Analysis of equations of state for neutron star modeling

Joseph E. Nyhan, Prof. Ben Kain

Spring 2022

Abstract

Neutron stars are complex physical objects often modelled as a hydrodynamical system. Within these models, the two parameters energy density and pressure are related by an equation known as the "equation of state". This relationship allows us to calculate energy density if we know pressure, and visa versa. The equation of state itself is important because it encodes the information about interactions between the fundamental particles within the star. Furthermore, as neutron stars are extreme examples of gravitation, and not microscopically observable on Earth, the physics within neutron stars are still not well known. By postulating and analyzing different interactions within a neutron star, and therefore different equations of state, we can simulate the observable characteristics of the resulting star and compare them to empirical data in order to better understand how neutron stars behave on a fundamental level.

Contents

Chapter 1
Introduction

1.1 Neutron Stars

A neutron star is an incredibly dense stellar object, the core left over after a supergiant star undergoes a supernovae explosion. While one may think that a "neutron star" will contain only neutrons, it also contains protons and electrons; the star instead gets its name from the fact that it is overall neutral (meaning it has an equal number of protons and electrons). Neutron stars have radii of approximately 10 km, with masses of about $1\odot$, a solar mass. This explains their incredible density; there is about the same (or more) mass crammed into a neutron star than would barely fit inside Washington, DC. Neutron stars are interesting for research because of their extreme conditions: incredibly strong gravity, high energy densities, and large pressures. Because of these conditions, neutron stars must be described by physics's most advanced and technical theories; by studying these situations, we can better understand how our theories work in these extreme cases, and where they might not be so effective.

1.2 Equations of State

Within our model of a neutron star, there is a fundamental relationship between the *energy density*, denoted ϵ, and the *pressure*, denoted P, known as the *equation of state* (EOS). This relationship can be written in a functional form

$$\epsilon = \epsilon(P) \quad \Longleftrightarrow \quad P = P(\epsilon).$$

Most importantly, this relationship shows that an EOS allows us to know P if we know ϵ, and visa versa.

Within our macroscopic models of neutron stars, the EOS is a way to encode the fundamental interparticle interactions within the star. An EOS is therefore calculated by describing those quantum mechanical, subatomic interactions and then determining what that model predicts for P and ϵ. However, the true model of what occurs within a neutron star is unknown, as the conditions on and within a neutron star are not reproducible on earth for studies; therefore, the true EOS of a neutron star is also unknown.

1.3 Report Outline

This report will explore the derivation, calculation, and predictions of the EOSs resulting from the Quantum Hadrodynamics (QHD) model. In chapter 2, the predictions made by neutron stars will be analyzed through the lens of the Tolman-Oppenheimer-Volkoff (TOV) equations. In chapter 3, Quantum Hadrodynamics and the simple, unrealistic EOS QHD-I will be carefully derived, analyzed, and numerically calculated. In chapter 4, a more advanced EOS model will be analyzed, and its calculation described in detail. Finally, in the appendices, full code files discussed throughout are included.

1.4 Thanks

I want to extend my thanks, first, to Prof. Ben Kain, for all of his help and guidance on this project and in general this year; I have really enjoyed working with him. Second, I want to thank Prof. Oxley for serving as my reader on this thesis. Third, I would like to thank the Holy Cross College Honors program and Holy Cross Department of Physics for the opportunity to write this thesis. Finally, I want to thank my family for all of their help and support over the years; I couldn't have done it without them.

Chapter 2
Static Solutions

Within our study of equations of state, we want to see which predictions each unique equation makes about the macroscopic (observable) properties of a neutron star. To do so, we introduce the Tolman-Oppenheimer-Volkoff (TOV) equations, a time independent description of a spherically symmetric neutron star. By solving the TOV equations, we can calculate theoretical observables, such as the total mass and radius of an individual star, and compare them to empirical data gathered from real neutron stars. This chapter will introduce the TOV equations, show how they are solved, and how through analysis we can determine the aforementioned observable quantities of note.

2.1 The Tolman-Oppenheimer-Volkoff (TOV) Equations

The Tolman-Oppenheimer-Volkoff (TOV) equations are the below system of two coupled differential equations

$$\frac{dm}{dr} = 4\pi r^2 \epsilon, \quad \frac{dP}{dr} = -\frac{(4\pi r^3 P + m)(\epsilon + P)}{r^2(1 - 2m/r)}. \tag{2.1}$$

Within these equations, there are four important variables: the *radial coordinate*, r, the *mass*, m, the *pressure*, P, and the *energy density*, ϵ. Within this model, we consider neutron stars to be spherically symmetric; this means that only the distance from the center of the star is important. Furthermore, the radius r is the independent variable; therefore, the other variables can be written as functions of this radius: $m = m(r)$, $P = P(r)$, and $\epsilon = \epsilon(r)$.

The parameter m is defined as the total amount of energy within a spherical shell of radius r. Technically, this is not identical to mass; however, due to Einstein's mass-energy equivalence, it is common and convenient to call this parameter mass. This paper will continue this convention.

At this point, we have two variables remaining, yet only one unaccounted for equation in this system. It is here we can finally show the importance of the equation of state within our neutron star calculations. Within this system, the energy density and pressure are related directly by an equation $\epsilon = \epsilon(P)$ known as "the equation of state" (EOS). This relationship is important because it allows us to determine the current value of ϵ if we already know the value of pressure.[1] The EOS will be derived and analyzed in depth in the later sections of

[1]Practically, it is also easy to find pressure if we know energy density, however that is not necessary in this calculation.

this paper; at this point, however, it is important to understand that the EOS encodes the interactions between the particles within the neutron star, and based on the model used to describe those interactions, it will change. For this derivation, we leave the EOS as a general function. After including an EOS in our TOV equations, we now just have two variables to evolve: m and P.

To determine a solution to the system of equations in (2.1), we need will solve an initial value problem. As we want to know information about the star from its center radially outward, we therefore need initial conditions for both m and P at the center of the star, $r = 0$. Determining an initial condition for $m(r = 0)$ is straightforward; as m represents the total mass contained within a radius r, at the center of the star, no mass is enclosed, so $m(0) = 0$. We treat the initial condition for P, $P(0)$, called the *central pressure*, as a free parameter. Every static solution is uniquely specified by a value of the central pressure; thus, we simply choose a reasonable value (typically $P(0)$ somewhere between $10^{-6}\,\mathrm{GeV}^4$ and $10^{-1}\,\mathrm{GeV}^4$) and begin our integration. These initial conditions are summarized as

$$m(0) = 0, \quad P(0) \in [10^{-6}\,\mathrm{GeV}^4, 10^{-1}\,\mathrm{GeV}^4]. \tag{2.2}$$

When beginning our integration, we cannot, however, start directly at $r = 0$, as dP/dr has terms containing $1/r$. To determine dP/dr at $r = 0$, we would need to take a limit; however, numerically, we can avoid this by starting at very small r. Therefore, we simply start at a tiny value of r, say $r \approx 10^{-8}$. This effectively approximates the limit and is accurate enough for our purposes.

We want our integration to terminate once we reach the edge of the star, as we are not interested in anything beyond that point. To find this outer edge, we define the total radius of the star, R, as the radius when

$$P(R) = 0. \tag{2.3}$$

In practice, once we reach a very small pressure, $P \sim 10^{-12}$, we can end the integration. Once we have found R, we can determine $M = m(R)$, the total mass enclosed at radius R. The total mass M and total radius R are important to our analyses of different equations of state, as they represent experimentally observable properties of real neutron stars. These theoretically calculated properties can be compared to observed properties to gauge the validity of any given equation of state.

By determining a solution to the TOV equations, we calculate something called a "static solution," a time independent description of a neutron star. A solution contains three curves: $m(r)$, $\epsilon(r)$, and $P(r)$. Of most importance are the pressure and energy density curves; however because they are related by the EOS, we only need to save one curve, as we can simply calculate the other when desired.

2.2 Computing Static Solutions

To compute static solutions, we use numerical integration techniques for solving ordinary differential equations (ODEs). In this situation, we use a specific technique known as the fourth-order Runge-Kutta algorithm (RK4). First, we need a differential equation of the

form

$$\frac{dy}{dx} = f(x, y),$$

where x is the independent variable, and y is the function whose solution we wish to find. Importantly, y can be a vector valued function, so we can evolve a system of coupled differential equations using this method. Given a current value of the function, $y(x)$, RK4 allows to find an approximate value of the function a small step h forward in x: $y(x + h)$. Computationally, the algorithm is

$$y(x_i + h) \cong y(x_i) + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \tag{2.4}$$

where

$$\begin{aligned} k_1 &= hf(x_i, y_i), \\ k_2 &= hf(x_i + h/2, y_i + k_1/2), \\ k_3 &= hf(x_i + h/2, y_i + k_2/2), \\ k_4 &= hf(x_i + h, y_i + k_3). \end{aligned}$$

If y is a vector valued function, then each of the k_i values will also be vector valued. To determine a solution for y, we start with an initial condition, $y(x_0)$. Then, after choosing a step size h, we use the initial value of y to determine k_1 through k_4. Finally, we can then calculate $y(x_0 + h)$ using (2.4). At this point, we repeat the process using the newly calculated values of y as our initial data, allowing us to calculate the value of y one more step forward. This algorithm is repeated as necessary.

For our system of coupled ODEs in (2.1), the independent variable is r and our function is the vector is $y = (m, P)$. Using the initial conditions $y_0 = (0, P_0)$ from (2.2), we can begin our integration from the center of the star and work outwards with step size $h = \Delta r$, a small step in the radial direction. We continually use the newly calculated values of y at some radius r to determine the value of $y(r + \Delta r)$. As we want the integration to terminate when P gets too small, every time we calculate new values, we check to make sure that P is still in an acceptable range, and if it is too small, we terminate.

This method allows us to generate static solution curves for various equations of state $\epsilon(P)$ and central pressures $P(0)$. For an equation of state called "SLy" from [3] with a central pressure $P(0) = 10^{-4}\,\mathrm{GeV}^4$, an example of the resulting curves for pressure and energy density are shown in Figure 2.1.

By observation, in Figure 2.1, we can see that the total radius, R, is about 10 km, as this is when the pressure goes to zero. Using this value, we can determine what the total mass is by plugging it into the mass function (not pictured). These values are vital within section 2.3.

2.3 Analysis of Static Solutions

In Figure 2.1, we looked at one static solution produced using a realistic equation of state called SLy. The decision to use SLy was simply exemplary in this case; we could have used a

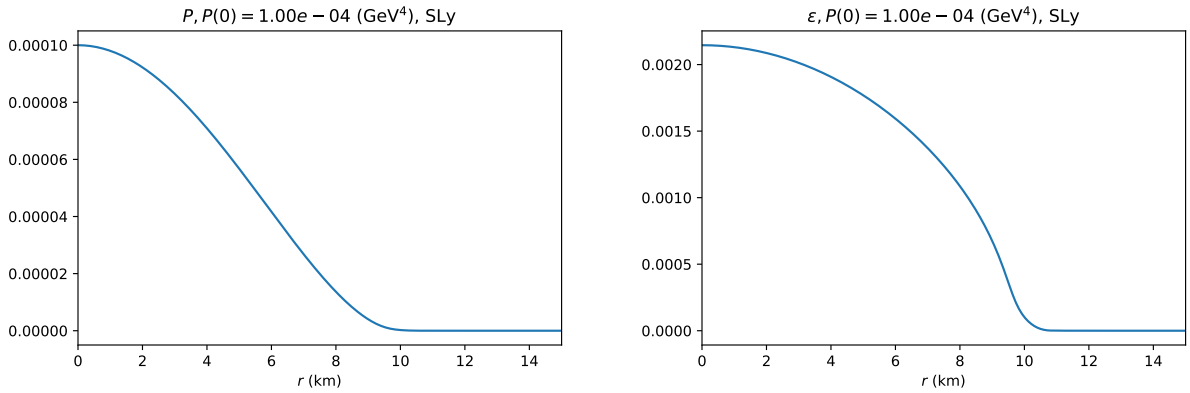

Figure 2.1. The pressure P (left) and energy density ϵ (right) for $P(0) = 10^{-4}\,\mathrm{GeV}^4$ with equation of state SLy from [3].

different equation of state and the static solutions would have been different. To determine the predictions that an individual equation of state will make over a range of values, it is impractical to simply look at static solution curves, such as those in Figure 2.1. Instead, we look at the macroscopic properties, M and R, that each predicts over a wide range of central pressure values.

To do this, we create a range of central pressure values, approximately $P(0) \in [10^{-6}, 10^{-1}]$ (GeV^4). In practice, it is useful to use a logarithmic range of values, such that the values are more densely packed near $P = 10^{-6}\,\mathrm{GeV}^4$; within our code, we use the `numpy` function `logspace` to create an array of values with logarithmic spacing. Now, we can iterate through these pressure values and create static solutions for each central pressure. For every static solution, we integrate until we find R, at which time we also calculate M. These values are stored along with the central pressure value from which they were produced.

Using these values, we plot two curves: M vs. P_0 and M vs. R. These are shown in Figure 2.2.

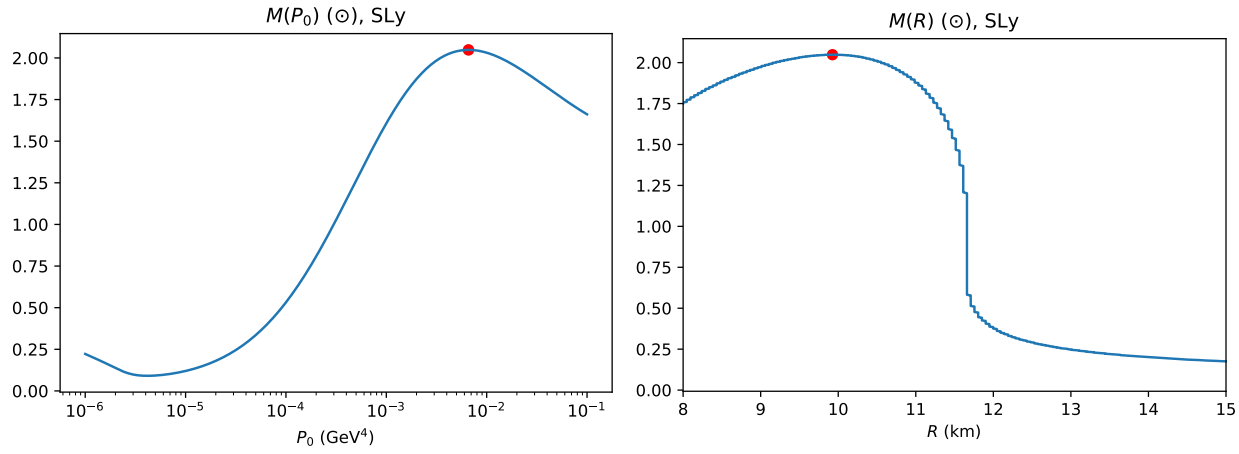

Figure 2.2. The curves M vs. P_0 (left) and M vs. R (right) for the SLy equation of state from [3].

From these curves, we deduce three special values: the *critical pressure*, the *critical radius*, and the *critical mass*. The critical mass is the maximum mass that the EOS predicts, and the critical pressure and critical radius are the pressure and radius, respectively, where the critical mass is reached. The critical mass and critical radius are used as measuring sticks for how realistic an equation of state is; if they are on the same order as the largest neutron stars that have been actually observed, the equation of state is considered to be a candidate for a "correct" description of the star. Usually, the critical mass is on the order of about 2 solar masses, while the critical radius is on the order of about 10 km. Within the above plots, we have converted the units of radius to km and the units of mass to *solar masses*, denoted $\odot$, where $1\,\odot = 1.989 \times 10^{30}$ kg.

In [3], besides the SLy EOS, there is an additional EOS called FPS, derived and implemented in a similar fashion, just with a different parameterization. Furthermore, in another paper by the same authors, [5], there are three additional EOSs of similar form. All of these EOSs are considered realistic equations of state and are computed by creating analytical fits from empirical observational data from neutron stars. For demonstration, we have overlaid the mass-radius and mass-pressure curves to demonstrate the differences in predictions of different equations of state. These curves are shown in Figure 2.3.

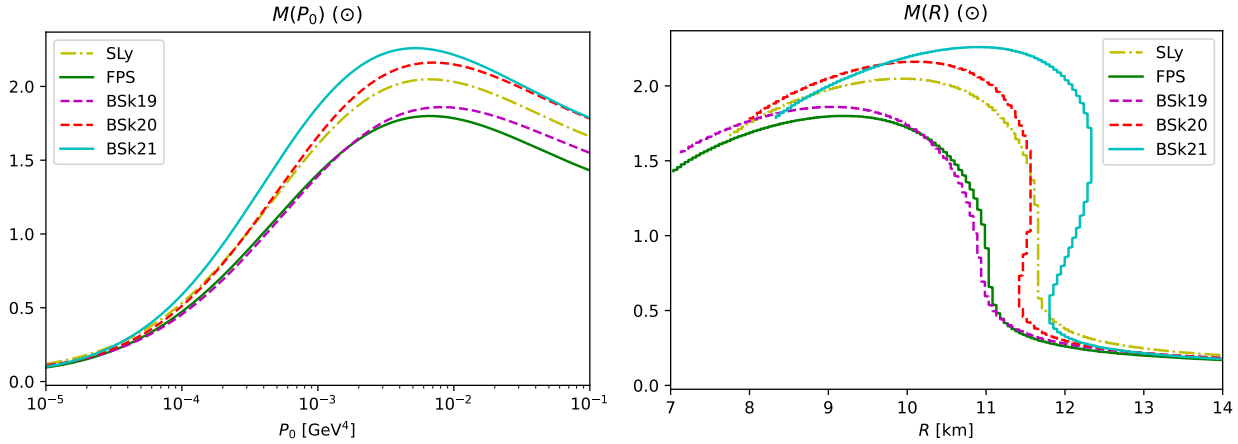

Figure 2.3. The mass-central pressure (left) and mass-radius (right) diagrams for the five equations of state from [3, 5].

As we can see from looking at these curves, these EOSs produce curves that are qualitatively similar yet with slightly different numerical values. For example, the EOS BSk21 predicts a maximum mass of $2.26\,\odot$ and a critical radius of 10.8 km, while FPS predicts a maximum mass of $1.80\,\odot$ with a critical radius of 9.2 km, with the predictions of the three other EOSs falling somewhere in between. We see that these five models therefore produce results that are well within the same order of magnitude with similar qualitative features, however their predictions are most definitely not the same. As the actual EOS within a neutron star is unknown, these differences are non-trivial and studying them may lead to a better understanding of the real interactions within neutron stars in the future.

2.4 Implementation of Static Solutions and Analyses

Throughout this section, important code blocks will be included for ease of reading; however, the entire code file is included at the end of this paper in Appendix A. Any line numbers provided will directly reference this Appendix.

To begin, we wanted to create a program that would allow for an arbitrary equation of state and central pressure to be chosen and would then produce the corresponding static solution. In line 42, the boolean flag `make_static_solution` selects this mode, and the central pressure specified in line 48 is the central pressure used as an initial condition in the TOV equations. Furthermore, lines 12 through 39 are used to specify which equation of state is to be used.

The EOSs that we use when solving the TOV equations are not actually analytical equations or functions defined within our code. Instead, we have various `.txt` files that contain tabulated values of both ϵ and P. When a given EOS is chosen, we read in that text file and use the SciPy function `interp1d` to create an "interpolation function." This function takes the P values as independent variables and ϵ values; then, whenever this function is called throughout the remainder of the code, it will use an interpolation routine to predict an approximate value of ϵ for an inputted value of P. The implementation of this procedure is between lines 70 and 90.[2]

We also initialize other important parameters. We want to produce static solutions from a minimum r value ($r = 0$)[3] out to a maximum specified value, in this case, $r = 200$. Furthermore, we want our points spaced at a regular interval, small enough to give us adequate precision, but not too small as to hinder the code's speed too much. Therefore, we choose our radial step size, `dr`, to be 0.02. Then, we create an array of these r values, spaced out at the desired step size. Finally, we set the initial condition of m to 0. This is all given between lines 56 and 64.

```
def f_onefluid(r,y):
    m, p = y

    rho = rho_from_P(p)

    N = 1-2*m/r

    fm = 4*np.pi*r**2*rho
    frho = (-(4*np.pi*r**3*p + m)*(rho+p)/(r**2*N))

    return fm, frho

def event(r,y):
    m, p = y
```

[2]There are two special EOSs called the "ultra-relativisitic equation of state" (UR EOS) and "polytropic equation of state" that are purely analytical. As this is the case, in lines 84-90, we choose the analytical equations of state if either the UR or polytropic EOS is chosen, and otherwise we use the interpolating function as described above.

[3]As mentioned before, we must choose an actual r_0 value that is just slightly bigger than 0, as using $r = 0$ in (2.1) would cause the denominator of dP/dr to contain a division by zero.

```
    return p - p_tol
```

After the necessary parameters are initialized, we define the functions necessary for integration. In the above code block, we define the vector funciton `f_onefluid`, a function of both the independent variable r and vector function y. Then, y is unpacked into m and P, the equation of state is used to determine ϵ (or, in the code, called `rho`), and finally the right-hand sides of the equations in (2.1) are evaluated and returned as a tuple. Secondly, we define a function called `event`, which is also a function of r and y. This function compares the current P value to a parameter called `p_tol`; `p_tol` is the smallest effective pressure value for which we consider the pressure to have gone to zero. On line 61, we define this threshold to be 10^{-11}. This function will be used to terminate the integration once we have reached the edge of the star.

```
sol = solve_ivp(f_onefluid,[r0,rmax], [m0,p0], method='RK45', atol = 1e
   -12, rtol = 1e-12, t_eval=r_vals, events=event)
```

Now, on line 115, we begin the integration for our static solution. We open files, with the current parameters we are using (such as the radial step size and central pressure) in the file name. In the above code block (and on line 128), we use a SciPy function called `solve_ivp` to perform the numerical integration (using the Runge-Kutta method) and return an array containing the solutions for both P and m. We use `solve_ivp` and do not implement RK4 on our own for speed purposes; `solve_ivp` has special optimizations in regards to step size under the hood that make the calculations more efficient when the values are very large and very small. Importantly, we pass the `event` function to `solve_ivp` as a parameter; throughout the integration, `solve_ivp` checks to see if the pressure has dropped below the tolerance, and if it has, it terminates the integration there. Then, we write the static solution to a file, where we fill out any points after the termination but before the $r_{\max}$ value with zeros. These files are then subsequently read into a different program for plotting, producing the curves like those shown in Figure 2.1.

For the analyses (i.e. producing the mass-radius and mass-pressure curves), we use a similar procedure to that described above, instead we iterate over a wide range of pressure values. If the boolean flag `p0_analysis` is set on line 43, then we will enter the loop on line 149. From there, we loop over all P_0 values in the `p0_vals` array, creating a static solution for each and saving the critical mass, critical radius, and critical pressure values to a file. Then after the completion of the loop, we can use an additional program to create a plot. When calculating the critical mass, we use an optimization routine to determine the top of the mass-pressure or mass-radius curve more precisely, and use that calculated critical value to determine the critical radius or pressure, respectively.

For the remainder of this project, this program will be used to produce the plots and analyses for the equations of state that we derive.

Chapter 3
Quantum Hadrodynamics and the QHD-I Parameter Set

This chapter will describe the derivation of an equation of state from Quantum Hadrodynamics and the QHD-I parameter set, as described within [1].

3.1 Introduction

Quantum Hadrodynamics (QHD) is a model of nuclear matter within a neutron star. Within this model, the forces and interactions between *baryons* (e.g. protons and neutrons) are modelled by the exchange of particles known as *mesons*. QHD, importantly, requires some form of experimental input in order for the model to be complete and make predictions; this experimental input is given in terms of *coupling constants*, which quantify the strength of the interactions between the baryons and mesons. These coupling constants are unique to neutron stars, and therefore must be calculated by analyzing the matter within a neutron star. This is very difficult, however, because the intense densities and pressures present in neutron stars are not reproducible on earth. Therefore, physicists have interpolated data from other nuclear experiments and from observations of neutron stars to compute coupling constants; due to this "interpolation" process, many parameter sets, all slightly different, have been proposed. Finally, the equations of QHD can be difficult to solve, so simplifications can be made to make finding a solution more practical.

In this chapter, the formalism behind calculating the equations to be solved from QHD will be analyzed. Specifically, we will look at the simplest QHD parameter set, called QHD-I. Then, we will make simplifications in the form of the *relativistic mean field* approximations. Finally, we will numerically calculate the EOS from the equations we have derived along the way, and discuss its predictions by analyzing the results produced by the TOV equations.

3.2 Derivation of equations of motion

Within Quantum Hadrodynamics I (QHD-I), there are three important fields: $\phi(x)$, the *scalar meson field*, $V_\mu(x)$, the *vector meson field*, and $\psi(x)$, the *baryon field*. We also define the *Dirac adjoint baryon field*, given by $\bar{\psi}(x) = \psi^\dagger(x)\gamma^0$. Using these fields, we write the

Lagrange density for the QHD-I parameter set

$$\begin{aligned}\mathscr{L} &= \bar{\psi}[\gamma_\mu(i\partial^\mu - g_\nu V^\mu) - (M - g_s\phi)]\psi \\ &\quad + \frac{1}{2}\big(\partial_\mu\phi\partial^\mu\phi - m_s^2\phi^2\big) - \frac{1}{4}V_{\mu\nu}V^{\mu\nu} + \frac{1}{2}m_\omega^2 V_\mu V^\mu, \end{aligned} \tag{3.1}$$

where $V_{\mu\nu} \equiv \partial_\mu V_\nu(x) - \partial_\nu V_\mu(x)$, g_ν and g_ω are the coupling constants, and M, m_s, and m_ω are the baryon, scalar meson, and vector meson masses, respectively. This Lagrange density is given in [1, p. 56].

The Euler-Lagrange equations, for a Lagrange density $\mathscr{L}$ over a classical field φ_α, are given by

$$\partial_\nu\left(\frac{\partial\mathscr{L}}{\partial(\partial_\nu\varphi_\alpha)}\right) - \frac{\partial\mathscr{L}}{\partial\varphi_\alpha} = 0. \tag{3.2}$$

By solving the Euler-Lagrange equations, we can determine the equations of motion, which tells us how the system behaves. To determine these equations of motion, we must apply (3.2) to (3.1) for each field in the system: $\varphi_\alpha = \{\phi, V_\mu, \psi, \bar{\psi}\}$.

For the scalar meson field, when $\varphi_\alpha = \phi(x)$, the first term in (3.2) gives

$$\partial_\nu\left(\frac{\partial\mathscr{L}}{\partial(\partial_\nu\phi)}\right) = \frac{1}{2}\left[\frac{\partial}{\partial(\partial_\nu\phi)}(\partial_\mu\partial^\mu\phi)\right] = \partial_\nu\partial^\nu\phi,$$

while the second term gives

$$\frac{\partial\mathscr{L}}{\partial\phi} = \bar{\psi}[+g_s]\psi + \frac{1}{2}(-m_s(2\phi)) = g_s\bar{\psi}\psi - m_s\phi.$$

Combining, we get the first equation of motion,

$$\partial_\nu\left(\frac{\partial\mathscr{L}}{\partial(\partial_\nu\phi)}\right) - \frac{\partial\mathscr{L}}{\partial\phi} = \partial_\nu\partial^\nu\phi - (g_s\bar{\psi}\psi - m_s\phi) = 0 \quad \Rightarrow \quad \partial_\nu\partial^\nu\phi + m_s\phi = g_s\bar{\psi}\psi. \tag{3.3}$$

This is the form given in [1] (8.1a).

For the vector meson field, when $\varphi_\alpha = V_\mu$, the first term in (3.2) gives

$$\partial_\nu\left(\frac{\partial\mathscr{L}}{\partial(\partial_\nu V_\mu)}\right) = \partial_\nu V^{\mu\nu},$$

after many simplifications, including using the definition of $V^{\mu\nu}$ and the relabelling of indices. The second term gives

$$\frac{\partial\mathscr{L}}{\partial V_\mu} = \frac{\partial}{\partial V_\mu}\bar{\psi}\left[\gamma_\alpha(-g_v V^\alpha) + \frac{1}{2}m_\omega^2 V^\alpha V_\alpha\right] = -g_v\bar{\psi}\gamma^\mu\psi + m_\omega^2 V^\mu,$$

and combining we have

$$\partial_\nu\left(\frac{\partial\mathscr{L}}{\partial(\partial_\nu V_\mu)}\right) - \frac{\partial\mathscr{L}}{\partial V_\mu} = \partial_\nu V^{\mu\nu} - (-g_v\bar{\psi}\gamma^\mu\psi + m_\omega^2 V^\mu) = 0. \tag{3.4}$$

However, this is not the form given in [1]; to reach his form, we leverage the anti-symmetry of $V_{\mu\nu}$, namely

$$V_{\mu\nu} = -V_{\nu\mu}.$$

Therefore, we make the above substitution, multiply (3.4) by -1, and send $\mu \leftrightarrow \nu$ to obtain

$$\partial_\mu V^{\mu\nu} + m_\omega^2 V^\nu = g_v \bar{\psi}\gamma^\nu\psi, \tag{3.5}$$

as given in [1] (8.1b).

Next, we have the two equations of motion from the baryon field. For $\varphi_\alpha = \bar{\psi}$, applying (3.2) is straight forward, as there is no $\partial_\nu\bar{\psi}$ dependence in $\mathcal{L}$, so the first term in (3.2) is zero. Thus, we obtain

$$\partial_\nu\left(\frac{\partial\mathcal{L}}{\partial(\partial_\nu\bar{\psi})}\right) - \frac{\partial\mathcal{L}}{\partial\bar{\psi}} = [\gamma_\mu(i\partial^\mu - g_v V^\mu) - (M - g_s\phi)]\psi = 0. \tag{3.6}$$

For the final case, when $\varphi_\alpha = \psi$, the first term in (3.2) gives

$$\partial_\nu\left(\frac{\partial\mathcal{L}}{\partial(\partial_\nu\psi)}\right) = \partial_\nu\left[\frac{\partial}{\partial(\partial_\nu\psi)}\bar{\psi}i\gamma_\alpha\partial^\alpha\psi\right] = i\partial_\nu\bar{\psi}\gamma^\nu,$$

while the second gives

$$\frac{\partial\mathcal{L}}{\partial\psi} = \bar{\psi}[\gamma_\mu(i\partial^\mu - g_v V^\mu) - (M - g_s\phi)].$$

Combining, we get our fourth equation

$$i\partial_\nu\bar{\psi}\gamma^\nu - \bar{\psi}[\gamma_\mu(i\partial^\mu - g_v V^\mu) - (M - g_s\phi)] = 0. \tag{3.7}$$

In summary, here are the four equations of motion for QHD-I:

$$\partial_\nu\partial^\nu\phi + m_s^2\phi = g_s\bar{\psi}\psi, \tag{3.3}$$
$$\partial_\mu V^{\mu\nu} + m_\omega^2 V^\nu = g_v\bar{\psi}\gamma^\nu\psi, \tag{3.5}$$
$$[\gamma_\mu(i\partial^\mu - g_v V^\mu) - (M - g_s\phi)]\psi = 0, \tag{3.6}$$
$$i\partial_\nu\bar{\psi}\gamma^\nu - \bar{\psi}[\gamma_\mu(i\partial^\mu - g_v V^\mu) - (M - g_s\phi)] = 0. \tag{3.7}$$

The first three are given in [1]. These equations of motion will be used throughout the remainder of this derivation; they are the foundation for our understanding of how the system behaves.

3.3 Relativistic Mean Field Simplifications

In [1] §8.3, we find that the equations of motion listed above are very difficult to solve in their current form. To make them more manageable, we approximate them with *relativistic*

mean field (RMF) simplifications, where we take each field to be its ground state expectation value. For the meson fields, this simplification yields

$$\phi \quad \Rightarrow \quad \langle \Phi | \, \phi \, | \Phi \rangle = \langle \phi \rangle \equiv \phi_0, \tag{3.8}$$

$$V_\mu \quad \Rightarrow \quad \langle \Phi | \, V_\mu \, | \Phi \rangle = \langle V_\mu \rangle \equiv \delta_{\mu 0} V_0, \tag{3.9}$$

where $|\Phi\rangle$ represents the ground state. These results arise from arguing that, in their ground states, ϕ and V_μ should be independent of space and time, as the system is bot uniform and stationary; therefore, ϕ_0 and V_0 are constants. Furthermore, because the system is at rest and the baryon flux, $\bar{\psi}\gamma^i\psi$, is zero, the spatial components of the expected value of V_μ, $\langle V_\mu \rangle$, must vanish [1].

For the baryon field, a "normal order" (i.e. normalized) expectation value must be taken, as because otherwise, the vacuum would be taken into account and the traditional expectation value would diverge. This "normal ordered" expectation value is denoted with a ":". Thus, these are the expectation values for the baryon field

$$\bar{\psi}\psi \quad \Rightarrow \quad \langle \Phi | : \bar{\psi}\psi : | \Phi \rangle = \left\langle \bar{\psi}\psi \right\rangle, \tag{3.10}$$

$$\bar{\psi}\gamma^\mu\psi \quad \Rightarrow \quad \langle \Phi | : \bar{\psi}\gamma^\mu\psi : | \Phi \rangle = \left\langle \psi\gamma^0\psi \right\rangle. \tag{3.11}$$

Intuitively, these simplifications mean that the system is approximated as its "average." Before the simplifications, there are high-energy, strong interactions, as well as low-energy, weak interactions; now, we instead treat every interaction as *the average* interaction. Given these simplifications, we can rewrite the equations of motion given in equations (3.3), (3.5), (3.6), and (3.7) as

$$m_s^2\phi_0^2 = g_s \left\langle \bar{\phi}\phi \right\rangle \tag{3.12}$$

$$m_\omega^2 V_0 = g_v \left\langle \bar{\phi}\gamma^0\phi \right\rangle \tag{3.13}$$

$$[i\gamma_\mu\partial^\mu - g_v\gamma_0 V_0 - (M - g_s\phi_0)]\psi = 0 \tag{3.14}$$

$$i\partial_\mu\bar{\psi}\gamma^\mu - \bar{\psi}[i\gamma_\mu\partial^\mu - g_v\gamma_0 V_0 - (M - g_s\phi)] = 0. \tag{3.15}$$

From [1, p. 40] (6.14), where we consider the star to be a spherically symmetric, stationary fluid, we get the following forms for ϵ and P

$$\epsilon = \left\langle T^{00} \right\rangle, \quad P = \frac{1}{3}\left\langle T^{ii} \right\rangle, \tag{3.16}$$

where $T^{\mu\nu}$ is the energy-momentum tensor, given in [1, p. 40] (6.13), defined by

$$T^{\mu\nu} = \frac{\partial \mathcal{L}}{\partial(\partial_\mu\varphi_\alpha)}\partial^\nu\varphi_\alpha - \mathcal{L}\eta^{\mu\nu}. \tag{3.17}$$

Given the simplifications at the beginning of this section, we can form the Lagrangian Density for an RMF QHD-1 framework, a modification of (3.1). Making the substitutions $\phi \to \phi_0$ and $V_\mu = \delta_{\mu 0}V_0$, we get

$$\mathcal{L}_{\text{RMF}} = \bar{\psi}[i\gamma_\mu\partial^\mu - g_v\gamma_0 V_0 - (M - g_s\phi_0)]\psi - \frac{1}{2}m_s^2\phi_0^2 + \frac{1}{2}m_\omega^2 V_0^2, \tag{3.18}$$

as given in [1]. Importantly, many of the kinetic terms in $\mathcal{L}$ vanish, as the derivative of a constant (which both V_0 and ϕ_0 are) is zero.

Next, we form the energy momentum tensor using (3.17) with $\varphi_\alpha \in \{\phi_0, V_0, \psi\}$. There is no dependence of $\mathcal{L}_{\text{RMF}}$ on the derivatives of ϕ_0 and V_0, so for those values of φ_α, the first term in (3.17) is zero. For ψ, however, there is a ∂^μ within the first term in the brackets, so we must deal with a $\partial_\mu\psi$ derivative.

To start, we distribute the ψ and $\bar{\psi}$ into the brackets, and notice that only the first term has any $\partial_\mu\psi$ dependence, so

$$\frac{\partial\mathcal{L}_{\text{RMF}}}{\partial(\partial_\mu\psi)}\partial^\nu\varphi_\alpha = \frac{\partial}{\partial(\partial_\mu\psi)}\left[\bar{\psi}i\gamma_\sigma\partial^\sigma\psi\right]\partial^\nu\psi + \cancelto{0}{\frac{\partial}{\partial(\partial_\mu\psi)}[\cdots]}\ \partial^\nu\psi = i\bar{\psi}\gamma^\mu\partial^\nu\psi. \tag{3.19}$$

The second term in (3.17) is straightforward. However, we can simplify by substituting (3.6) into the Lagrangian density when we form this term, such that

$$\begin{aligned}\mathcal{L}\eta^{\mu\nu} &= \eta^{\mu\nu}\left(\bar{\psi}[i\gamma_\mu\partial^\mu - g_v\gamma_0 V_0 - (M - g_s\phi_0)]\psi - \frac{1}{2}m_s^2\phi_0^2 + \frac{1}{2}m_\omega^2 V_0^2\right)\\ &= \eta^{\mu\nu}\left(-\frac{1}{2}m_s^2\phi_0^2 + \frac{1}{2}m_\omega^2 V_0^2\right).\end{aligned} \tag{3.20}$$

Therefore, by (3.17), the RMF energy momentum tensor takes the form

$$T^{\mu\nu}_{\text{RMF}} = i\bar{\psi}\gamma^\mu\partial^\nu\psi - \eta^{\mu\nu}\left(-\frac{1}{2}m_s^2\phi_0^2 + \frac{1}{2}m_\omega^2 V_0^2\right). \tag{3.21}$$

If we now use this result in (3.16), we obtain equations for ϵ and P

$$\epsilon = \left\langle T^{00}\right\rangle = \left\langle i\bar{\psi}\gamma^0\partial^0\psi\right\rangle + \frac{1}{2}m_s^2\phi_0^2 - \frac{1}{2}m_\omega^2 V_0^2, \tag{3.22}$$

$$P = \frac{1}{3}\left\langle T^{ii}\right\rangle = \left\langle i\bar{\psi}\gamma^i\partial^i\psi\right\rangle - \frac{1}{2}m_s^2\phi_0^2 + \frac{1}{2}m_\omega^2 V_0^2. \tag{3.23}$$

Now that we have expressions for ϵ and P, we need to evaluate the expectation values in order to calculate numerical values for the equation of state.

Within [1] §8.4, there is a lengthy derivation of the expectation values present in (3.12), (3.13), (3.22), and (3.23). These expectation values are non-trivial to derive, but are beyond the scope of this paper. Therefore, we will take the results given in [1] and use them in calculation.

After evaluating the expectation values, we have

$$\phi_0 = \frac{g_s}{m_s^2}\frac{1}{\pi^2}\int_0^{k_f} dk\,\frac{k^2 m^*}{\sqrt{k^2 + m^{*2}}}, \tag{3.24}$$

$$V_0 = \frac{g_v}{m_\omega^2}\frac{k_f^3}{3\pi^2}, \tag{3.25}$$

where once again, $m^* \equiv M - g_s\phi$ is the *reduced mass*. If we assume spherical symmetry and substitute in the results for the expectation values into (3.22) and (3.23), we get the

following expressions

$$\epsilon = \frac{1}{2}m_s^2\phi_0^2 + \frac{1}{2}m_\omega^2 V_0^2 + \frac{1}{\pi^2}\int_0^{k_f} dk\, k^2\sqrt{k^2 + m^{*2}}, \tag{3.26}$$

$$P = -\frac{1}{2}m_s^2\phi_0^2 + \frac{1}{2}m_\omega^2 V_0^2 + \frac{1}{3}\left(\frac{1}{\pi^2}\int_0^{k_f} dk\, \frac{k^4}{\sqrt{k^2 + m^{*2}}}\right). \tag{3.27}$$

In the next section, we will evaluate these equations to calculate numerical values for the equation of state.

3.4 Numerical Generation of the Equation of State

During this section, important code blocks will be included for ease of reading. The entire code file is provided at the end of this paper in Appendix C.

We wish to generate tabulated values of the equation of state using the equations for ϵ, P and ϕ_0 at the end of the previous section. Simply, this process requires looping through various values of k_f; at each iteration, we find the corresponding value of ϕ_0 by using a root finding routine on (3.24), as m^* depends on ϕ_0, then computing V_0 independently using (3.25), and then finally substituting those values into (3.26) and (3.27) and storing those values. After creating a table of values for ϵ and P, we can verify the validity of the EOS by solving the TOV equations and comparing the results of static solutions, mass-radius curves, and mass-pressure curves to other, previously calculated equations of state. An in-depth description of the TOV equations and their implementation is given in chapter 2.

To begin, we choose a value of k_f for the entirety of the iteration; then, we find the value of ϕ_0 for that k_f value. We define a function in Python for ϕ_0 from (3.24), given below.

```
def f_phi0(phi0, kf):
    mstar = M - g_s * phi0

    def f(k):
        return k**2 * mstar/sqrt(k**2 + mstar**2)
    integral, err = quad(f,0,kf)

    return phi0 - (g_s/m_s**2) * (gamma/(2*pi**2)) * integral
```

While most of the function is straight forward, the middle block may be cryptic without context. These lines are responsible for numerically computing the integral at the end of (3.24), given the current k_f value. `f(k)` is simply a function definition for the integrand of that integral, while the final line uses the SciPy function `quad` to numerically integrate `f` from 0 to k_f. `quad` returns a tuple containing both the numerical value of the integration and the bounds on that value's error, so we unpack it to get the value we want and call it `integral`.

To determine the value of ϕ_0, we must use a root finding routine because in (3.24), ϕ_0 appears on both sides of the equation. Thus, we subtract all terms to one side of the equality such that is set equal to zero; this is the return value of the `f_phi0` function above. This function will later be passed to SciPy function called `root` that will find when this function equals zero, meaning we have the desired value of ϕ_0.

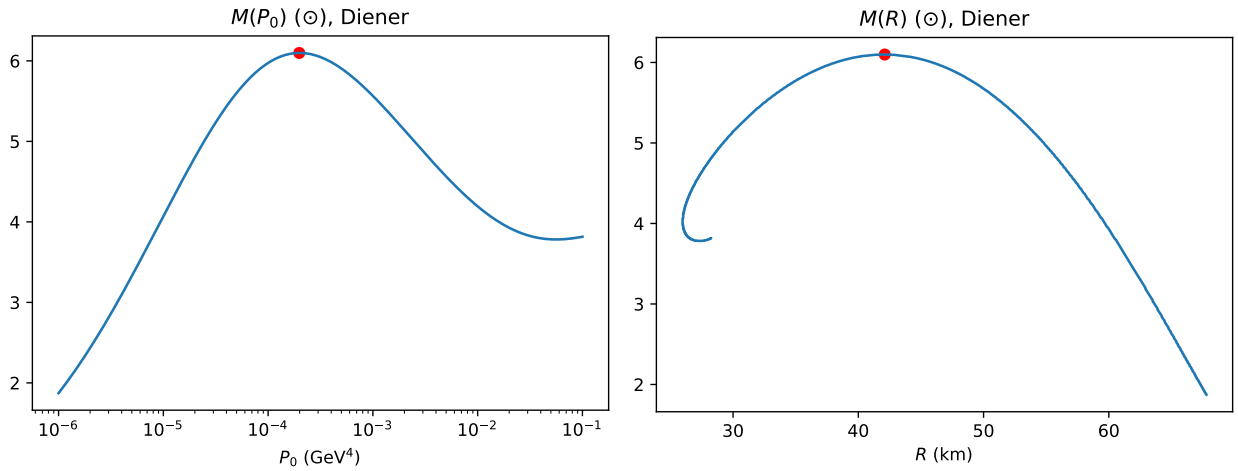

Figure 3.1. The mass-pressure curve (left) and mass-radius curve (right) for the QHD-1 equation of state given in [1].

For the remaining parameters, V_0, P, and ϵ, we simply define functions to determine their values. In the equations for P and ϵ, we again use numerical integration using `quad` to determine the value of each integral, just as with the ϕ_0 equation.

```
kfs = logspace(-6,0,12000,base=10)

energy_densities = []
pressures = []

phi0_guess = 10**(-1)

for kf in kfs:

    res = root(f_phi0, phi0_guess, args=kf)
    phi0 = res.x[0]

    V0 = f_V0(kf)
    eps = f_eps(phi0, V0, kf)
    p = f_P(phi0, V0, kf)

    energy_densities.append(eps)
    pressures.append(p)

    phi0_guess = phi0
```

By the above code block, we now calculate numerical values for P and ϵ. We create an array of k_f values and loop through each. During every iteration, we calculate the root of the ϕ_0 function, which gives us ϕ_0, and we use that value, as well as the value of V_0 to determine the values of P and ϵ. These values are saved and written to file. Notably, our root finding needs an initial guess, so we use the previously determined value of ϕ_0 as the guess for the current iteration, as we know it will be relatively close.

After tabulating values for this QHD-1 equation of state, we can create mass-radius and mass-pressure curves as derived in chapter 2. This produces the results given in Figure 3.1. From these plots and this analysis, we calculate the critical (maximum) mass that this EOS predicts to be $6.1\,\odot$ with a critical radius of 42.1 km.

Importantly, these critical mass and critical radius values are noticeably larger than those given for the SLy family of EOSs. This is because the QHD-I EOS is not considered a "realistic" EOS due to the extensive simplifications that it makes. Furthermore, the solution that we calculated here had the RMF simplifications, on top of the QHD-I simplifications, so this EOS should not be expected to give values that realistically predict neutron stars. This EOS instead was used as a teaching aid for how to effectively apply a theoretical model to numerically calculate an EOS and use that EOS to make predictions about the observable properties of the neutron stars it describes.

Chapter 4
Advanced QHD Parameter Sets

4.1 Introduction

This chapter will explore a more general QHD formalism and will implement two parameter sets: FSUGold and NL3. In this formalism, additional meson fields, in the form of the rho meson field, and higher order coupling terms for the familiar meson fields, the scalar and vector meson fields, will be included. Furthermore, the equation of state will be made more realistic by including constraints for beta-equilibrium, including the addition of two lepton species (electrons and muons).

This chapter will not go into the same detail as chapter 3 for its derivations. These were beyond the scope of the project and not the main intention; instead, it was most important to practice calculating an EOS and to reproduce the results in [1].

4.2 Equations of Motion and RMF Simplifications

In this chapter, we work with a more general Lagrange density for QHD

$$\begin{aligned}\mathscr{L} = \psi\Big[\gamma^\mu\Big(i\partial_\mu - g_v V_\mu - \frac{g_\rho}{2}\boldsymbol{\tau}\cdot\boldsymbol{b}_\mu\Big) - (M - g_s\phi)\Big]\psi \\ + \frac{1}{2}\partial_\mu\partial^\mu\phi - \frac{1}{2}m_s^2\phi^2 - \frac{\kappa}{3!}(g_s\phi)^3 - \frac{\lambda}{4!}(g_s\phi)^4 \\ - \frac{1}{4}V^{\mu\nu}V_{\mu\nu} + \frac{1}{2}m_\omega^2 V^\mu V_\mu + \frac{\zeta}{4!}(g_v^2 V^\mu V_\mu)^2 \\ - \frac{1}{4}\boldsymbol{b}^{\mu\nu}\boldsymbol{b}_{\mu\nu} + \frac{1}{2}m_\rho^2\boldsymbol{b}^\mu\cdot\boldsymbol{b}_\mu + \Lambda_v(g_v^2 V^\mu V_\mu)(g_\rho^2\boldsymbol{b}^\mu\cdot\boldsymbol{b}_\mu),\end{aligned} \tag{4.1}$$

where

$$V_{\mu\nu} \equiv \partial_\mu V_\nu - \partial_\nu V_\mu, \quad \boldsymbol{b}_{\mu\nu} \equiv \partial_\mu\boldsymbol{b}_\nu - \partial_\nu\boldsymbol{b}_\mu.$$

These fields are identical to those in chapter 3, with the addition of the $\boldsymbol{b}$ field and additional coupling constants: ϕ represents the scalar meson field, V_μ the vector meson field, ψ the baryon field, and $\boldsymbol{b} = (b_1^\mu, b_2^\mu, b_3^\mu)$, the three isospin components of the rho meson field. $\boldsymbol{\tau} = (\tau_1, \tau_2, \tau_3)$ is called the *isospin* operator, defined in terms of the Pauli Matrices; see [1, pp. 72-73] for more details.

The additional coupling constants arise from allowing higher order terms in V_μ and ϕ into the Lagrange density. For the scalar field, ϕ, this includes κ and λ, the third and fourth order self coupling constants, respectively. For the vector meson field, ζ is the self coupling constant. Finally, the coupling between the rho meson and vector meson fields is quantified by Λ_v.

Intuitively, including additional meson fields and higher order couplings will make the Lagrangian more accurate, while at the same time more complicated. These extra terms better model the meson exchange and the interactions between the fields, but at the expense of computability.

For this system, we once again introduce the relativistic mean field (RMF) simplifications, where we take all fields to be their average values. These simplifications are

$$\phi \quad \Rightarrow \quad \langle \Phi | \, \phi \, | \Phi \rangle = \langle \phi \rangle \equiv \phi_0, \tag{4.2}$$

$$V^\mu \quad \Rightarrow \quad \langle \Phi | \, V^\mu \, | \Phi \rangle = \langle V^\mu \rangle \equiv \delta^{\mu 0} V_0, \tag{4.3}$$

$$\boldsymbol{b}^\mu \quad \Rightarrow \quad \langle \Phi | \, b_i^\mu \, | \Phi \rangle = \langle b_i^\mu \rangle \equiv \delta^{\mu 0} \delta_{i3} b_0. \tag{4.4}$$

The simplifications for ϕ and V^μ are identical to those in section 3.3. However, the simplifications are more complicated for $\boldsymbol{b}^\mu$. Because $\boldsymbol{b}^\mu$ contains three components, each with four components of their own, we write $\boldsymbol{b}^\mu$ as b_i^μ, where the lower index only takes the values $i \in \{1, 2, 3\}$. By [1, p. 74], only the $i = 3$ component of $\boldsymbol{b}^\mu$ has a non-vanishing expectation value, so we add an extra Kronecker-delta to enforce this condition.

Once again, we also must form the expectation values of the baryon operators $\bar{\psi}$ and ψ with the various other operators in the Lagrange density. These give

$$\bar{\psi}\psi \quad \Rightarrow \quad \langle \Phi | : \psi\psi : | \Phi \rangle = \left\langle \bar{\psi}\psi \right\rangle, \tag{4.5}$$

$$\bar{\psi}\gamma^\mu\psi \quad \Rightarrow \quad \langle \Phi | : \psi\gamma^\mu\psi : | \Phi \rangle = \left\langle \psi\gamma^0\psi \right\rangle, \tag{4.6}$$

$$\bar{\psi}\gamma^\mu\tau_i\psi \quad \Rightarrow \quad \langle \Phi | : \psi\gamma^\mu\tau_i\psi : | \Phi \rangle = \left\langle \psi\gamma^0\tau_3\psi \right\rangle. \tag{4.7}$$

We once again follow the procedure described in chapter 3. We first apply the Euler-Lagrange equations in (3.2), as we did in section 3.2. This yields four equations of motion for the cases of $\varphi \in \left\{\phi_0, V_0, b_0, \bar{\psi}\right\}$. We immediately apply the above RMF simplifications; each kills terms with derivatives and introduces expectation values containing the baryon operators, like those in the above equations. From here, there is once again a rigorous mathematical procedure for calculating the forms of these expectation values, which is beyond the scope of this paper. After substituting the expressions for these expectation values, the three equations of motion for the meson fields reduce to, as shown in [1, p. 79]

$$\phi_0 = \frac{g_s}{m_s^2}\left[\frac{1}{\pi^2}\left(\int_0^{k_p} dk \, \frac{k^2 m^*}{\sqrt{k^2 + m^{*2}}} + \int_0^{k_n} dk \, \frac{k^2 m^*}{\sqrt{k^2 + m^{*2}}}\right) - \frac{\kappa}{2}(g_s\phi_0)^2 - \frac{\lambda}{6}(g_s\phi_0)^3\right] \tag{4.8}$$

$$V_0 = \frac{g_v}{m_s^2}\left[\rho_p + \rho_n - \frac{\zeta}{6}(g_v V_0)^3 - 2\Lambda_v (g_v V_0)(g_\rho b_0)^2\right] \tag{4.9}$$

$$b_0 = \frac{g_\rho}{m_\rho^2}\left[\frac{1}{2}(\rho_p - \rho_n) - 2\Lambda_v (g_v V_0)^2 (g_\rho b_0)\right] \tag{4.10}$$

There is an analogous way to determine equations for ϵ and P from the Lagrange density for this more advanced system as in section 3.3. After doing so, we once again apply the RMF simplifications and determine the values of the expectation values. This gives the forms of ϵ and P in (9.20) in [1, p. 79]. However, as mentioned in the introduction, this EOS takes into account the effects of leptons, too, namely electrons and muons. This requires the inclusion of two additional integrals over the Fermi-momenta of both the electrons and muons, given on [1, p. 92]. These forms of ϵ and P are given below.

$$\begin{aligned}
\epsilon = &+ \frac{1}{2}m_s^2\phi_0^2 + \frac{\kappa}{3!}(g_s\phi_0)^3 + \frac{\lambda}{4!}(g_s\phi_0)^4 - \frac{1}{2}m_\omega^2 V_0^2 - \frac{\zeta}{4!}(g_v V_0)^4 \\
&- \frac{1}{2}m_\rho^2 b_0^2 - \Lambda_v(g_v V_0)^2(g_\rho b_0)^2 + g_v V_0(\rho_n + \rho_p) + \frac{1}{2}(\rho_p - \rho_n) \\
&+ \frac{1}{\pi^2}\left[\int_0^{k_p} dk\, k^2\sqrt{k^2 + m^{*2}} + \int_0^{k_n} dk\, k^2\sqrt{k^2 + m^{*2}}\right] \\
&+ \frac{1}{\pi^2}\left[\int_0^{k_e} dk\, k^2\sqrt{k^2 + m_e^2} + \int_0^{k_\mu} dk\, k^2\sqrt{k^2 + m_\mu^2}\right] \qquad (4.11) \\
P = &- \frac{1}{2}m_s^2\phi_0^2 - \frac{\kappa}{3!}(g_s\phi_0)^3 - \frac{\lambda}{4!}(g_s\phi_0)^4 + \frac{1}{2}m_\omega^2 V_0^2 + \frac{\zeta}{4!}(g_v V_0)^4 \\
&+ \frac{1}{2}m_\rho^2 b_0^2 + \Lambda_v(g_v V_0)^2(g_\rho b_0)^2 \\
&+ \frac{1}{3\pi^2}\left[\int_0^{k_p} dk\, \frac{k^4}{\sqrt{k^2 + m^{*2}}} + \int_0^{k_n} dk\, \frac{k^4}{\sqrt{k^2 + m^{*2}}}\right] \\
&+ \frac{1}{3\pi^2}\left[\int_0^{k_e} dk\, k^2\sqrt{k^2 + m_e^2} + \int_0^{k_\mu} dk\, k^2\sqrt{k^2 + m_\mu^2}\right] \qquad (4.12)
\end{aligned}$$

4.3 Equilibrium Conditions

As described above, this chapter will explore a more realistic equation of state by including additional constraints to enforce beta-equilibrium. This section will describe those constraints and how they fit into the model.

To begin, we define the number density for a species of particle ρ_x and its corresponding Fermi-momenta by the equations

$$\rho_x = \frac{k_x^3}{3\pi^2} \quad \Longleftrightarrow \quad k_x = (3\pi^2\rho_x)^{1/3}. \qquad (4.13)$$

This will be a useful result that will be used constantly throughout section 4.4.

In the chapter 3, our loop variable was k_f; however, in this calculation, we choose to use ρ, the total baryon number density, as the loop variable. Due to the above equation, we can directly relate these two quantities as needed.

At any iteration of the loop variable ρ, the number densities of baryons must stay constant; thus, we have

$$\rho = \rho_n + \rho_p, \qquad (4.14)$$

where ρ_n is the number density of the first species, neutrons, while ρ_p is the number density of the second species, protons. That is, throughout the iteration, the total number density of protons and neutrons must stay constant. This condition comes from the decay of neutrons into protons, electrons, and anti-neutrinos called beta-decay.

There is a reverse process, however, where a proton and electron combine to form a neutron and a neutrino. This process is possible and likely due to the high energies of electrons present in the extreme conditions within a neutron star. These "competing" processes work until *beta-equilibrium* is reached, where, in a given iteration, ρ_p and ρ_n are such that ϵ is minimized. If we define the chemical potential as

$$\mu_x \equiv \frac{\partial \epsilon}{\partial \rho_x}, \tag{4.15}$$

for the xth chemical potential, the condition for beta-equilibrium can therefore be written as

$$\mu_n = \mu_p + \mu_e. \tag{4.16}$$

Using a convenient result in (10.7) on [1, p. 90], we can rewrite this more generally as

$$\sqrt{k_n^2 + m^{*2}} = \sqrt{k_p^2 + m^{*2}} + g_\rho b_0 + \sqrt{k_e^2 + m_e^2}. \tag{4.17}$$

This expression is found by calculating the chemical potentials μ_n, μ_p, and μ_e by differentiating (4.11) with respect to different number densities by (4.15) and substituting into (4.16).

Furthermore, within neutron stars, there is a third reaction in which a muon decays into an electron, a neutrino, and an anti-neutrino. Once the energies are large enough, and muon states have become populated, the muon formation and decay must be in equilibrium; this is quantified by the equality of their chemical potentials

$$\mu_\mu = \mu_e, \quad \mu_e = \sqrt{k_e^2 + m_e^2}, \quad \mu_\mu = \sqrt{k_\mu^2 + m_\mu^2}. \tag{4.18}$$

Our final constraint is charge conservation; because muons and electrons have equal charges, opposite that of protons, this constraint takes the form

$$\rho_p = \rho_e + \rho_\mu \quad \Rightarrow \quad k_p = \left(k_e^3 + k_\mu^3\right)^{1/3}. \tag{4.19}$$

These four constraint equations, (4.14), (4.17), (4.18), and (4.19), in conjunction with the equations of motion in (4.8), (4.9), and (4.10) must be solved concurrently. This will be described in detail below.

4.4 Numerical Generation of the Equation of State

Within this section, important code blocks will be included for reading convenience; however, any line numbers will reference the complete code sample given at the conclusion of this paper in Appendix C.

This section will describe the procedure for numerically calculating the equation of state from the constraint equations in section 4.3 and the equations of motion in section 4.2, and then the following process for computing ϵ and P.

In lines 1-53 of the file, the values of the constants for the two analyzed parameter sets, NL3 and FSUGold, are implemented. This includes the initialization of the coupling constants and the masses of the fields.

```
@njit
def k_from_rho(rho):
    return (3*np.pi**2*rho)**(1/3)

@njit
def rho_from_k(k):
    return k**3/(3*np.pi**2)
```

At this point, we define the two above functions. These are implementations of the equation (and its inverse) present in (4.13). Of note is the `@njit` "decorator" from the Numba library present on both of these functions. Numba is a Python library that has tools for improving the speed of Python code through "just-in-time" (jit) code compilation. In these situations, where the code is simple and repeatable, with predictable datatypes, providing this decorator can make these functions much faster; whenever possible, we applied it to functions that we were writing.

We next had to decide how to solve our set of seven coupled equations efficiently. Specifically, we have the constraint equations: the conservation of baryon number density (4.14), the beta equilibrium condition (4.17), the equality of the muon and electron chemical potentials (4.18), charge equality (4.19), followed by the equations of motion: the ϕ_0 equation (4.8), the V_0 equation (4.9), and the b_0 equation (4.10). At each step in the process, we must solve *all* of these equations at the same time to determine a solution. Fortunately, there are numerous solvers implemented to solve such a system by minimizing a set of residuals; in our case, we chose the SciPy solver titled `least_squares`.

```
def ns_system(vec, rho):
    rho_n, rho_p, rho_e, rho_mu, phi0, V0, b0 = vec

    k_n = k_from_rho(rho_n)
    k_p = k_from_rho(rho_p)
    k_e = k_from_rho(rho_e)
    k_mu = k_from_rho(rho_mu)

    mu_mu = np.sqrt(k_mu**2 + m_mu**2)
    mu_e = np.sqrt(k_e**2 + m_e**2)

    mstar = (M - g_s*phi0)

    f = lambda k: (k**2*mstar)/np.sqrt(k**2 + mstar**2)
    p_int, p_err = quad(f,0,k_p)
    n_int, n_err = quad(f,0,k_n)
```

```
    return np.array([
        # conservation of baryon number density
        rho - (rho_n + rho_p),
        # beta equilibrium condition
        np.sqrt(k_n**2 + mstar**2) - (np.sqrt(k_p**2 + mstar**2) +
           g_rho*b0 + np.sqrt(k_e**2 + m_e**2)),
        # equilibrium of muons and electrons
        mu_mu - mu_e,
        # charge equilibrium
        k_p - (k_e**3 + k_mu**3)**(1/3),
        # phi0 equation
        phi0 -  g_s/(m_s**2) * (1/np.pi**2 * (p_int + n_int) -kappa/2 *
            (g_s * phi0)**2 - lmbda/6 * (g_s * phi0)**3),
        # V0 equation
        V0 - g_v/m_omega**2 * (rho_p + rho_n - zeta/6 * (g_v * V0)**3 -
            2*Lmbda_v*(g_v * V0)*(g_rho * b0)**2),
        # b0 equation
        b0 - g_rho/m_rho**2 * (1/2 * (rho_p - rho_n) - 2*Lmbda_v*(g_v *
            V0)**2*(g_rho * b0))
    ])
```

Therefore, we created a function for the evolution of the system of equations included in section 4.2 and section 4.3 called `ns_system`, shown above. This function was to be passed to `least_squares` as the evolution function. `least_squares` requires a function whose argument is a vector[1] (tuple) containing all the variables whose solution we are looking for, with a return value of an array containing the *residuals* of the equations in the system. The solver then minimizes the system of residuals as a system and returns a vector containing the value of the variables that *caused* that minimum condition, the solution that we are looking for.

Our `ns_system` function, therefore, was written to satisfy these conditions. First, the variable `vec` contains the seven variables whose values we want to determine: $\rho_n, \rho_p, \rho_e, \rho_\mu$, ϕ_0, V_0, and b_0. Importantly, by decision, we decide to determine the number densities (ρ_x) of the different species of particles present, and not the corresponding Fermi-momentas (k_x). From the known information, then calculate the Fermi-momentum of each particle, the chemical potentials μ_μ and μ_e, the reduced mass m^*, and the integrals present in the ϕ_0 equation; once again, the integrals are calculated numerically using the SciPy `quad` function. Then, each of the equations mentioned above are implemented as a residual by subtracting the right-hand side of the equation to find an expression equal to zero. During the minimization, the residual is squared and minimized, and the resulting solution values of the vector are returned by `least_squares`.

After the definition of `ns_system`, we implement another sister function named `ns_system_no_muons`. There comes a point during the evolution when the muon Fermi-momenta, defined by $k_\mu = \sqrt{\mu_\mu^2 - m_\mu^2}$, becomes imaginary; at this point, no more muons are produced,

[1]Our function has two parameters; the first is a vector, and the second is a parameter ρ necessary for determining the rest of the values within this evolution equation. The SciPy solver (which will be shown later) allows for additional arguments to be passed by using the `args` keyword and passing a list of arguments in the order they should be passed; this is how the value of ρ is injected to `ns_system` throughout its calls.

and the evolution should no longer include muons at all. In this function, the parameter `vec` does not include ρ_μ and does not contain the residual $\mu_\mu = \mu_e$, the equivalence of chemical potentials between electrons and muons. Once k_μ becomes close enough to zero, we begin evolving this function instead; this switch will be specifically outlined below.

```
def eps(vec):
    if len(vec) == 6:
        rho_n, rho_p, rho_e, phi0, V0, b0 = vec
        rho_mu = 0
    else:
        rho_n, rho_p, rho_e, rho_mu, phi0, V0, b0 = vec

    # omitted ...
```

After this point, we define both the ϵ and P functions; the beginning of the ϵ function is shown above, while the P function is omitted, as it has the same initial form and simply performs some different calculations in its body. Importantly, both functions are equipped to handle both the muon and non-muon cases; this is shown by the lines above. If the vector only contains six variables, it means that muons are not currently being produced, so the muon number density ρ_μ should be zero; otherwise, we should unpack the vector as expected. In the body of the function, in the "omitted" section, both functions carry out numerical integrations and simple numerical calculations to return the values of ϵ and P.

```
rho_exp = np.linspace(-1, -5, 5000)
x0 = [.2, .2, .2, .2, .2, .2, .2]

n = -1
for i,exp in enumerate(rho_exp):
    rho = 10**(exp)

    if x0[3] <= 1e-12:
        n = i
        break

    sol = least_squares(ns_system, x0, args=[rho], method='lm')

    x0 = sol.x
    output.write(f'{eps(x0):.16e},␣{P(x0):.16e}\n')

x0 = x0[0], x0[1], x0[2], x0[4], x0[5], x0[6]

for i in range(n, len(rho_exp)):
    # no muon section ...
```

At this point, we are ready to calculate the equation of state. We begin with an initial guess `x0`; by trial and error, we found an adequate value to be .2 for all variables. Then, we begin looping through our exponents[2], beginning at -1. In each iteration of the loop, we

[2]To easily work on a logarithmic range, we simply iterate over an exponent linearly, and each point, just raise 10 to that exponent.

first calculated the ρ value, our total baryon number density, using this exponent. Towards the bottom, we create `sol`, the solution to our `least_squares` minimization, with our ρ value as an argument and the `x0` value as an initial guess. Then, we extract the solution, set it as our initial guess for the next iteration, and write the ϵ and P values to file for the current solution.

In the middle of the function, we have a break condition; this is the point at which the muons are no longer produced. `x0[3]` is the current ρ_μ value, and once that drops below a threshold, in this case 10^{-12}, we break from the current `for` loop. At this point, we form a new initial condition, without the previous value of ρ_μ (`x0[3]` is no longer present) and then begin iterating through the remainder of the exponent values. The omitted section, "no muon section," performs the same procedure as described above, however with the evolution function `ns_system_no_muons`. After this second loop is completed, the equation of state will be tabulated in a file.

At this point, we solve the TOV equations using the NL3 parameter set in order to compare to the plot given on [1, p. 117]. This plot is for a neutron star in beta-equilibrium with no crustal effects included. Our reproduction is given below in Figure 4.1.

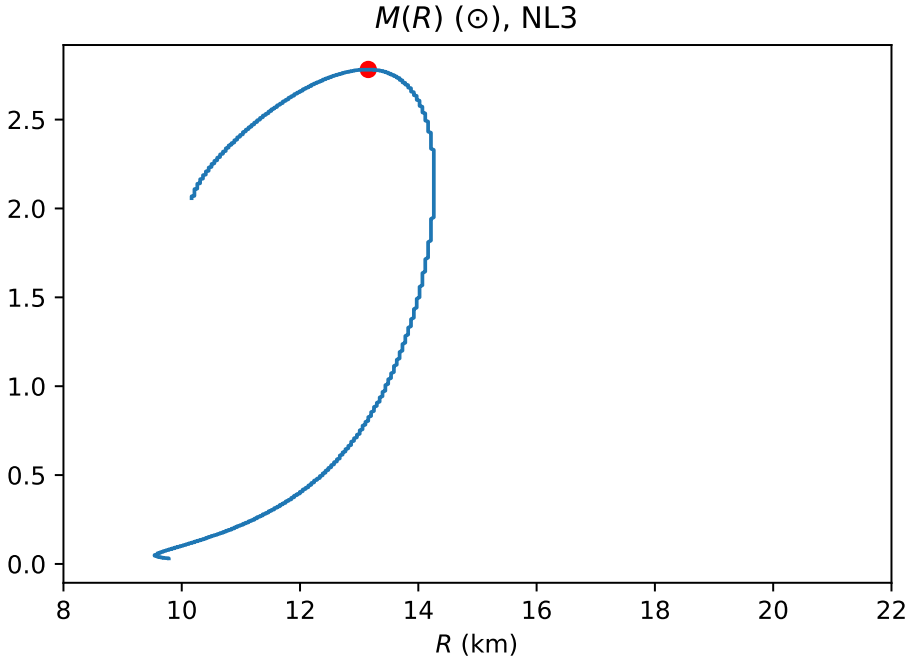

Figure 4.1. The $M(R)$ plot for the NL3 parameter set for comparison to Fig. A.3 on [1, p. 117]

Our plot matches up perfectly with the one given in [1], demonstrating the validity of our code and calculation. Originally, our goal was to continue by calculating crustal effects and adding them to this equation of state, which is valid for the inner core. We then would have compared to the plots given on [1, p. 102], both for NL3 and FSUGold.

While the implementation of the EOSs for NL3 and FSUGold are not considered complete, as they simply include descriptions of the inner crust, we still calculated the critical

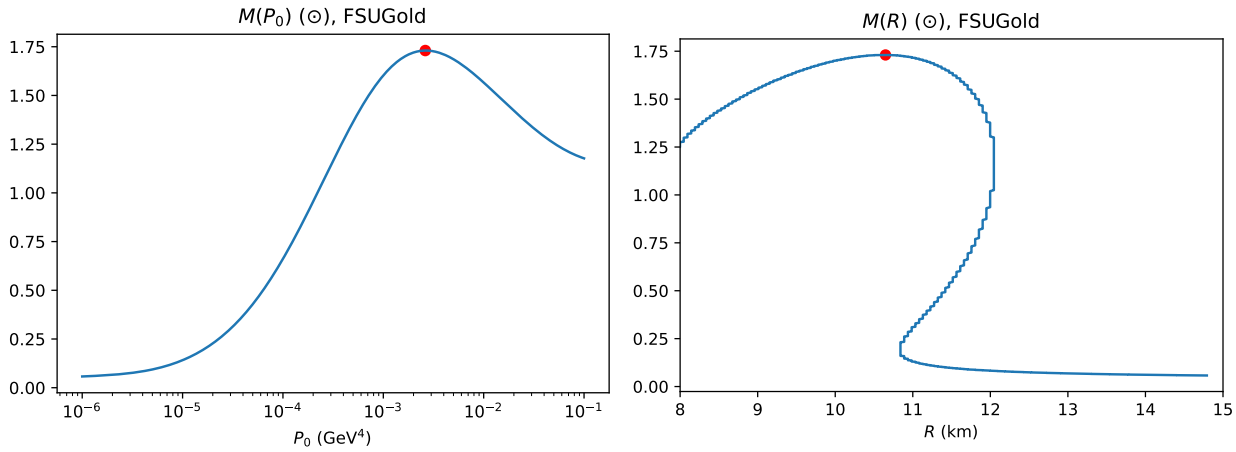

Figure 4.2. The mass-pressure curve (left) and mass-radius curve (right) for the FSUGold equation of state given in [1].

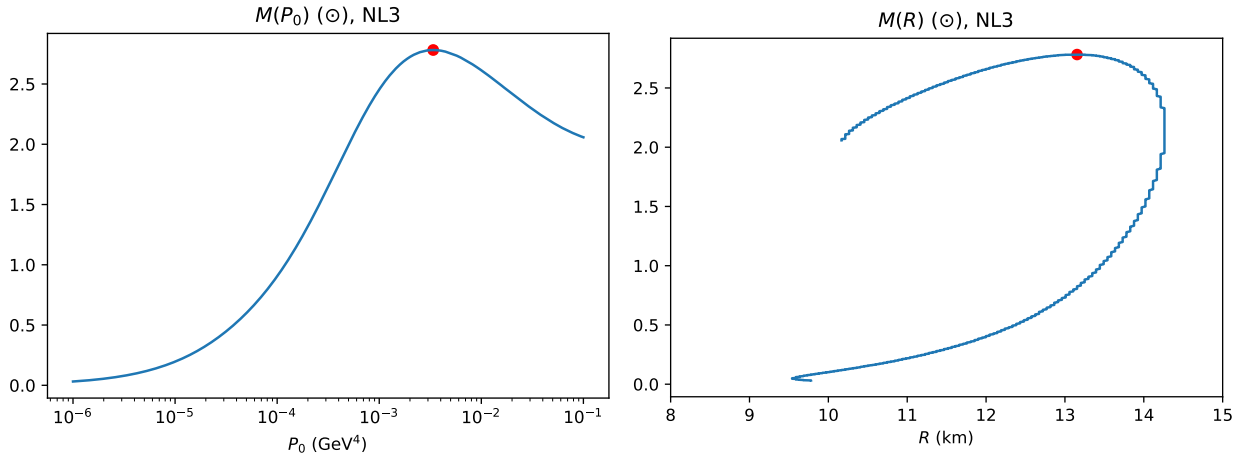

Figure 4.3. The mass-pressure curve (left) and mass-radius curve (right) for the NL3 equation of state given in [1].

mass and radius from the TOV equations to present and to compare the results to those in the QHD-I EOS from chapter 3. From Figure 4.2, we get predictions for FSUGold: $M_{\max} = 1.73\,\odot$ and $R_{\max} = 10.6\,\text{km}$. Similarly, Figure 4.3 gives the following predictions for NL3: $M_{\max} = 2.78\,\odot$ and $R_{\max} = 13.2\,\text{km}$. These values are much closer to those of the realistic, analytic EOSs of the SLy family given in Figure 2.3. This is a testament to the additional steps taken within their derivation: the inclusion of the rho-meson field, and the additional constraints present to satisfy beta-equilibrium. These steps make the calculated equations of state much more realistic. In comparision, the predictions that QHD-I makes ($M_{\max} = 6.1\,\odot$ and $R_{\max} = 42.1\,\text{km}$) seem quite unrealistic.

Appendix A
Code: TOV Implementation

```
# Joe Nyhan, 15 July 2021
# File for computing static solutions for fermionic stars via the
   TOV model.

import numpy as np
import pandas as pd
from scipy.integrate import solve_ivp
from scipy.interpolate import interp1d
import matplotlib.pyplot as plt

# EOS
eos_UR        = 0
eos_polytrope = 0

eos_SLy       = 1
eos_FPS       = 0

eos_BSk19     = 0
eos_BSk20     = 0
eos_BSk21     = 0

if eos_UR:
    eos = "UR"
    Gamma = 1.3
if eos_polytrope:
    eos = "polytrope"
    Gamma = 2
    K = 100
if eos_SLy:
    eos = "SLy"
if eos_FPS:
    eos = "FPS"
if eos_BSk19:
    eos = "BSk19"
if eos_BSk20:
```

```
35      eos = "BSk20"
36  if eos_BSk21:
37      eos = "BSk21"
38
39  # MODES
40  make_static_solution = 0
41  p0_analysis = 1
42
43  # PARAMETERS
44
45  if make_static_solution:
46      p0 = 1e-4
47
48  if p0_analysis:
49      pmin = 1e-6
50      pmax = 1e-1
51      NUM_POINTS = 1000
52      p0_vals = np.logspace(round(np.log10(pmin)),round(np.log10(pmax)
            ),NUM_POINTS, base=10.0)
53
54  r0 = 0.000001
55  rmax = 200
56  dr = 0.02
57
58
59  p_tol = 1e-11
60  r_vals = np.arange(r0,rmax,dr)
61
62  m0 = 0
63
64  if make_static_solution: path = "../input"
65  elif p0_analysis: path = "./p0_analysis"
66
67  # FUNCTIONS
68
69  vals_path = ""
70  if eos_UR or eos_polytrope:
71      pass
72  else:
73      vals_path = f"0-{eos}_vals.vals"
74
75      df = pd.read_csv(vals_path)
76      interp_rhos = df.iloc[:,0].to_numpy()
77      interp_ps = df.iloc[:,1].to_numpy()
78
79
80  rho_from_P_interp = interp1d(interp_ps, interp_rhos, fill_value="
```

```
   extrapolate")
81
82 def rho_from_P(p):
83     if eos_UR:
84         return p/(Gamma-1)
85     elif eos_polytrope:
86         return (p/K)**(1/Gamma)
87     else:
88         return rho_from_P_interp(p)
89
90
91 def event(r,y):
92     m, p = y
93
94     return p - p_tol
95
96 event.terminal = True
97 event.direction = 0
98
99 def f_onefluid(r,y):
100     m, p = y
101
102     rho = rho_from_P(p)
103
104     N = 1-2*m/r
105
106     fm = 4*np.pi*r**2*rho
107     frho = (-(4*np.pi*r**3*p + m)*(rho+p)/(r**2*N))
108
109     return fm, frho
110
111 # ONE STATIC SOLUTIION
112
113 if make_static_solution:
114     if eos_polytrope:
115         P_path = f"{path}/polytrope_K{K:.1f}_gamma{Gamma:.1f}_p{p0
               :.8f}_dr{dr:.3f}_P.txt"
116         rho_path = f"{path}/polytrope_K{K:.1f}_gamma{Gamma:.1f}_p{p0
               :.8f}_dr{dr:.3f}_rho.txt"
117
118     # if eos_SLy:
119     else:
120         P_path = f"{path}/{eos}_p{p0:.8f}_dr{dr:.3f}_P.txt"
121         rho_path = f"{path}/{eos}_p{p0:.8f}_dr{dr:.3f}_rho.txt"
122
123     P_out = open(P_path,"w")
124     rho_out = open(rho_path,"w")
```

```
125
126     sol = solve_ivp(f_onefluid,[r0,rmax], [m0,p0], method='RK45',
           atol = 1e-12, rtol = 1e-12, t_eval=r_vals, events=event)
127
128     m, p = sol.y
129
130     for i in range(len(r_vals)):
131         if i < len(p):
132             P_out.write(f"{p[i]:.16e}\n")
133             rho_out.write(f"{rho_from_P(p[i]):.16e}\n")
134         else:
135             P_out.write(f"{0:.16e}\n")
136             rho_out.write(f"{0:.16e}\n")
137
138     P_out.close()
139     rho_out.close()
140
141     print(f"COMPLETED.")
142
143
144 # P0 ANALYSIS
145
146 if p0_analysis:
147     output = open(f"{path}/{eos},p{pmin:.3e}-p{pmax:.3e}.vals","w")
148
149     print(f"Number␣of␣points:{len(p0_vals)};␣for␣EoS:␣{eos}")
150
151     def evolution(p0_vals):
152         for i in range(len(p0_vals)):
153
154             p0 = p0_vals[i]
155             sol = solve_ivp(f_onefluid,[r0,rmax], [m0,p0], method='
                   RK45', atol = 1e-12, rtol = 1e-12, t_eval=r_vals,
                   events=event)
156             M = sol.y[0][-1]
157             R = sol.t[-1]
158
159             if M < 0: print(f"M␣<0"); break
160
161             output.write(f"{p0:.6e},{M:.6e},{R:.6e}\n")
162
163             if i % 10 == 0: print(f"Timestep␣{i:3d}␣reached,␣{M=}.")
164
165     evolution(p0_vals)
166
167     output.close()
```

Appendix B
Code: QHD-I Implementation

```
# Joe Nyhan, 18 January 2021
#%%
from numpy import sqrt, pi, logspace
import matplotlib.pyplot as plt
from scipy.integrate import quad
from scipy.optimize import root

gamma = 2

# masses (in GeV)
m_s = .520
m_omega = .783
M = .939

# other constants
g_s = sqrt(109.6)
g_v = sqrt(190.4)


def f_phi0(phi0, kf):
    """see Diener p. 66, eq 8.21a"""
    mstar = M - g_s * phi0
    def f(k):
        return k**2 * mstar/sqrt(k**2 + mstar**2)

    integral, err = quad(f,0,kf)

    return phi0 - (g_s/m_s**2) * (gamma/(2*pi**2)) * integral

def f_V0(kf):
    """see Diener p. 66, eq 8.21b"""
    rho = 2 * (gamma/(6*pi**2))*kf**3

    return (g_v/m_omega**2) * rho

```

```
36 def f_eps(phi0, V0, kf):
37     """see Diener p. 66, eq 8.24a"""
38     mstar = M - g_s * phi0
39     def f(k):
40         return k**2 * sqrt(k**2 + mstar**2)
41     integral, err = quad(f,0,kf)
42 
43     return 1/2 * m_s**2 * phi0**2 + 1/2 * m_omega**2 * V0**2 + (
           gamma/(2*pi**2)) * integral
44 
45 def f_P(phi0, V0, kf):
46     """see Diener p. 66, eq 8.24b"""
47     mstar = M - g_s * phi0
48 
49     def f(k):
50         return k**4 / sqrt(k**2 + mstar**2)
51     integral, err = quad(f,0,kf)
52 
53     return -1/2 * m_s**2 * phi0**2 + 1/2 * m_omega**2 * V0**2 +
           (1/3) * (gamma/(2*pi**2)) * integral
54 
55 kfs = logspace(-6,0,12000,base=10)
56 
57 energy_densities = []
58 pressures = []
59 
60 phi0_guess = 10**(-1)
61 
62 for kf in kfs:
63 
64     res = root(f_phi0, phi0_guess, args=kf)
65     phi0 = res.x[0]
66 
67     V0 = f_V0(kf)
68     eps = f_eps(phi0, V0, kf)
69     p = f_P(phi0, V0, kf)
70 
71     energy_densities.append(eps)
72     pressures.append(p)
73 
74     phi0_guess = phi0
```

Appendix C
Code: Advanced QHD Implementation

```
# Joe Nyhan, 22 February 2022
# Calculates a table of values for EoS from the advanced QHD
   parameter sets

import numpy as np
from scipy.integrate import quad
from scipy.optimize import least_squares
from numba import njit
import sys
np.set_printoptions(threshold=sys.maxsize)

# ========== PARAMETER SET

NL3      = 1
FSU_GOLD = 0

# ========== PARMETERS

# masses (GeV)
M    = .939
m_e  = 511e-6
m_mu = .1057
rho0 = .01

if NL3:
    # parameters
    g_s     = np.sqrt(104.3871)
    g_v     = np.sqrt(165.5854)
    g_rho   = np.sqrt(79.6000)
    kappa   = 3.8599e-3 #GeV
    lmbda   = -.01591
    zeta    = 0.00
    Lmbda_v = 0.00

    # masses (GeV)
```

```
35      m_s     = .5081940
36      m_omega = .7825
37      m_rho   = .763
38
39  elif FSU_GOLD:
40      # parameters
41      g_s     = np.sqrt(112.1996)
42      g_v     = np.sqrt(204.5469)
43      g_rho   = np.sqrt(138.4701)
44      kappa   = 1.4203e-3 #GeV
45      lmbda   = 0.0238
46      zeta    = 0.06
47      Lmbda_v = 0.03
48
49      # masses (GeV)
50      M       = .939
51      m_s     = .4915000
52      m_omega = .783
53      m_rho   = .763
54
55  # ========== EQUATIONS OF MOTION
56
57  @njit
58  def k_from_rho(rho):
59      return (3*np.pi**2*rho)**(1/3)
60
61  @njit
62  def rho_from_k(k):
63      return k**3/(3*np.pi**2)
64
65  def ns_system(vec, rho):
66      rho_n, rho_p, rho_e, rho_mu, phi0, V0, b0 = vec
67
68      k_n = k_from_rho(rho_n)
69      k_p = k_from_rho(rho_p)
70      k_e = k_from_rho(rho_e)
71      k_mu = k_from_rho(rho_mu)
72
73      mu_mu = np.sqrt(k_mu**2 + m_mu**2)
74      mu_e = np.sqrt(k_e**2 + m_e**2)
75
76      mstar = (M - g_s*phi0)
77
78      f = lambda k: (k**2*mstar)/np.sqrt(k**2 + mstar**2)
79      p_int, p_err = quad(f,0,k_p)
80      n_int, n_err = quad(f,0,k_n)
81
```

```
82      return np.array([
83          # conservation of baryon number density
84          rho - (rho_n + rho_p),
85          # beta equilibrium condition
86          np.sqrt(k_n**2 + mstar**2) - (np.sqrt(k_p**2 + mstar**2) +
               g_rho*b0 + np.sqrt(k_e**2 + m_e**2)),
87          # equilibrium of muons and electrons
88          mu_mu - mu_e,
89          # charge equilibrium
90          k_p - (k_e**3 + k_mu**3)**(1/3),
91          # phi0 equation
92          phi0 -  g_s/(m_s**2) * (1/np.pi**2 * (p_int + n_int) -kappa
               /2 * (g_s * phi0)**2 - lmbda/6 * (g_s * phi0)**3),
93          # V0 equation
94          V0 - g_v/m_omega**2 * (rho_p + rho_n - zeta/6 * (g_v * V0)
               **3 - 2*Lmbda_v*(g_v * V0)*(g_rho * b0)**2),
95          # b0 equation
96          b0 - g_rho/m_rho**2 * (1/2 * (rho_p - rho_n) - 2*Lmbda_v*(
               g_v * V0)**2*(g_rho * b0))
97      ])
98
99  def ns_system_no_muons(vec, rho):
100     rho_n, rho_p, rho_e, phi0, V0, b0 = vec
101
102     k_n = k_from_rho(rho_n)
103     k_p = k_from_rho(rho_p)
104     k_e = k_from_rho(rho_e)
105
106     mstar = (M - g_s*phi0)
107
108     f = lambda k: (k**2*mstar)/np.sqrt(k**2 + mstar**2)
109     p_int, p_err = quad(f,0,k_p)
110     n_int, n_err = quad(f,0,k_n)
111
112     return np.array([
113         rho - (rho_n + rho_p), # conservation of baryon number
               density
114         np.sqrt(k_n**2 + mstar**2) - (np.sqrt(k_p**2 + mstar**2) +
               g_rho*b0 + np.sqrt(k_e**2 + m_e**2)), # beta equilibrium
               condition
115         k_p - k_e, # charge equilibrium
116         phi0 -  g_s/(m_s**2) * (1/np.pi**2 * (p_int + n_int) -kappa
               /2 * (g_s * phi0)**2 - lmbda/6 * (g_s * phi0)**3), # phi0
                equation
117         V0 - g_v/m_omega**2 * (rho_p + rho_n - zeta/6 * (g_v * V0)
               **3 - 2*Lmbda_v*(g_v * V0)*(g_rho * b0)**2),
118         b0 - g_rho/m_rho**2 * (1/2 * (rho_p - rho_n) - 2*Lmbda_v*(
```

```
            g_v * V0)**2*(g_rho * b0))
    ])

def eps(vec):
    if len(vec) == 6:
        rho_n, rho_p, rho_e, phi0, V0, b0 = vec
        rho_mu = 0
    else:
        rho_n, rho_p, rho_e, rho_mu, phi0, V0, b0 = vec

    k_n = k_from_rho(rho_n)
    k_p = k_from_rho(rho_p)
    k_e = k_from_rho(rho_e)
    k_mu = k_from_rho(rho_mu)

    mstar = (M - g_s*phi0)

    first_line = 1/2 * (m_s**2 * phi0**2) + kappa/np.math.factorial
       (3) * (g_s*phi0)**3 + lmbda/np.math.factorial(4) * (g_s*phi0)
       **4 - 1/2 *m_omega**2*V0**2 - zeta/np.math.factorial(4) * (
       g_v*V0)**4
    second_line = - 1/2 * m_rho**2*b0**2 - Lmbda_v* (g_v*V0)**2*(
       g_rho*b0)**2 + g_v*V0*(rho_n + rho_p) + 1/2 * g_rho*b0*(rho_p
       -rho_n)

    integrand = lambda k, m: k**2 * np.sqrt(k**2 + m**2)

    first_integral, first_err = quad(integrand, 0,k_p, args=mstar)
    second_integral, second_err = quad(integrand, 0,k_n, args=mstar)
    third_integral, third_err = quad(integrand, 0,k_e, args=m_e)
    fourth_integral, fourth_err = quad(integrand, 0,k_mu, args=m_mu)

    integrals = 1/np.pi**2 * (first_integral + second_integral +
       third_integral + fourth_integral)

    return first_line + second_line + integrals

def P(vec):
    if len(vec) == 6:
        rho_n, rho_p, rho_e, phi0, V0, b0 = vec
        rho_mu = 0
    else:
        rho_n, rho_p, rho_e, rho_mu, phi0, V0, b0 = vec

    k_n = k_from_rho(rho_n)
    k_p = k_from_rho(rho_p)
    k_e = k_from_rho(rho_e)
```

```
159     k_mu = k_from_rho(rho_mu)
160
161     mstar = (M - g_s*phi0)
162
163     first_line = - 1/2 * (m_s**2 * phi0**2) - kappa/np.math.
            factorial(3) * (g_s*phi0)**3 - lmbda/np.math.factorial(4) * (
            g_s*phi0)**4 + 1/2 *m_omega**2*V0**2 + zeta/np.math.factorial
            (4) * (g_v*V0)**4
164     second_line = + 1/2 * m_rho**2*b0**2 + Lmbda_v* (g_v*V0)**2*(
            g_rho*b0)**2
165
166     integrand = lambda k,m: k**4/np.sqrt(k**2 + m**2)
167     integrand_leptons = lambda k, m: k**2 * np.sqrt(k**2 + m**2)
168
169     first_integral, first_err = quad(integrand, 0,k_p, args=mstar)
170     second_integral, second_err = quad(integrand, 0,k_n, args=mstar)
171     third_integral, third_err = quad(integrand_leptons, 0,k_e, args=
            m_e)
172     fourth_integral, fourth_err = quad(integrand_leptons, 0,k_mu,
            args=m_mu)
173
174     integrals = 1/(3*np.pi**2) * (first_integral + second_integral +
             third_integral + fourth_integral)
175
176     return first_line + second_line + integrals
177
178
179 upper = -1
180 lower = -5
181 density = 1000
182 rho_exp = np.linspace(upper,lower,(upper-lower)*density)
183
184 output = open('./0-NL3_vals.vals', 'w')
185
186 x0 = [.2, .2, .2, .2, .2, .2, .2]
187
188 n = -1
189 for i, exp in enumerate(rho_exp):
190
191     if i % 100 == 0: print(i)
192     rho = 10**(exp)
193
194     if x0[3] <= 1e-12:
195         n = i
196         break
197
198     sol = least_squares(ns_system, x0, args=[rho], method='lm')
```

```
199
200     x0 = sol.x
201     output.write(f'{eps(x0):.16e},␣{P(x0):.16e}\n')
202
203 x0 = x0[0], x0[1], x0[2], x0[4], x0[5], x0[6]
204
205 for i in range(n, len(rho_exp)):
206
207     if i % 100 == 0: print(i)
208
209     exp = rho_exp[i]
210     rho = 10**(exp)
211     sol = least_squares(ns_system_no_muons, x0, args=[rho], method='
            lm')
212
213     x0 = sol.x
214     output.write(f'{eps(x0):.16e},␣{P(x0):.16e}\n')
215
216 output.close()
```

Bibliography

[1] Jacobus Petrus William Diener. "Relativistic mean-field theory applied to the study of neutron star properties". PhD thesis. Stellenbosch University, 2008.

[2] Norman K. Glendenning. *Compact Stars*. Springer, 1997.

[3] P. Haensel and A. Y. Potekhin. "Analytical representations of unified equations of state of neutron-star matter". In: *Astronomy & Astrophysics* 428.1 (Nov. 2004), pp. 191–197. ISSN: 1432-0746. DOI: 10.1051/0004-6361:20041722. URL: http://dx.doi.org/10.1051/0004-6361:20041722.

[4] "Neutron Stars". In: *COSMOS - The SAO Encyclopedia of Astronomy*. URL: https://astronomy.swin.edu.au/cosmos/n/neutron+star#:~:text=Neutrons%20stars%20are%20extreme%20objects,weigh%20around%20a%20billion%20tonnes..

[5] A. Y. Potekhin et al. "Analytical representations of unified equations of state for neutron-star matter". In: *Astronomy & Astrophysics* 560 (Dec. 2013), A48. ISSN: 1432-0746. DOI: 10.1051/0004-6361/201321697. URL: http://dx.doi.org/10.1051/0004-6361/201321697.