Neutron Stars (TOV equation)

Matteo Zandi

11 October 2024

What is a neutron star?

A neutron star is the remnant of a star with initial mass greater than 8 $M_\odot$.

What is a neutron star?

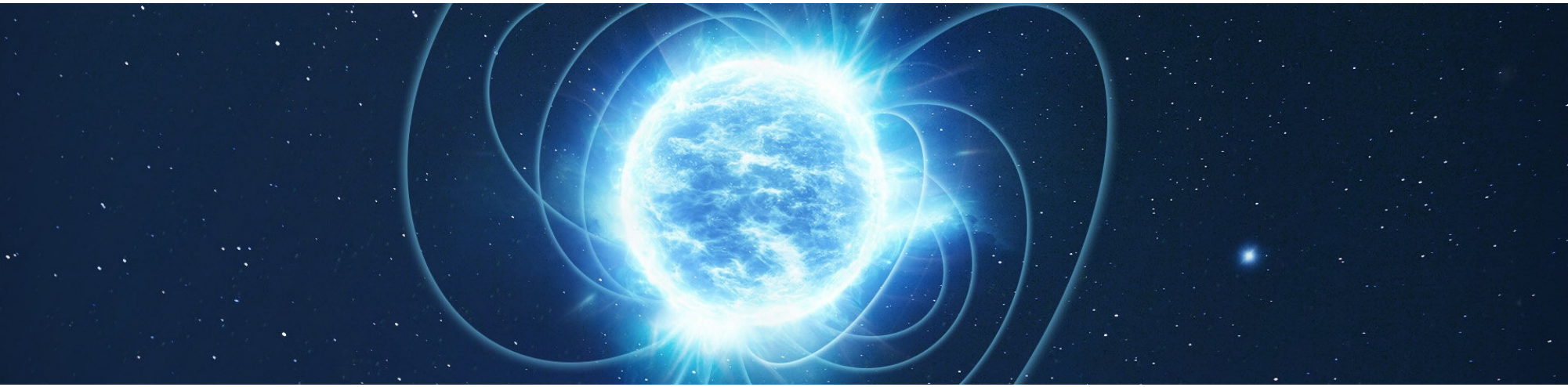
A neutron star is the remnant of a star with initial mass greater than 8 $M_\odot$. After the fusion of all heavier elements up to iron, the outer layers are expelled in a supernova explosion and, if the remaining core has a mass of about 2 to 3 $M_\odot$, it becomes a neutron star.

What is a neutron star?

A neutron star is the remnant of a star with initial mass greater than 8 $M_\odot$. After the fusion of all heavier elements up to iron, the outer layers are expelled in a supernova explosion and, if the remaining core has a mass of about 2 to 3 $M_\odot$, it becomes a neutron star. It is composed of degenerate neutron matter, with a radius of about 10 kilometers.

What is a neutron star?

A neutron star is the remnant of a star with initial mass greater than 8 $M_\odot$. After the fusion of all heavier elements up to iron, the outer layers are expelled in a supernova explosion and, if the remaining core has a mass of about 2 to 3 $M_\odot$, it becomes a neutron star. It is composed of degenerate neutron matter, with a radius of about 10 kilometers.



© ESA

Therefore, we need a model to describe the matter content of a neutron star and a framework to get the equations to solve. The results will be the radius and the mass of the neutron star.

The TOV equation

The framework is general relativity. Consider a spherical symmetric astrophysical object such as a neutron star. Its radius and its mass can be determined by solving a system of ordinary differential equations composed by

The TOV equation

The framework is general relativity. Consider a spherical symmetric astrophysical object such as a neutron star. Its radius and its mass can be determined by solving a system of ordinary differential equations composed by

$$\begin{cases} \epsilon = \epsilon(p) \\ \\ \\ \end{cases}$$

The TOV equation

The framework is general relativity. Consider a spherical symmetric astrophysical object such as a neutron star. Its radius and its mass can be determined by solving a system of ordinary differential equations composed by

$$\begin{cases} \epsilon = \epsilon(p) \\ \frac{dm}{dr} = \frac{4\pi r^2 \epsilon(r)}{c^2} \end{cases}$$

The TOV equation

The framework is general relativity. Consider a spherical symmetric astrophysical object such as a neutron star. Its radius and its mass can be determined by solving a system of ordinary differential equations composed by

$$\begin{cases} \epsilon = \epsilon(p) \\ \frac{dm}{dr} = \frac{4\pi r^2 \epsilon(r)}{c^2} \\ \frac{dp}{dr} = \frac{G\epsilon(r)m(r)}{c^2 r^2}\left(1+\frac{p(r)}{\epsilon(r)}\right)\left(1+\frac{4\pi r^3 p(r)}{m(r)c^2}\right)\left(1-\frac{2Gm(r)}{c^2 r}\right) \end{cases}, \quad (1)$$

The TOV equation

The framework is general relativity. Consider a spherical symmetric astrophysical object such as a neutron star. Its radius and its mass can be determined by solving a system of ordinary differential equations composed by

$$\begin{cases} \epsilon = \epsilon(p) \\ \frac{dm}{dr} = \frac{4\pi r^2 \epsilon(r)}{c^2} \\ \frac{dp}{dr} = \frac{G\epsilon(r)m(r)}{c^2 r^2}\left(1+\frac{p(r)}{\epsilon(r)}\right)\left(1+\frac{4\pi r^3 p(r)}{m(r)c^2}\right)\left(1-\frac{2Gm(r)}{c^2 r}\right) \end{cases}, \quad (1)$$

with boundary conditions

The TOV equation

The framework is general relativity. Consider a spherical symmetric astrophysical object such as a neutron star. Its radius and its mass can be determined by solving a system of ordinary differential equations composed by

$$\begin{cases} \epsilon = \epsilon(p) \\ \frac{dm}{dr} = \frac{4\pi r^2 \epsilon(r)}{c^2} \\ \frac{dp}{dr} = \frac{G\epsilon(r)m(r)}{c^2 r^2}\left(1+\frac{p(r)}{\epsilon(r)}\right)\left(1+\frac{4\pi r^3 p(r)}{m(r)c^2}\right)\left(1-\frac{2Gm(r)}{c^2 r}\right) \end{cases} , \quad (1)$$

with boundary conditions

$$p(r = R_{object}) = 0 \ ,$$

The TOV equation

The framework is general relativity. Consider a spherical symmetric astrophysical object such as a neutron star. Its radius and its mass can be determined by solving a system of ordinary differential equations composed by

$$\begin{cases} \epsilon = \epsilon(p) \\ \frac{dm}{dr} = \frac{4\pi r^2 \epsilon(r)}{c^2} \\ \frac{dp}{dr} = \frac{G\epsilon(r)m(r)}{c^2 r^2}\left(1+\frac{p(r)}{\epsilon(r)}\right)\left(1+\frac{4\pi r^3 p(r)}{m(r)c^2}\right)\left(1-\frac{2Gm(r)}{c^2 r}\right) \end{cases} , \quad (1)$$

with boundary conditions

$$p(r = R_{object}) = 0 \ , \quad p(r = 0) = p_0 \ ,$$

The TOV equation

The framework is general relativity. Consider a spherical symmetric astrophysical object such as a neutron star. Its radius and its mass can be determined by solving a system of ordinary differential equations composed by

$$\begin{cases} \epsilon = \epsilon(p) \\ \frac{dm}{dr} = \frac{4\pi r^2 \epsilon(r)}{c^2} \\ \frac{dp}{dr} = \frac{G\epsilon(r)m(r)}{c^2 r^2}\left(1+\frac{p(r)}{\epsilon(r)}\right)\left(1+\frac{4\pi r^3 p(r)}{m(r)c^2}\right)\left(1-\frac{2Gm(r)}{c^2 r}\right) \end{cases} , \quad (1)$$

with boundary conditions

$$p(r = R_{object}) = 0 \ , \quad p(r = 0) = p_0 \ , \quad m(r = 0) = 0 \ .$$

Solver.py

So, one parameter of the model is the initial central pressure p_0.
Our goal is to numerically implement

Solver.py

So, one parameter of the model is the initial central pressure p_0. Our goal is to numerically implement

- an equation of state;

Solver.py

So, one parameter of the model is the initial central pressure p_0.
Our goal is to numerically implement

- an equation of state;
- the system of ordinary differential equation;

Solver.py

So, one parameter of the model is the initial central pressure p_0.
Our goal is to numerically implement

- an equation of state;
- the system of ordinary differential equation;
- numerical integration, which results into the whole radius evolution of mass and pressure;

Solver.py

So, one parameter of the model is the initial central pressure p_0.
Our goal is to numerically implement

- an equation of state;
- the system of ordinary differential equation;
- numerical integration, which results into the whole radius evolution of mass and pressure;
- the mass and the radius of the neutron star.

Solver.py

So, one parameter of the model is the initial central pressure p_0.
Our goal is to numerically implement

- an equation of state;
- the system of ordinary differential equation;
- numerical integration, which results into the whole radius evolution of mass and pressure;
- the mass and the radius of the neutron star.

It is achieved by means of four classes.

EquationOfState

Class to represent an equation of state, i.e. energy density as function of pressure.

EquationOfState

Class to represent an equation of state, i.e. energy density as function of pressure.

The main methods are

```
def load_from_file(file_path):
    ...
    data = np.loadtxt(file_path, delimiter=",", skiprows=1)
    self.number_densities = np.append(self.number_densities, data[:, 0])
    self.pressures = np.append(self.pressures, data[:, 1])
    self.energy_densities = np.append(self.energy_densities, data[:, 2])

def interpolate():
    ...
    eos = CubicSpline(self.pressures, self.energy_densities)
    return eos
```

TOVSystem

Class to represent the system of ordinary differential equations.

TOVSystem

Class to represent the system of ordinary differential equations.

The main methods are

```
def dmdr(r, p, m):
    ...
    e = self.eos(p)
    dmdr = 4 * np.pi * r**2 * e / c**2
    return dmdr

def dpdr(r, p, m):
    ...
    e = self.eos(p)

    if self.relativity_corrections:
        first_term = G * e * m / c**2 / r**2
        second_term = 1 + p / e
        third_term = 1 + 4 * np.pi * r**3 * p / m / c**2
        fourth_term = (1 - 2 * G * m / c**2 / r) ** (-1)
        dpdr = -first_term * second_term * third_term * fourth_term
    else:
        dpdr = -G / c**2 / r**2 * m * e

    return dpdr
```

SolverTOVSinglePressure

Class to integrate numerically the system of 2 ordinary differential equations (mass and pressure equations) given a single initial central pressure. Implement the 4th-order Runge-Kutta algorithm with a breaking condition when the pressure becomes zero or negative, because it means that the surface of the object is reached.

SolverTOVSinglePressure

The main method is

```
def solve(step_r, p0):
    ...

    pressure_is_negative = False

    while not pressure_is_negative:
        r = self.r_values[-1]
        p = self.p_values[-1]
        m = self.m_values[-1]

        p_new, m_new = self.runge_kutta_4th_step(r, step_r, p, m)
        r_new = r + step_r

        if p_new <= 0:
            pressure_is_negative = True

        self.r_values = np.append(self.r_values, r_new)
        self.p_values = np.append(self.p_values, p_new)
        self.m_values = np.append(self.m_values, m_new)
```

SolverTOVRangePressure

Class to integrate numerically the system of 2 ordinary differential equations (mass and pressure equations) given a range of initial central pressures. It uses the class SolverTOVSinglePressure for each initial central pressure.

SolverTOVRangePressure

Class to integrate numerically the system of 2 ordinary differential equations (mass and pressure equations) given a range of initial central pressures. It uses the class SolverTOVSinglePressure for each initial central pressure.
The main method is

```
def solve(step_r, initial_pressures):
    ...
    for p0 in initial_pressures:
        self.solver_single_p = SolverTOVSinglePressure(
            self.eos, self.relativity_corrections
        )
        self.solver_single_p.solve(step_r, p0)
        r, m, p = self.solver_single_p.get()
        self.masses = np.append(self.masses, m[-1])
        self.radii = np.append(self.radii, r[-1])
```

Full code

The full code is in solver.py.

Choices

There are four critical choices:

Choices

There are four critical choices:

- the use of NumPy arrays;

Choices

There are four critical choices:

- the use of NumPy arrays;
- the use of CubicSpline;

Choices

There are four critical choices:

- the use of NumPy arrays;
- the use of CubicSpline;
- the not use of solve_ivp;

Choices

There are four critical choices:

- the use of NumPy arrays;
- the use of CubicSpline;
- the not use of solve_ivp;
- the not use of performance improvement.

Equation of state

The equation of state depends on the matter content and the framework in which is computed. In this chapter, we will consider protons, neutrons and electrons in statistical mechanics and nuclear physics. In the next chapter, also advanced methods will be considered, such as particle physics and QCD.

Fermi gas of neutrons

Consider a degenerate ($T = 0$) ideal (non-interacting) Fermi gas of neutrons. In the non-relativistic limit ($k_F \ll mc$), the equation of state is

$$p = \frac{\hbar^2}{15\pi^2 m_n}\left(\frac{3\pi^2}{m_n c^2}\right)^{5/3}\epsilon^{5/3} = K_{nonrel}\epsilon^{5/3};$$

Fermi gas of neutrons

Consider a degenerate ($T = 0$) ideal (non-interacting) Fermi gas of neutrons. In the non-relativistic limit ($k_F \ll mc$), the equation of state is

$$p = \frac{\hbar^2}{15\pi^2 m_n}\left(\frac{3\pi^2}{m_n c^2}\right)^{5/3}\epsilon^{5/3} = K_{nonrel}\epsilon^{5/3};$$

whereas in the generic case, the equation of state can be computed from

$$p(x) = \frac{\epsilon_0}{24}\Big((2x^3 - 3x)(1+x^2)^{1/2} + 3\sinh^{-1}(x)\Big)\ ,$$

$$\epsilon(x) = nm_n c^2 + \frac{\epsilon_0}{8}\Big((2x^3 + x)(1+x^2)^{1/2} - \sinh^{-1}(x)\Big)\ .$$

The main functions are

```
def compute_pressure_gen(x, p):
    ...
    p_calculated = (
        e_0_n / 24 * ((2 * x**3 - 3 * x) * (1 + x**2) ** (1 / 2)
        + 3 * np.arcsinh(x))
    )
    return p_calculated - p

def compute_x_gen(p):
    ...
    x = opt.toms748(
        compute_pressure_gen, 1e-4, 1e2, args=(p,)
    )
    return x

def compute_energy_density_gen(p):
    ...
    x = compute_x_gen(p)
    e = compute_energy_density_from_x_gen(x)
    return e
```

Fermi gas of neutrons, protons and electrons

If we add also electrons and protons, the equation of state can be computed from

$$p_{tot} = \sum_i p_i \ , \quad p_i = \frac{\epsilon_0}{24}\Big((2x_i^3 - 3x_i)(1+x_i^2)^{1/2} + 3\,\mathrm{sinh}^{-1}(x_i)\Big) \ ,$$

$$\epsilon_{tot} = \sum_i \epsilon_i \ , \quad \epsilon_i = n_i m_i c^2 + \frac{\epsilon_0}{8}\Big((2x_i^3 + x_i)(1+x_i^2)^{1/2} - \mathrm{sinh}^{-1}(x_i)\Big) \ ,$$

with constraint

$$\sqrt{k_n^2 c^2 + m_n^2 c^4} = \sqrt{k_p^2 c^2 + m_p^2 c^4} + \sqrt{k_e^2 c^2 + m_e^2 c^4} \ .$$

The main functions are

```
def beta_equilibrium_condition(k_p, k_n):
    ...
    mu_p = (k_p**2 * c**2 + m_p**2 * c**4) ** (
        1 / 2
    )
    mu_e = (k_p**2 * c**2 + m_e**2 * c**4) ** (
        1 / 2
    )
    mu_n = (k_n**2 * c**2 + m_n**2 * c**4) ** (
        1 / 2
    )
    return mu_n - mu_p - mu_e

def compute_k_p(k_n):
    ...
    k_p = opt.toms748(
        beta_equilibrium_condition, 1e-17, 1e2, args=(k_n,)
    )
    return k_p
```

```
def compute_pressure_from_x_npe(x, e_0):
    ...
    p = e_0 / 24 * ((2 * x**3 - 3 * x) * (1 + x**2) ** (1 / 2)
    + 3 * np.arcsinh(x))
    return p

def compute_pressure_npe(k_n):
    ...
    k_p = compute_k_p(k_n)
    x_e = k_p / m_e / c
    x_p = k_p / m_p / c
    x_n = k_n / m_n / c
    p_e = compute_pressure_from_x_npe(x_e, e_0_e)
    p_n = compute_pressure_from_x_npe(x_n, e_0_n)
    p_p = compute_pressure_from_x_npe(x_p, e_0_p)
    p_tot = p_n + p_p + p_e
    return p_tot

def compute_k_n(n):
    ...
    k_n = hbar * (3 * np.pi**2 * n) ** (1 / 3)
    return k_n
```

```
def equation_of_state_npe(number_densities):
    ...
    for k in neutron_fermi_momenta:
        p_below = compute_pressure_pe(k)
        p_above = compute_pressure_npe(k)

        if p_below < 3.038e24:
            pressures_npe = np.append(pressures_npe, p_below)
            e_below = compute_energy_density_pe(k)
            energy_densities_npe = np.append(energy_densities_npe, e_below)
        else:
            pressures_npe = np.append(pressures_npe, p_above)
            e_above = compute_energy_density_npe(k)
            energy_densities_npe = np.append(energy_densities_npe, e_above)
    ...
```

Empirical interactions

If we add empirical interactions

$$\frac{\epsilon(n)}{n} = m_n c^2 + < E_0 > u^{2/3} + \frac{A}{2}u + \frac{B}{\sigma+1}u^{\sigma} \ ,$$

the equation of state can be computed from

$$p(n) = n_0 \Big(\frac{2}{3} < E_0 > u^{5/3} + \frac{A}{2}u^2 + \frac{B\sigma}{\sigma+1}u^{\sigma+1} + (2^{2/3}-1) < E_0 > (\frac{2}{3}u^{5/3} - u^2) + S_0 u^2 \Big) \ ,$$

$$\epsilon(n) = n \Big(m_n c^2 + 2^{2/3} < E_0 > u^{2/3} + \frac{A}{2}u + \frac{B}{\sigma+1}u^{\sigma} + (S_0 - (2^{2/3}-1) < E_0 >) u \Big) \ .$$

Skyrme interactions

If we add Skyrme interactions

$$V(x,y) = \delta^3(x-y)\left(\frac{t_3 n}{6} - t_0\right) ,$$

the equation of state can be computed from

$$p(n) = \frac{2(3\pi^2\hbar^3)^{2/3}}{10 m_n} n^{5/3} + \frac{t_3}{12} n^3 - \frac{t_0}{4} n^2 ,$$

$$\epsilon(n) = m_n n c^2 + \frac{3(3\pi^2\hbar^3)^{2/3}}{10 m_n} n^{5/3} + \frac{t_3}{24} n^3 - \frac{t_0}{4} n^2 .$$

For the last two, the same procedure as the generic Fermi gas has been implemented.

Full code

The full code is in fermi_eos.py and fermi_eos_examples.ipynb.

Choices

There are three critical choices:

Choices

There are three critical choices:

- the use of NumPy arrays;

Choices

There are three critical choices:

- the use of NumPy arrays;
- the use of toms748 algorithm;

Choices

There are three critical choices:

- the use of NumPy arrays;
- the use of toms748 algorithm;
- to either use root finding or give a range of number densities as input.

CompOSE archive

We can also use solver.py for more realistic equations of state, present in the CompOSE archive.

compose.py

Sound speed, gravitational redshift and Newtonian moment of inertia are added as well

```
def compute_sound_speed(range_p, range_e):
    ...
    p_prime = numerical_derivative(range_e, range_p)
    sound_speeds = np.array([])
    sound_speeds = np.maximum(p_prime, 0)
    return sound_speeds

def compute_redshift(r, m):
    ...
    r = r * km
    m = m * m_sun
    z = (1 - 2 * G * m / (r * c**2)) ** (-1 / 2) - 1
    return z

def compute_moment_inertia(r, m):
    ...
    r = r * km
    m = m * m_sun
    I = (2 / 5) * m * r**2
    return I
```

Full code

The full code is in compose.py and compose_examples.ipynb.

The end.