous cycle */
2580             C_p_old=0.0,          // integration constants in previous cycle
2581      C_dif_old=0.0,
2582
2583      test11,test12,test13,
2584
2585      dif1=1.0,              /* | chem_n_max_old - chem_n_max | */
2586             dif2=1.0,             /* |  Not Checkedchem_p_max_old - chem_p_max | */
2587             dif3=1.0,             /* | omega_02_old-  omega_02     | */
2588             dif4=1.0,             /* | C_p_old -    C_p | */
2589             dif5=1.0;             /* | C_dif_old - C_dif |  */
```

```
2590
2591          //     eps=1.0e-5;    //     accuracy
2592
2593
2594
2595         for(iter=1;iter<=repet;iter++)
2596        {
2597              counter++;
2598
2599               printf(" %d    %2.9f    %2.9f    %2.9f     %2.9f     %2.9f
                      \n",counter,dif1,dif2,dif3,dif4,dif5);
2600
2601       compute_last_points();
2602      // Find Gravitational Potential
2603      total_density();
2604      compute_PHI();
2605
2606      // Find new u.
2607
2608      compute_grad_u();
2609       compute_Pi_fun();
2610      compute_B_field();
2611              compute_grad_PI();
2612      compute_Bcc();
2613      compute_magnetic_functions();
2614
2615      compute_u_density();
2616      compute_u_pot();
2617
2618
2619      underrelaxation();
2620      compute_B_field_components();
2621       compute_B_flux();
2622
2623      // Evaluate integration constants ////////////////////////
2624
2625          //  Protons
2626
2627           omega_02 = 2.0*(phi[1][n_ra]-M[1][n_ra]+M[KDIV][n_rb]-phi[KDIV][n_rb]);
2628           omega_02= 0.0;
2629                C_p = phi[1][n_ra]-M[1][n_ra]-omega_02/2.0;
```

```
2630
2631    //           omega_02 = 2.0*(phi[1][n_ra]-phi[KDIV][n_rb]);
2632     //         C_p = phi[1][n_ra]-omega_02/2.0;    // magnetic field and matter do not
            interact
2633
2634        // Chemical potential for protons
2635         for(i=1;i<=KDIV;i++)
2636                  {
2637                  for(j=1;j<=NDIV;j++)
2638              {
2639                chem_p[i][j]=-phi[i][j]+varpi[i][j]*varpi[i][j]*omega_02/2.0 + M[i][j]+C_p;
2640       //        chem_p[i][j]=-phi[i][j]+varpi[i][j]*varpi[i][j]*omega_02/2.0 +C_p; //
             magnetic field and matter do not interact
2641        }
2642           }
2643
2644         // Neutrons
2645
2646            C_dif = chem_p[1][cc_ra]-M[1][cc_ra];
2647    // C_dif = chem_p[1][cc_ra]; // magnetic field and matter do not interact
2648
2649       // Chemical potential for neutrons
2650
2651          for(i=1;i<=KDIV;i++)
2652                  {
2653                  for(j=1;j<=NDIV;j++)
2654              {
2655          chem_n[i][j]=chem_p[i][j] - M[i][j] - C_dif ;
2656     //     chem_n[i][j]=chem_p[i][j] - C_dif ;    // magnetic field and matter do not
            interact
2657              }
2658           }
2659
2660           ////////////////////////////////////////////////////////////////
2661
2662             // New densities
2663
2664            chem_p_max = max(chem_p);
2665      chem_n_max = max(chem_n);
2666
2667      // Protons
```

```
2668        for(i=1;i<=KDIV;i++)
2669                {
2670                 for(j=1;j<=NDIV;j++)
2671                  {
2672                   if(chem_p[i][j]>=0)
2673                     {
2674                      if(Np_index==0.0) rho_p[i][j]=1.0;
2675                      else
2676                          rho_p[i][j]= x_p_0*pow( (chem_p[i][j]/chem_p_max) ,Np_index );
2677               }
2678                     else rho_p[i][j]=0.0;
2679          }
2680                }
2681
2682              // Neutrons
2683                for(i=1;i<=KDIV;i++)
2684                 {
2685                  for(j=1;j<=NDIV;j++)
2686                   {
2687                    if(chem_n[i][j]>=0)
2688                      {
2689                       if(Nn_index==0.0) rho_n[i][j]=1.0;
2690                       else
2691                           rho_n[i][j]= (1.0-x_p_0)*pow( (chem_n[i][j]/chem_n_max) ,Nn_index
                                );
2692               }
2693                     else rho_n[i][j]=0.0;
2694          }
2695                }
2696
2697        ////////////////////////////////////////////////////////////////
2698
2699
2700
2701
2702
2703          dif1=fabs( chem_n_max_old - chem_n_max);
2704          dif2=fabs( chem_p_max_old - chem_p_max);
2705          dif3=fabs( omega_02_old - omega_02 );
2706          dif4=fabs(C_p_old - C_p);
2707          dif5=fabs(C_dif_old - C_dif);
```

```
2708
2709        chem_n_max_old = chem_n_max;
2710        chem_p_max_old = chem_p_max;
2711          omega_02_old = omega_02;
2712               C_p_old = C_p;
2713             C_dif_old = C_dif;
2714
2715      u_converge();
2716
2717       // Divergence of B field  (divB)
2718
2719             div_B();
2720
2721
2722       } // end while
2723    } // end of function
2724
2725    //*************************************************************************
2726    /* Compute various quantities.                                           */
2727    /*************************************************************************/
2728    void comp()
2729    {
2730      int i,                        /* counter */
2731          j,                        /* counter */
2732          j_b;                      /* last point inside star */
2733
2734     double s=0.0,                 /* individual term in sum */
2735            sum=0.0,               /* intermediate sum */
2736
2737            dv1[NDIV+1],           /* integrated quantity in volume */
2738            dm1[NDIV+1],           /* integrated quantity in mass */
2739           dmi1[NDIV+1],          /* integrated quantity in moment of inertia */
2740            dw1[NDIV+1],           /* integrated quantity in potential energy */
2741         dp_en1[NDIV+1],      // integrated quantity for proton EOS
2742         dn_en1[NDIV+1],      // integrated quantity for neutron EOS
2743          dmag1[KDIV+1]; /* integrated quantity in Magnetic Energy */
2744
2745     /* Initialize variables */
2746
2747     m=0.0;
2748     v=0.0;
```

```
2749      mi=0.0;
2750      am=0.0;
2751      ke=0.0;
2752      w=0.0;
2753      Pi_p=0.0;
2754      Pi_n=0.0;
2755      Emag=0.0;
2756
2757      ////////////////////////
2758
2759         //Compute star surface
2760            compute_last_points();
2761            star_surface();
2762            neutron_surface();
2763            cc_surface();
2764            total_density();
2765            compute_B_field();
2766            compute_B_field_components();
2767            compute_Bcc();
2768            compute_magnetic_functions();
2769            fix_grid(B_pol_norm);
2770            fix_grid(u);
2771
2772            // compute B flux
2773
2774            compute_B_flux();
2775
2776          // CORRECT DENSITY OUTSIDE STAR      Could also correct rho_n and rho_p.
2777
2778            for(i=1;i<=KDIV;i++)
2779            {
2780              for(j=1;j<=NDIV;j++) {
2781          if(r[j]>r_bound[i]) rho[i][j]=0.0;
2782                                    }
2783            }
2784
2785        ////////////////////////////////////
2786
2787          // Volume
2788
2789           for(i=1;i<=KDIV-2;i+=2)
```

```
2790            {
2791             s= (4.0/9.0)*PI*(mu[i+1]-mu[i])*(pow(r_bound[i],3.0)
2792                          +4.0*pow(r_bound[i+1],3.0)+pow(r_bound[i+2],3.0));
2793             v+=s;
2794            }
2795
2796          ////////////////////////////////////
2797
2798          // Mass
2799          sum=0.0;
2800           for(j=1;j<=NDIV;j++)
2801            {
2802             for(i=1;i<=KDIV-2;i+=2)
2803              {
2804               s=((mu[i+1]-mu[i])/3.0)*(rho[i][j]+4*rho[i+1][j]+rho[i+2][j]);
2805               sum+=s;
2806              }
2807               dm1[j]=sum;
2808               sum=0.0;
2809            }
2810
2811           for(j=1;j<=NDIV-2;j+=2)
2812            {
2813             s=(4.0/3.0)*PI*(r[j+1]-r[j])*(r[j]*r[j]*dm1[j]
2814                +4.0*r[j+1]*r[j+1]*dm1[j+1]
2815                                          +r[j+2]*r[j+2]*dm1[j+2]);
2816             m+=s;
2817            }
2818
2819          ////////////////////////////////////
2820
2821            // Moment of inertia
2822           sum=0.0;
2823           for(j=1;j<=NDIV;j++)
2824            {
2825             for(i=1;i<=KDIV-2;i+=2)
2826              {
2827               s=((mu[i+1]-mu[i])/3.0)*( (1.0-mu[i]*mu[i])*rho[i][j]
2828                                        +4.0*(1.0-mu[i+1]*mu[i+1])*rho[i+1][j]
2829                                         +(1.0-mu[i+2]*mu[i+2])*rho[i+2][j]);
2830               sum+=s;
```

```
2831              }
2832               dmi1[j]=sum;
2833               sum=0.0;
2834             }
2835
2836
2837            for(j=1;j<=NDIV-2;j+=2)
2838             {
2839              s=(4.0/3.0)*PI*(r[j+1]-r[j])*(pow(r[j],4.0)*dmi1[j]
2840                                            +4.0*pow(r[j+1],4.0)*dmi1[j+1]
2841                                            +pow(r[j+2],4.0)*dmi1[j+2]);
2842              mi+=s;
2843             }
2844
2845           ////////////////////////////////
2846
2847            // Angular momentum
2848
2849            am=mi*pow(omega_02,0.5);
2850
2851            ////////////////////////////
2852
2853            // Kinetic energy
2854
2855            ke=0.5*mi*omega_02;
2856
2857           ////////////////////////
2858
2859              // Gravitational energy
2860            sum=0.0;
2861
2862            for(j=1;j<=NDIV;j++)
2863             {
2864              for(i=1;i<=KDIV-2;i+=2)
2865               {
2866                s=((mu[i+1]-mu[i])/3.0)*(rho[i][j]*phi[i][j]
2867                                         +4.0*rho[i+1][j]*phi[i+1][j]
2868                                         +rho[i+2][j]*phi[i+2][j]);
2869                sum+=s;
2870               }
2871                dw1[j]=sum;
```

```
2872              sum=0.0;
2873            }
2874
2875
2876          for(j=1;j<=NDIV-2;j+=2)
2877            {
2878              s=(2.0/3.0)*PI*(r[j+1]-r[j])*(r[j]*r[j]*dw1[j]
2879                                +4.0*r[j+1]*r[j+1]*dw1[j+1]
2880                                +r[j+2]*r[j+2]*dw1[j+2]);
2881              w+=s;
2882            }
2883
2884          ////////////////////////
2885
2886          // Internal energy densities (energy functional)
2887
2888          chem_p_max=max(chem_p);
2889          chem_n_max=max(chem_n);
2890
2891          for(j=1;j<=NDIV;j++)
2892            {
2893              for(i=1;i<=KDIV;i++)
2894                {
2895                  n_ener[i][j]= (chem_n_max/( 1.0+ (1.0/Nn_index) ) )*( pow(1.0-x_p_0,
                          (-1.0/Nn_index) ) )*pow(rho_n[i][j],1.0+(1.0/Nn_index) );
2896          p_ener[i][j]= (chem_p_max/( 1.0+ (1.0/Np_index) ) )*( pow(x_p_0, (-1.0/Np_index) )
              )*pow(rho_p[i][j],1.0+(1.0/Np_index) );
2897                }
2898            }
2899
2900         ////////////////////////////////////////
2901
2902          // Internal energy for protons ----------------------
2903          sum=0.0;
2904          for(j=1;j<=NDIV;j++)
2905            {
2906              for(i=1;i<=KDIV-2;i+=2)
2907                {
2908                  s=((mu[i+1]-mu[i])/3.0)*(p_ener[i][j]+4*p_ener[i+1][j]+p_ener[i+2][j]);
2909                  sum+=s;
2910                }
```

```
2911            dp_en1[j]=sum;
2912            sum=0.0;
2913          }
2914
2915
2916         for(j=1;j<=NDIV-2;j+=2)
2917          {
2918           s=(4.0/3.0)*PI*(r[j+1]-r[j])*(r[j]*r[j]*dp_en1[j]
2919                          + 4.0*r[j+1]*r[j+1]*dp_en1[j+1] + r[j+2]*r[j+2]*dp_en1[j+2]);
2920           Pi_p+=s;
2921          }
2922
2923         ////////////////////////////////////
2924
2925
2926         // Internal energy for neutrons ----------------------
2927         sum=0.0;
2928         for(j=1;j<=NDIV;j++)
2929          {
2930           for(i=1;i<=KDIV-2;i+=2)
2931            {
2932             s=((mu[i+1]-mu[i])/3.0)*(n_ener[i][j]+4*n_ener[i+1][j]+n_ener[i+2][j]);
2933             sum+=s;
2934            }
2935             dn_en1[j]=sum;
2936             sum=0.0;
2937          }
2938
2939
2940         for(j=1;j<=NDIV-2;j+=2)
2941          {
2942           s=(4.0/3.0)*PI*(r[j+1]-r[j])*(r[j]*r[j]*dn_en1[j]
2943                          + 4.0*r[j+1]*r[j+1]*dn_en1[j+1] + r[j+2]*r[j+2]*dn_en1[j+2]);
2944           Pi_n+=s;
2945          }
2946
2947         ////////////////////////////////////
2948
2949         // Magnetic Energy
2950         sum=0.0;
2951
```

```
2952
2953            for(j=1;j<=NDIV;j++)
2954            {
2955              for(i=1;i<=KDIV-2;i+=2)
2956                {
2957                 s=( (mu[i+1]-mu[i])/3.0 )* (pow((1-mu[i]*mu[i]),0.5)*rho_p[i][j]*deriv_r(M,i,j)
2958          +4.0*pow((1-mu[i+1]*mu[i+1]),0.5)*rho_p[i+1][j]*deriv_r(M,i+1,j)
2959          +pow((1-mu[i+2]*mu[i+2]),0.5)*rho_p[i+2][j]*deriv_r(M,i+2,j) );
2960                 sum+=s;
2961                }
2962                 dmag1[j]=sum;
2963                 sum=0.0;
2964            }
2965
2966
2967          for(j=1;j<=NDIV-2;j+=2)
2968            {
2969             s=(4.0/3.0)*PI*(r[j+1]-r[j])*(pow(r[j],3.0)*dmag1[j]
2970                        +4.0*pow(r[j+1],3.0)*dmag1[j+1]
2971                        +pow(r[j+2],3.0)*dmag1[j+2]);
2972             Emag+=s;
2973            }
2974
2975          compute_toroidal_magnetic_energy();
2976         ////////////////////////////////////
2977
2978
2979           /* Keplerian angular velocity */
2980
2981          omega_k2 = -deriv_r(chem_p,1,n_ra)
2982                     +omega_02
2983                     +deriv_r(M,1,n_ra);
2984
2985
2986         ////////////////////////////////////
2987
2988
2989           /* Virial test */
2990
2991          vt=fabs(2.0*ke+w+3.0*(Pi_p/Np_index+Pi_n/Nn_index)+Emag)/fabs(w);
2992
```

```
} //end of function


//&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
//********************************************************************** Print the
    results
void print_results(int n_rb1,double r_ratio1){

  FILE *fres;

   fres = fopen("results.txt", "w");

  printf("******************************************************************************\n\n");
  printf("(rigid) Rotating Magnetized (type-II) Superconductive Neutron Stars  \n");
  printf("------------------------------------------------------------\n\n");
  printf("Grid dimensions  r  mu   %d  %d \n",NDIV,KDIV);
  printf("Proton N index:                  %2.1f \n",Np_index);
  printf("Neutron N index:                 %2.1f \n",Nn_index);
  printf("Central proton fraction:         %3.2f \n",x_p_0);
  printf("k0 constant:                     %3.2e \n",k0/Sqrt_4PI);
  printf("alpha constant:                  %3.2e \n",alpha_c*Sqrt_4PI);
  printf("zeta constant:                   %3.2e \n",zeta);
  printf("superconductivity constant h_c: %3.2e \n",h_c);
  printf("underrelaxation parameter:       %3.2e \n",undrlx_c);
  printf("-----------------------------------------------------------\n\n");

  printf("Surf Eq point | Surf Pole point | CC bound Eq point |\n");
  printf("     %d        |      %d         |         %d         |\n\n",n_ra,n_rb1,cc_ra);
  printf("               r_proton_pole | r_cc_ra          | \n");
  printf("                        %3.2e  |  %3.2e  | \n\n",r[n_rb1],r[cc_ra]);

  printf("r_cc_eq      |  r_cc_pole |\n");
  printf(" %3.2e  |  %3.2e  | \n\n",r_cc[1],r_cc[KDIV]);
  printf("r_neut_eq    |r_neut_pole  | \n");
  printf("  %3.2e  |  %3.2e  | \n\n",r_neutron[1],r_neutron[KDIV]);
  printf("-----------------------------------------------------------\n\n");
  printf("Omega_0^2    |Omega_K^2    |Omega_0^2/Omega_K^2  | \n");
```

```
3033        printf("  %3.2e  |  %3.2e  |        %3.2e        |
                \n\n",omega_02/(4.0*PI),omega_k2/(4.0*PI),omega_02/omega_k2);
3034        printf("    M       |    V        |    J        | Mu_p_max    | Mu_n_max    | \n");
3035        printf("   %3.2e |  %3.2e  |  %3.2e  |  %3.2e  |  %3.2e  |
                \n\n",m,v,am,chem_p_max,chem_n_max);
3036        printf("-------------------------------------------------------\n\n");
3037
3038        printf(" Emag/|W|    |  U/|W|      |  T/|W|      |    W        | U_p/|W|     | U_n/|W|
                |    C_p       |  C_dif | \n");
3039        printf("   %3.2e |  %3.2e  |  %3.2e  |  %3.2e  |  %3.2e  |   %3.2e |  %3.2e  |  %3.2e
                 |\n\n",
3040       Emag/fabs(w),(Pi_p+Pi_n)/fabs(w),ke/fabs(w),fabs(w)/(4.0*PI),Pi_p/fabs(w)
3041           ,Pi_n/fabs(w),C_p/(4.0*PI),C_dif/(4.0*PI));
3042        printf("  Emag_tor/Emag :  %3.2e   \n",Emagtor/Emag);
3043        printf("-------------------------------------------------------------------------------\n\n");
3044
3045        printf("Virial Test  :   %3.2e          |   Total num mag charge :  %3.5e
                \n",vt,magcharge);
3046        printf("                                |Total B flux upper hemisphere :  %3.5e
                \n\n",BFlux_surf);
3047
3048        printf("Iterations   :   %d \n\n",counter);
3049
3050        printf(" Magnetic field strength ratio   %f   Bpol  %3.5e G
                \n",magratio,3.87*1.0e+17*B[KDIV][n_rb1]);
3051
3052        printf("*********************************************************************\n\n");
3053
3054
3055        //---------------------- in file
3056
3057        fprintf(fres,"*********************************************************************\n\n");
3058        fprintf(fres,"(rigid) Rotating Magnetized (type-II) Superconductive Neutron Stars
                \n");
3059        fprintf(fres,"-------------------------------------------------------\n\n");
3060        fprintf(fres,"Grid dimensions  r  mu   %d  %d \n",NDIV,KDIV);
3061        fprintf(fres,"Proton N index:                  %2.1f \n",Np_index);
3062        fprintf(fres,"Neutron N index:                 %2.1f \n",Nn_index);
3063        fprintf(fres,"Central proton fraction:         %3.2f \n",x_p_0);
3064        fprintf(fres,"k0 constant:                     %3.2e \n",k0/Sqrt_4PI);
3065        fprintf(fres,"alpha constant:                  %3.2e \n",alpha_c*Sqrt_4PI);
```

```
3066      fprintf(fres,"zeta constant:                    %3.2e \n",zeta);
3067      fprintf(fres,"superconductivity constant h_c: %3.2e \n",h_c);
3068      fprintf(fres,"underrelaxation parameter:        %3.2e \n",undrlx_c);
3069      fprintf(fres,"-------------------------------------------------------\n\n");
3070
3071      fprintf(fres,"Surf Eq point | Surf Pol point | CC bound Eq point |\n");
3072      fprintf(fres,"     %d         |      %d          |         %d
              |\n",n_ra,n_rb1,cc_ra);
3073      fprintf(fres,"            r_p_pol      | r_eq_boundary      | \n");
3074      fprintf(fres,"               %3.2e   |   %3.2e   | \n\n",r[n_rb1],r[cc_ra]);
3075
3076      fprintf(fres,"r_cc_eq     |   r_cc_pole |\n");
3077      fprintf(fres," %3.2e   |   %3.2e   | \n\n",r_cc[1],r_cc[KDIV]);
3078      fprintf(fres,"r_neut_eq    |r_neut_pole   | \n");
3079      fprintf(fres,"   %3.2e   |   %3.2e   | \n\n",r_neutron[1],r_neutron[KDIV]);
3080      fprintf(fres,"-------------------------------------------------------\n\n");
3081      fprintf(fres,"Omega_0^2    |Omega_K^2    |Omega_0^2/Omega_K^2  | \n");
3082      fprintf(fres,"   %3.2e   |   %3.2e   |        %3.2e         | \n\n",omega_02/(4.0*PI),
3083       omega_k2/(4.0*PI),omega_02/omega_k2);
3084      fprintf(fres,"     M         |    V         |     J         | Mu_p_max    | Mu_n_max    | \n");
3085      fprintf(fres,"    %3.2e |  %3.2e  |  %3.2e  |  %3.2e  |  %3.2e  |
              \n\n",m,v,am,chem_p_max,chem_n_max);
3086      fprintf(fres,"-------------------------------------------------------\n\n");
3087
3088      fprintf(fres," Emag/|W|    |  U/|W|       |  T/|W|       |     W         | U_p/|W|      |
              U_n/|W|      |    C_p        |  C_dif  | \n");
3089      fprintf(fres,"    %3.2e |  %3.2e  |  %3.2e  |  %3.2e  |  %3.2e  |    %3.2e |  %3.2e  |
              %3.2e  |\n\n",
3090     Emag/fabs(w),(Pi_p+Pi_n)/fabs(w),ke/fabs(w),fabs(w)/(4.0*PI),Pi_p/fabs(w),
3091      Pi_n/fabs(w),C_p/(4.0*PI),C_dif/(4.0*PI));
3092      fprintf(fres,"  Emag_tor/Emag :  %3.2e   \n",Emagtor/Emag);
3093      fprintf(fres,"-------------------------------------------------------------------------------\n\n");
3094
3095      fprintf(fres,"Virial Test  :   %3.2e          |    Total num mag charge :  %3.5e
              \n\n",vt,magcharge);
3096      fprintf(fres,"                                |Total B flux upper hemisphere :  %3.5e
              \n\n",BFlux_surf);
3097
3098      fprintf(fres,"Iterations   :   %d \n\n",counter);
3099
```

```
3100      fprintf(fres," Magnetic field strength ratio   %f   Bpol  %3.5e G
              \n",magratio,3.87*1.0e+17*B[KDIV][n_rb1]);
3101
3102      fprintf(fres,"*************************************************************************\n\n");
3103
3104      fclose(fres);
3105
3106    }
3107
3108
3109    //************************************************ Export in files
3110    void export_in_files(void){
3111
3112     int i,
3113         j;
3114
3115      /* Export in files */
3116    FILE *fr,
3117         *fmu,
3118
3119         *frho,
3120         *frho_p,
3121         *frho_n,
3122         *fchem_n,
3123         *fchem_p,
3124         *fu,
3125         *fphi,
3126
3127         *frbound,
3128         *frneutron,
3129         *frcc,
3130
3131         *flast_r_p,
3132         *flast_mu_p,
3133         *flast_r_cc,
3134         *flast_mu_cc,
3135
3136         *fdf_du,
3137         *fBcc,
3138         *fmag_f_N,
3139         *fM,
```

```
3140            *fucc,
3141
3142            *fPi_fun,
3143            *fgrad_Pi_r,
3144            *fgrad_Pi_theta,
3145            *fgrad_u_r,
3146            *fgrad_u_theta,
3147            *fF,
3148
3149            *fB,
3150            *fBpol,
3151            *fBtor,
3152
3153            *fBdiv;
3154
3155            fr = fopen("r.txt", "w");
3156            fmu = fopen("mu.txt", "w");
3157            frho = fopen("rho.txt", "w");
3158            frho_p = fopen("rho_p.txt", "w");
3159            frho_n = fopen("rho_n.txt", "w");
3160            fchem_n = fopen("ch_n.txt", "w");
3161            fchem_p = fopen("ch_p.txt", "w");
3162            fu      = fopen("u.txt", "w");
3163            frbound = fopen("rbound.txt", "w");
3164            frneutron=fopen("rneutron.txt", "w");
3165            frcc=fopen("rcc.txt","w");
3166            fphi=fopen("phi.txt","w");
3167            fBpol=fopen("Bpol.txt","w");
3168            fBtor=fopen("Btor.txt","w");
3169            fB=fopen("B.txt","w");
3170
3171            fBdiv=fopen("divB.txt","w");
3172
3173            flast_r_p=fopen("last_r_p.txt","w");
3174            flast_mu_p=fopen("last_mu_p.txt","w");
3175            flast_r_cc=fopen("last_r_cc.txt","w");
3176            flast_mu_cc=fopen("last_mu_cc.txt","w");
3177
3178            fdf_du=fopen("df_du.txt","w");
3179            fBcc=fopen("Bcc.txt","w");
3180            fmag_f_N=fopen("mag_f_N.txt","w");
```

```
3181        fM=fopen("M.txt","w");
3182        fucc=fopen("ucc.txt","w");
3183
3184        fPi_fun=fopen("Pi_fun.txt","w");
3185        fgrad_Pi_r=fopen("gradPi_r.txt","w");
3186        fgrad_Pi_theta=fopen("gradPi_theta.txt","w");
3187        fgrad_u_r=fopen("gradu_r.txt","w");
3188        fgrad_u_theta=fopen("gradu_theta.txt","w");
3189        fF=fopen("F.txt","w");
3190
3191        for(i=1;i<=KDIV;i++){
3192          for(j=1;j<=NDIV;j++){
3193
3194          fprintf(frho," %4.14f ",rho[i][j]);
3195          fprintf(frho_p," %4.14f ",rho_p[i][j]);
3196          fprintf(frho_n," %4.14f ",rho_n[i][j]);
3197          fprintf(fchem_p," %4.14f ",chem_p[i][j]);
3198          fprintf(fchem_n," %4.14f ",chem_n[i][j]);
3199          fprintf(fu," %4.14f ",u[i][j]);
3200          fprintf(fphi," %4.14f ",phi[i][j]);
3201          fprintf(fBpol," %4.14f ",B_pol_norm[i][j]);
3202          fprintf(fBtor," %4.14f ",B_tor_norm[i][j]);
3203          fprintf(fB," %4.14f ",B[i][j]);
3204
3205          fprintf(fBdiv," %4.14f ",divB[i][j]);
3206
3207          fprintf(fdf_du," %4.14f ",df_du[i][j]);
3208          fprintf(fmag_f_N," %4.14f ",mag_f_N[i][j]);
3209          fprintf(fM," %4.14f ",M[i][j]);
3210
3211          fprintf(fPi_fun," %4.14f ",Pi_fun[i][j]);
3212          fprintf(fgrad_Pi_r," %4.14f ",grad_Pi_r[i][j]);
3213          fprintf(fgrad_Pi_theta," %4.14f ",grad_Pi_theta[i][j]);
3214          fprintf(fgrad_u_r," %4.14f ",grad_u_r[i][j]);
3215          fprintf(fgrad_u_theta," %4.14f ",grad_u_theta[i][j]);
3216          fprintf(fF," %4.14f ",F[i][j]);
3217
3218
3219            } fprintf(frho,"\n");
3220        fprintf(frho_p,"\n");
3221        fprintf(frho_n,"\n");
```

```
3222          fprintf(fchem_p,"\n");
3223          fprintf(fchem_n,"\n");
3224                  fprintf(fu,"\n");
3225                  fprintf(fphi,"\n");
3226                  fprintf(fBpol,"\n");
3227                  fprintf(fBtor,"\n");
3228          fprintf(fB,"\n");
3229
3230          fprintf(fBdiv,"\n");
3231
3232          fprintf(fdf_du,"\n");
3233          fprintf(fmag_f_N,"\n");
3234          fprintf(fM,"\n");
3235
3236          fprintf(fPi_fun,"\n");
3237          fprintf(fgrad_Pi_r,"\n");
3238          fprintf(fgrad_Pi_theta,"\n");
3239          fprintf(fgrad_u_r,"\n");
3240          fprintf(fgrad_u_theta,"\n");
3241          fprintf(fF,"\n");
3242    }
3243
3244      for(j=1;j<=NDIV;j++)  fprintf(fr," %4.14f \n",r[j]);
3245      for(i=1;i<=KDIV;i++)  fprintf(frbound," %4.14f \n",r_bound[i]);
3246      for(i=1;i<=KDIV;i++)  fprintf(fmu," %4.14f \n",mu[i]);
3247      for(i=1;i<=KDIV;i++)  fprintf(frneutron," %4.14f \n",r_neutron[i]);
3248      for(i=1;i<=KDIV;i++)  fprintf(frcc," %4.14f \n",r_cc[i]);
3249      for(i=1;i<=KDIV;i++)  fprintf(flast_r_p," %d \n",last_r_p[i]);
3250      for(j=1;j<=NDIV;j++)  fprintf(flast_mu_p," %d \n",last_mu_p[j]);
3251      for(i=1;i<=KDIV;i++)  fprintf(flast_r_cc," %d \n",last_r_cc[i]);
3252      for(j=1;j<=NDIV;j++)  fprintf(flast_mu_cc," %d \n",last_mu_cc[j]);
3253
3254      for(i=1;i<=KDIV;i++)  fprintf(fBcc," %4.14f \n",Bcc[i]);
3255      for(i=1;i<=KDIV;i++)  fprintf(fucc," %4.14f \n",u[i][last_r_cc[i]]);
3256
3257    fclose(fr);
3258    fclose(fmu);
3259    fclose(frho);
3260    fclose(frho_p);
3261    fclose(frho_n);
3262    fclose(fchem_n);
```

```
3263    fclose(fchem_p);
3264    fclose(fu);
3265    fclose(frbound);
3266    fclose(frneutron);
3267    fclose(frcc);
3268    fclose(fphi);
3269    fclose(fBpol);
3270    fclose(fBtor);
3271    fclose(fB);
3272
3273    fclose(fBdiv);
3274
3275    fclose(flast_r_p);
3276    fclose(flast_mu_p);
3277    fclose(flast_r_cc);
3278    fclose(flast_mu_cc);
3279
3280    fclose(fdf_du);
3281    fclose(fmag_f_N);
3282    fclose(fBcc);
3283    fclose(fM);
3284    fclose(fucc);
3285
3286    fclose(fPi_fun);
3287    fclose(fgrad_Pi_r);
3288    fclose(fgrad_Pi_theta);
3289    fclose(fgrad_u_r);
3290    fclose(fgrad_u_theta);
3291    fclose(fF);
3292
3293
3294    }
3295
3296
3297
3298    //    MAIN FUNCTION
3299    //*********************************************************************
3300    main(int argc, char **argv)
3301    {int i,
3302         n_rb;              // grid position of r_p_pol
3303
```

```
3304    double r_ratio;         // r_p_eq/r_p_pol
3305
3306
3307
3308            fdata           = fopen("data.txt","w");
3309
3310     fconv                  = fopen("conv.txt","w");
3311     fconvcc                = fopen("convcc.txt","w");
3312     fconvQ                 = fopen("convQ.txt","w");
3313     fmagcharge             = fopen("magcharge.txt","w");
3314     fFlux                  = fopen("Flux.txt","w");
3315
3316
3317
3318
3319
3320     // Make grid
3321
3322      make_grid();
3323
3324     // Default values
3325
3326       Np_index=1.0;
3327       Nn_index=0.9;
3328       r_ratio=1.0;
3329
3330
3331     // Read Options
3332
3333      for(i=1;i<argc;i++)
3334          if(argv[i][0]=='-'){
3335    switch(argv[i][1]){
3336                    case 'N':
3337      sscanf(argv[i+1],"%lf",&Np_index);
3338                             break;
3339
3340
3341                    case 'r':
3342      sscanf(argv[i+1],"%lf",&r_ratio);
3343                             break;
3344    }
```

```
3345          }
3346
3347
3348       // Parameters values
3349         Nn_index=0.9;
3350         alpha_c=200.0;//(2.0e+7)/Sqrt_4PI;
3351         zeta=1.0;
3352         x_p_0=0.15;
3353         h_c=0.1;
3354         k0=0.005*Sqrt_4PI;
3355
3356         eps=1.0e-6;
3357         undrlx_c=0.25;
3358
3359         //**************************
3360
3361      // Grid position of Proton equator r_p_eq
3362
3363      n_ra=(NDIV-1)/RMAX+1;
3364
3365         //For greater Grid depends on the RMAX ----------------------
3366
3367             //     n_ra=(NDIV-1)/3+1;
3368
3369      // Grid position of Proton pole r_p_pol
3370
3371         n_rb=r_ratio*n_ra;
3372
3373      // Grid Position of Neutron equator
3374
3375         cc_ra=0.9*n_ra;
3376
3377      // Initialize densities
3378
3379         guess_last_points();
3380         guess_density();
3381         guess_u();
3382
3383      // compute mathematical functions for integrals
3384
3385         comp_f_2n_p_2n();
```

```
3386
3387      // Main iteration and compute functions
3388        iterate(n_rb, Np_index,Nn_index);
3389
3390       //  iterate3(n_rb,Np_index,Nn_index,525);
3391
3392       iterate2(51);
3393
3394
3395        comp();
3396
3397        // <Bcc>/Hc1
3398        double a1=0.0;
3399         for(i=1;i<=KDIV;i++)
3400                {
3401           a1=a1+Bcc[i];
3402         }
3403        a1=a1/(1.0*KDIV);
3404        a1=a1/(rho_p[1][cc_ra]*h_c);
3405
3406        magratio=a1;
3407
3408      //  Print results on screen
3409
3410        print_results(n_rb,r_ratio);
3411
3412
3413      // Export in files
3414
3415        export_in_files();
3416
3417
3418        fclose(fdata);
3419
3420
3421        fclose(fconv);
3422        fclose(fconvcc);
3423        fclose(fconvQ);
3424        fclose(fmagcharge);
3425        fclose(fFlux);
3426
```

```
3427
3428
3429     } // END OF PROGRAM
```

Listing C.2: Source code for rotating double fluid magnetized type-II superconductive neutron stars

Bibliography

[1] Adkins C.J., 1983, Equilibrium thermodynamics, Cambridge University Press

[2] Annett J.F., 2003, Superconductivity, Superfluids and Condensates, Oxford University Press.

[3] Andersson N., Glampedakis K., Samuelsson L., 2008, MNRAS, 396, 894.

[4] Chandrasekhar S., 1961, Hydromagnetic Stability, Oxford at the Clarendon Press.

[5] Collins G.W. II, 2003, The Fundamentals of Stellar Astrophysics.

[6] Friedman J.L., Stergioulas N., 2013, Rotating Relativistic Stars, Cambridge Academic Press.

[7] Glampedakis K., Andersson N., Lander S.K., 2012, MNRAS, 420,1263.

[8] Glampedakis K., Andersson N., Samuelsson L., 2011, MNRAS, 410, 805.

[9] Griffiths D.J., 1999, Introduction to Electrodynamics, Prentice Hall.

[10] Hachisu I, 1986, ApJS, 61, 479.

[11] Jackson J.D., 1999, Classical Electrodynamics, John Wiley & Sons, Inc.

[12] Kiuchi K., Kotake K., 2008, MNRAS, 385, 1327.

[13] Komatsu H., Eriguchi Y., Hachisu I., MNRAS, 237, 355.

[14] Lander S.K., Jones D.I., 2009, MNRAS, 395, 2162 .

[15] Lander S.K., Andersson N., Glampedakis K., 2012, MNRAS, 419, 732.

[16] Lander S.K., Jones D.I., 2012, MNRAS, 424, 482.

[17] Lander S.K., 2013, Phys. Rev. Lett., 110, 071101.

[18] Lander S.K., 2014, MNRAS, 437, 424.

[19] Lang S., 1973, Calculus of Several Variables, Addison-Wesley Publishing Company.

[20] Mendell G., 1991, ApJ, 380, 515.

[21] Mendell G., 1991, ApJ, 380, 530.

[22] Passamonti A., Lander S.K., 2014, MNRAS, 438, 156.

[23] Prix R., 2004, Phys. Rev. D., 69, 043001.

[24] Prix R., Comer G.L., Andersson N., 2002, A& A, 381, 178.

[25] Tomimura Y., Eriguchi Y., 2005, MNRAS, 359, 1117.

[26] Tsuneto T., 1998, Superconductivity and Superfluidity, Cambridge University Press.