Mario L. Gutierrez Abed
364009832
mlg3843@rit.edu

Problem Set 4
Computational Astrophysics

II-I8-202I

Problem I. We will consider the following boundary value problem:

$$\frac{\mathrm{d}^2\psi}{\mathrm{d}r^2} + \frac{2}{r}\frac{\mathrm{d}\psi}{\mathrm{d}r} + 4\pi\rho = 0, \qquad 0 < r < r_{\max} \tag{Ia}$$

$$\left.\frac{\mathrm{d}\psi}{\mathrm{d}r}\right|_{r=0} = 0 \tag{Ib}$$

$$r_{\max}\left.\frac{\mathrm{d}\psi}{\mathrm{d}r}\right|_{r_{\max}} + \psi(r_{\max}) = 0. \tag{Ic}$$

In all cases, we you will need to solve this particular BVP using the Chebyshev collocation method we described in class.

- a) In class we demonstrated a code that solves this BVP with $r_{\max} = 20$ and $\rho(r) = e^{-r^2}/\pi^{3/2}$. Using this code, or one of your own, measure the L^∞-norm of the error in the solution as a function of the number of $N_{\max}$ (the highest order Chebyshev polynomial used in the expansion).
- b) Let $\rho(r)$ be given by

$$\rho(r) = \begin{cases} e^{-r} & \text{if } \ 0 \le r \le 1, \\ 0 & \text{if } \ r > 1. \end{cases} \tag{2}$$

  Solve the given BVP (take $r_{\max} = 5$). This time, measure the L^∞- and L^2-norms of the error by evaluating the LHS of Eq. (Ia) on a grid of points $r_i = i(5/M)$, $i = 1, 2, \ldots, M-1$, where $M \sim 1000$.
- c) Repeat the previous problem, but this time split the computational domain into two subdomains $[0, 1]$ and $[1, 5]$ (note that the point $r = 1$ is on both subdomains). Again, measure the error as in the previous problem.

Solution to a). In order to apply the Chebyshev collocation method we first need to shift our grid to $[-1, 1]$. This can be achieved from a general interval $[r_{\min}, r_{\max}]$ by applying the mapping

$$x(r) = 2\,\frac{r - r_{\min}}{r_{\max} - r_{\min}} - 1 \qquad \forall\, r \in [r_{\min}, r_{\max}], \tag{3}$$

so that $x(r_{\min}) = -1$ and $x(r_{\max}) = 1$. We will also need to translate back from $[-1, 1]$ to $[r_{\min}, r_{\max}]$ via

$$r(x) = r_{\min} + (x+1)\frac{r_{\max} - r_{\min}}{2} \qquad \forall\, x \in [-1, 1]. \tag{4}$$

The Chebyshev expansion is then given by

$$\psi(r) \approx \sum_{n=0}^{N_{\max}} c_n T_n(x(r)). \tag{5}$$

where the T_n are given by

$$T_n(x) = \cos(n \arccos(x)), \qquad -1 \le x \le 1. \tag{6}$$

The Gauß-Lobatto points are the $(N_{\max} + 1)$ roots of Eq. (6), which are given by

$$x_k = \cos\left(\frac{[N_{\max} - k]\pi)}{N_{\max}}\right), \qquad k = 0, \ldots, N_{\max}. \tag{7}$$

We will also make use of the first and second derivatives of the Chebyshev polynomials, which are given by

$$T_n'(x) = \frac{n \sin(n \arccos x)}{\sqrt{1-x^2}} \tag{8a}$$

$$= \frac{n \sin(n\varphi)}{\sin\varphi} \tag{8b}$$

$$T_n''(x) = -\frac{n^2 \cos(n \arccos x)}{1-x^2} + \frac{nx \sin(n \arccos x)}{\sqrt[3]{1-x^2}} \tag{8c}$$

$$= -\frac{n^2 \cos(n\varphi)}{\sin^2\varphi} + \frac{n \cos\varphi \sin(n\varphi)}{\sin^3\varphi}, \tag{8d}$$

where we used $x = \cos\varphi$. The problem with these expressions, however, is that they appear to singular at $x = \pm 1$ (or $\varphi \in \{0, \pi\}$). To remedy the situation we need to consider the limits as we approach these values; first we look at $T_n'(\pm 1)$:

$$\lim_{\varphi\to 0} \frac{n \sin(n\varphi)}{\sin\varphi} = n^2,$$
$$\lim_{\varphi\to \pi} \frac{n \sin(n\varphi)}{\sin\varphi} = (-1)^{n+1} n^2.$$

Similarly, for $T_n''(\pm 1)$,

$$\lim_{\varphi\to 0} \left(-\frac{n^2 \cos(n\varphi)}{\sin^2\varphi} + \frac{n \cos\varphi \sin(n\varphi)}{\sin^3\varphi} \right) = \frac{1}{3} n^2 (n^2 - 1),$$
$$\lim_{\varphi\to \pi} \left(-\frac{n^2 \cos(n\varphi)}{\sin^2\varphi} + \frac{n \cos\varphi \sin(n\varphi)}{\sin^3\varphi} \right) = \frac{(1)^{n+1}}{3} n^2 (n^2 - 1).$$

These expressions are written in the enclosed `chebyshev.py` file:

```python
'''
    Define the Chebyshev polynomial and its first and second derivatives
'''

import numpy as np
from functools import lru_cache

@lru_cache
def Cheby_poly(n, x):
    assert -1. <= x <= 1.
    phi = np.arccos(x)
    return np.cos(n * phi)


@lru_cache
def Cheby_poly_x(n, x):
    assert -1. <= x <= 1.
    phi = np.arccos(x)
    eps = 1.e-10

    if np.fabs(1.-x) < eps:
        der = n*n
    elif np.fabs(-1.-x) < eps:
        der = (-1.)**(n+1) * n*n
    else:
        der = n * np.sin(n * phi) / np.sin(phi)

    return der


@lru_cache
def Cheby_poly_xx(n, x):
    assert -1. <= x <= 1.

    phi     = np.arccos(x)
    sin_phi = np.sin(phi)
    cos_phi = np.cos(phi)
    eps     = 1.e-10


    if np.fabs(1.-x) < eps:
        der = 1./3. * n*n * (n*n - 1.)
    elif np.fabs(-1.-x) < eps:
        der = (-1.)**(n+1) * 1./3. * n*n * (n*n - 1.)
    else:
        der1 = - n * n * np.cos(n * phi) / sin_phi**2
        der2 = n * cos_phi * np.sin(n * phi) / sin_phi**3
        der  = der1 + der2

    return der
```

Hence, with ψ given as in Eq. (5), the Neumann condition (1b) on the left boundary is given by

$$\begin{aligned} 0 = \left.\frac{\mathrm{d}\psi}{\mathrm{d}r}\right|_{r=r_{\min}} &= \sum_{n=0}^{N_{\max}} c_n T_n'(x(0))\, x'(r_{\min}) \\ &= \frac{2}{r_{\max}-r_{\min}} \sum_{n=0}^{N_{\max}} T_n'(-1)\, c_n. \end{aligned} \tag{9}$$

Similarly, for the Robin condition (1c) on the right boundary,

$$\begin{aligned} 0 &= r_{\max}\left.\frac{\mathrm{d}\psi}{\mathrm{d}r}\right|_{r_{\max}} + \psi(r_{\max}) \\ &= r_{\max} \sum_{n=0}^{N_{\max}} c_n T_n'(x(r_{\max}))\, x'(r_{\max}) + \sum_{n=0}^{N_{\max}} c_n T_n(x(r_{\max})) \\ &= \frac{2\, r_{\max}}{r_{\max}-r_{\min}} \sum_{n=0}^{N_{\max}} c_n T_n'(1) + \sum_{n=0}^{N_{\max}} c_n T_n(1) \\ &= \sum_{n=0}^{N_{\max}} \left[\frac{2\, r_{\max}}{r_{\max}-r_{\min}} T_n'(1) + T_n(1) \right] c_n. \end{aligned} \tag{10}$$

Finally, at the interior points we have the full BVP (1a):

$$\begin{aligned} \frac{\mathrm{d}^2\psi}{\mathrm{d}r^2} + \frac{2}{r}\frac{\mathrm{d}\psi}{\mathrm{d}r} &= -4\pi\rho \\ \sum_{n=0}^{N_{\max}} c_n \left[T_n''(x(r_i))\,(x'(r_i))^2 + T_n'(x(r_i))\, \underbrace{x''(r)}_{=0} \right] + \frac{2}{r_i} \sum_{n=0}^{N_{\max}} c_n\, T_n'(x(r_i))\, x'(r_i) &= -4\pi\rho(r_i) \\ \sum_{n=0}^{N_{\max}} \left[T_n''(x(r_i)) \left(\frac{2}{r_{\max}-r_{\min}} \right)^2 + \frac{2}{r_i} T_n'(x(r_i)) \frac{2}{r_{\max}-r_{\min}} \right] c_n &= -4\pi\rho(r_i) \\ \frac{4}{r_{\max}-r_{\min}} \sum_{n=0}^{N_{\max}} \left[\frac{1}{r_{\max}-r_{\min}} T_n''(x(r_i)) + \frac{1}{r_i} T_n'(x(r_i)) \right] c_n &= -4\pi\rho(r_i), \end{aligned} \tag{11}$$

where $i = 1, \ldots, N_{\max}-1$. In matrix form,

$$\underbrace{\begin{bmatrix} \mathbf{T}_{00} & \cdots & \mathbf{T}_{0,n} & \cdots & \mathbf{T}_{0,N_{\max}} \\ \mathbf{T}_{10} & \cdots & \mathbf{T}_{1,n} & \cdots & \mathbf{T}_{1,N_{\max}} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{T}_{n,0} & \cdots & \mathbf{T}_{n,n} & \cdots & \mathbf{T}_{n,N_{\max}} \\ \vdots & \ddots & \vdots & \ddots & \vdots \\ \mathbf{T}_{N_{\max}-1,0} & \cdots & \mathbf{T}_{N_{\max}-1,n} & \cdots & \mathbf{T}_{N_{\max}-1,N_{\max}-1} \\ \mathbf{T}_{N_{\max},0} & \cdots & \mathbf{T}_{N_{\max},n} & \cdots & \mathbf{T}_{N_{\max},N_{\max}} \end{bmatrix}}_{\mathbf{T}} \underbrace{\begin{bmatrix} c_0 \\ c_1 \\ \vdots \\ c_n \\ \vdots \\ c_{N_{\max}-1} \\ c_{N_{\max}} \end{bmatrix}}_{\mathbf{c}} = \underbrace{\begin{bmatrix} 0 \\ \tilde{\varrho}_1 \\ \vdots \\ \tilde{\varrho}_n \\ \vdots \\ \tilde{\varrho}_{N_{\max}-1} \\ 0 \end{bmatrix}}_{\tilde{\varrho}}, \tag{12}$$

where

$$\tilde{\varrho}_n := -4\pi\rho(r_n)$$

and the T_{ij} are the terms multiplying the c_j coefficients in Eqs. (9)-(11) for the i^{th} collocation point (the top and bottom rows have the terms from Eqs. (9) and (10), respectively, while all other rows have the terms from Eq. (11), i.e., the interior points). Thus we have a dense $(N_{\max}+1) \times (N_{\max}+1)$ system

$$\mathbf{T}\mathbf{c} = \tilde{\boldsymbol{\varrho}},$$

which needs to be solved for $\mathbf{c}$. The spectral solver is written in the following class, which is included in the `spectral_solver.py` file:

```python
import numpy as np
import chebyshev as ch

# establish some default parameters
N_MAX_DEFAULT = 100
M_MAX_DEFAULT = 1000
R_MIN_DEFAULT = 0.
R_MAX_DEFAULT = 20.

''' ----------------------------
    Create Spectral Solver Class
    ---------------------------- '''

class SpectralSystem():

    '''   Constructor   '''
    def __init__(self, rho,
                       N_max = N_MAX_DEFAULT,
                       M_max = M_MAX_DEFAULT,
                       r_min = R_MIN_DEFAULT,
                       r_max = R_MAX_DEFAULT
                ):
        self.rho         = rho
        self.N_max       = N_max
        self.M_max       = M_max
        self.r_min       = r_min
        self.r_max       = r_max
        self.dxdr        = 2. / (self.r_max - self.r_min)
        self.x           = self.gauss_lobatto_grid(self.N_max)
        self.r           = self.r_of_x(self.x, self.r_min, self.r_max)
        self.source      = self.source_func(self.r)
        self.coeffs      = self.solve(self.spectral_matrix(), self.source)
        self.r_test      = self.r_test_grid(self.r_min, self.r_max, self.M_max)
        self.x_test      = self.x_of_r(self.r_test, self.r_min, self.r_max)
        self.source_test = self.source_func(self.r_test)
        self.y           = self.y_func(self.x_test, self.N_max, self.coeffs)
        self.y_r         = self.y_r_func(self.x_test, self.N_max, self.dxdr, self.coeffs)
        self.y_rr        = self.y_rr_func(self.x_test, self.N_max, self.dxdr, self.coeffs)


    '''   Class Methods   '''
    # Change of coordinates mappings
    def x_of_r(self, r, rmin, rmax):
        return 2. * (r - rmin)/(rmax - rmin) - 1.

    def r_of_x(self, x, rmin, rmax):
        return rmin + (x + 1.) * (rmax - rmin) / 2.


    # generate grid with Guass-Lobatto roots
    def gauss_lobatto_grid(self, nmax):

        grid = np.zeros((nmax + 1))
        for i in range(nmax + 1):
            grid[i] = np.cos((nmax - i) * np.pi / nmax)

        return grid


    # build the source vector 4*pi*rho
    def source_func(self, r):
        source_vec = np.zeros_like(r)
        for i in range(1, len(source_vec)-1):
            source_vec[i] = 4. * np.pi * self.rho(r[i])
        return source_vec

    # build spectral matrix
    def spectral_matrix(self):

        T = np.zeros((self.N_max+1, self.N_max+1))

        # top and bottom rows
        for j in range(self.N_max + 1):
            T[0,j]          = self.dxdr * ch.Cheby_poly_x(j, self.x[0])
            T[self.N_max,j] = self.r_max * self.dxdr * \
                              ch.Cheby_poly_x(j, self.x[self.N_max]) + \
                              ch.Cheby_poly(j, self.x[self.N_max])
        # rest of the matrix
        for i in range(1,self.N_max):
            for j in range(self.N_max + 1):
                inner  = 0.5 * self.dxdr * ch.Cheby_poly_xx(j, self.x[i]) + \
                         1./self.r[i] * ch.Cheby_poly_x(j, self.x[i])
                T[i,j] = 2. * self.dxdr * inner

        return T


    # Compute the solution coefficients c_n
    def solve(self, mat, src):
        C = np.linalg.solve(mat, -src)
        return C



    '''
        The following methods are only used when testing
        the numerical solution on some test grid
    '''

    def r_test_grid(self, rmin, rmax, mmax):
        grid     = np.zeros(mmax+1)
        grid[0]  = rmin
        grid[-1] = rmax

        for i in range(1, mmax):
            grid[i] = i  * rmax / mmax

        return grid



    '''
        Reconstruct the solution y(r) and its first and second derivatives
        evaluated on test grid
    '''

    def y_func(self, xgrid, nmax, c_n):

        temp  = np.zeros(nmax+1)
        y_vec = np.zeros_like(xgrid)

        for i in range(len(y_vec)):
            for j in range(len(temp)):
                temp[j] = ch.Cheby_poly(j, xgrid[i])
            y_vec[i] = np.dot(c_n,temp)

        return y_vec


    def y_r_func(self, xgrid, nmax, der, c_n):

        temp  = np.zeros(nmax+1)
        y_r_vec = np.zeros_like(xgrid)

        for i in range(len(y_r_vec)):
            for j in range(len(temp)):
                temp[j] = der * ch.Cheby_poly_x(j, xgrid[i])
            y_r_vec[i] = np.dot(c_n,temp)

        return y_r_vec

    def y_rr_func(self, xgrid, nmax, der, c_n):

        temp  = np.zeros(nmax+1)
        y_rr_vec = np.zeros_like(xgrid)

        for i in range(len(y_rr_vec)):
            for j in range(len(temp)):
                temp[j] = der * der * ch.Cheby_poly_xx(j, xgrid[i])
            y_rr_vec[i] = np.dot(c_n,temp)

        return y_rr_vec
```

We then measure the L^{∞}-norm of the error in the solution as a function of the number of N_{max} for a few values:

```python
import numpy as np

'''
    Define \rho function for Part a)
'''

def rho_1(r):
    return np.exp(-r*r) / np.pi**1.5

import spectral_solver as spec

'''
   Do an N test, calculating the residual for some N values
'''
N_test = np.array((50, 75, 100, 150, 200))

residuals = np.zeros_like(N_test, dtype = np.float64)

for N in N_test:

    system = spec.SpectralSystem(rho = rho_1, N_max = N)
    c      = system.coeffs
    T      = system.spectral_matrix()
    source = system.source
    lhs    = np.zeros(N+1)

    for i in range(N+1):
        lhs[i] = np.dot(T[i], c)

    res = np.linalg.norm(lhs + source, ord=np.inf)
    ind = np.where(N_test == N)[0][0]

    residuals[ind] = res


'''
    Plot residuals at the collocation points
'''

from matplotlib import pyplot as plt

plt.plot(N_test, residuals, 'ro-', label="Residuals")
plt.xlabel(r'$N$')
plt.ylabel(r'$L^\infty$- norm of residuals')
plt.legend(fancybox=True, framealpha=1, borderpad=1, shadow=True,  loc='upper right')
plt.show()
```

The residuals are zero at the collocation points, as expected. This is shown in the following plot (deviations from zero are, of course, due to round-off errors):

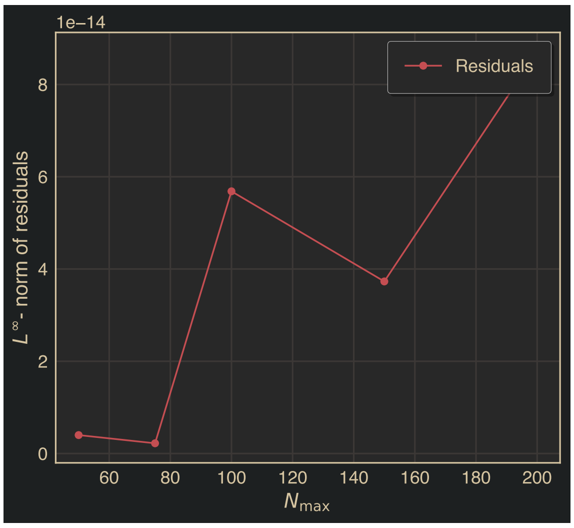

Figure 1: L^∞-norm of the residual for $N_{\max} \in \{50, 75, 100, 150, 200\}$.

We also plot the residuals on a test grid where the nodes are not the collocation points. We see improvement as $N_{\max}$ increases, as expected:

```python
'''
    Evaluate the numerical solution on a new test grid (not the collocation points)
'''

residuals = np.zeros_like(N_test, dtype=np.float64)

for N in N_test:

    test_system = spec.SpectralSystem(rho = rho_1, N_max = N)
    psi_r       = test_system.y_r
    psi_rr      = test_system.y_rr
    r           = test_system.r_test
    r[0]        = r[0] + 1e-8    # avoid dividing by zero
    test_source = test_system.source_test

    res       = psi_rr + 2./r * psi_r + test_source
    res[0] = 0.
    res_linf = np.linalg.norm(res, ord=np.inf)
    ind       = np.where(N_test == N)[0][0]
    residuals[ind] = res_linf
```

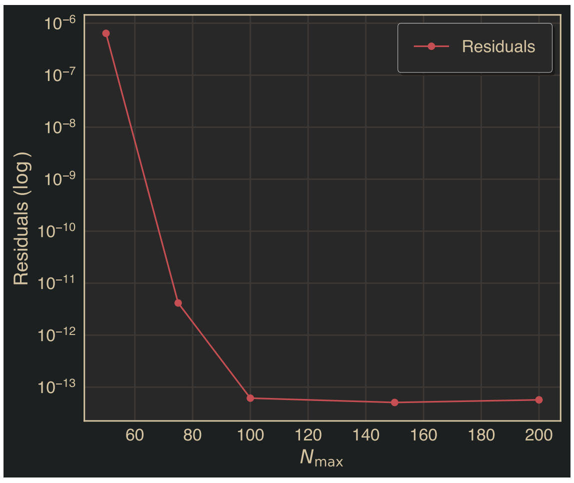

Figure 2: L^∞-norm of the residual for $N_{\max} \in \{50, 75, 100, 150, 200\}$ on a test grid. Plot is on a semilog scale.

Finally, we compare with the analytical solution:

```python
'''
    Compare numerical and solutions
'''

from scipy.special import erf

test_grid = test_system.r_test
sol = test_system.y

plt.plot(test_grid, sol, 'r-o', label="Approximate")
plt.plot(test_grid, erf(test_grid) / test_grid, 'g-o', label="Exact")
plt.xlabel(r'$r$')
plt.ylabel(r'$\psi(r)$')
plt.legend(fancybox=True, framealpha=1, borderpad=1, shadow=True,  loc='upper right')
plt.show()
```

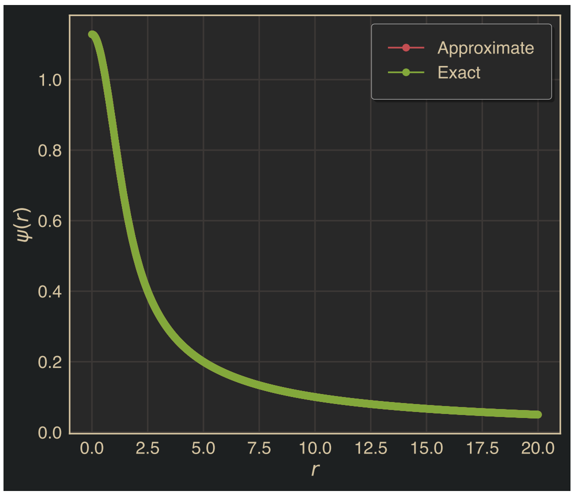

Figure 3: Comparison between analytical and numerical solutions.

♠

Solution to b). This time we calculate the L^∞- and L^2-norms of the residuals, using ρ given by Eq. (2) and evaluating on a test grid given by points $r_i = i(5/M), i = 1, 2, \ldots, M-1$, where $M = 1500$:

```python
'''
    Define \rho function for Parts b) and c)
'''

def rho_2(r):
    rho = np.where(0. <= r <= 1., np.exp(-r), 0.)
    return rho

'''
   Do an N test, calculating the L^2 amd L^\infty norms
   of the residuals for some N values
'''

N_test_2        = np.array((500, 750, 1000, 1250))
residuals_linf = np.zeros_like(N_test_2,  dtype=np.float64)
residuals_l2   = np.zeros_like(N_test_2,  dtype=np.float64)

for N in N_test_2:

    system_2 = spec.SpectralSystem(rho = rho_2, N_max = N, M_max = 1500)
    psi_r       = system_2.y_r
    psi_rr      = system_2.y_rr
    r            = system_2.r_test
    r[0]         = r[0] + 1e-8      #avoid dividing by zero
    test_source = system_2.source_test

    res      = psi_rr + 2./r * psi_r + test_source
    res[0]   = 0.
    res_linf = np.linalg.norm(res, ord=np.inf)
    res_l2   = np.linalg.norm(res)
    ind      = np.where(N_test_2 == N)[0][0]

    residuals_linf[ind] = res_linf
    residuals_l2[ind]   = res_l2

'''
    Plot L^2 and L^\infty norms of the residuals
'''

plt.plot(N_test_2, residuals_linf, 'mo-', label=r'$L^{\infty}$ norm')
plt.plot(N_test_2, residuals_l2,  'go-', label=r'$L^2$ norm')
plt.xlabel(r'$N$')
plt.ylabel('Residuals')
plt.legend(fancybox=True, framealpha=1, borderpad=1, shadow=True,  loc='upper right')
plt.show()
```

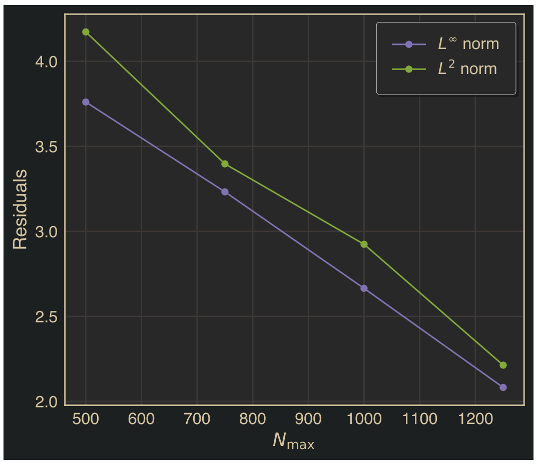

Figure 4: L^∞- and L^2-norms of the residuals, using ρ given by Eq. (2).

The results do improve with increasing N, but because of the discontinuity at $r = 1$ of the ρ function given by Eq. (2), the residuals are still quite large, even for relatively large $N_{\max}$. We shall remedy this in the next part. ♠

Solution to c). This time we solve the same problem using two adjacent grids $[r^L_{\min}, r^L_{\max}] = [0, 1]$ and $[r^R_{\min}, r^R_{\max}] = [1, 5]$. The Chebyshev expansion will then be

$$\psi_L(r) \approx \sum_{n=0}^{N^L_{\max}} c^L_n T_n(x_L(r^L)) \tag{13a}$$

$$\psi_R(r) \approx \sum_{n=0}^{N^R_{\max}} c^R_n T_n(x_R(r^R)), \tag{13b}$$

where

$$x_L\colon [r^L_{\min}, r^L_{\max}] \mapsto [-1, 1]$$
$$x_R\colon [r^R_{\min}, r^R_{\max}] \mapsto [-1, 1].$$

The resulting algebraic equations are similar as before, except that now we have expressions pertaining exclusively to the left and right grids, and we also have the following two condtions at the interface $r = 1$:

$$\psi_L(1) = \psi_R(1) \tag{14a}$$
$$\psi'_L(1) = \psi'_R(1) \tag{14b}$$

which, expanding via (13), become

$$\sum_{n=0}^{N^L_{\max}} c^L_n\, T_n(1) = \sum_{n=0}^{N^R_{\max}} c^R_n\, T_n(-1) \tag{15a}$$
$$\frac{2}{r^L_{\max} - r^L_{\min}} \sum_{n=0}^{N^L_{\max}} c^L_n\, T'_n(1) = \frac{2}{r^R_{\max} - r^R_{\min}} \sum_{n=0}^{N^R_{\max}} c^R_n\, T'_n(-1). \tag{15b}$$

Hence we end up with

$$\frac{2}{r^L_{\max} - r^L_{\min}} \sum_{n=0}^{N^L_{\max}} T'_n(-1)\, c^L_n = 0 \qquad \text{(Left grid, Left boundary)}$$
$$\frac{4}{r^L_{\max} - r^L_{\min}} \sum_{n=0}^{N^L_{\max}} \left[\frac{1}{r^L_{\max} - r^L_{\min}} T''_n\left(x_L(r^L_{i_L})\right) + \frac{1}{r^L_{i_L}} T'_n\left(x_L(r^L_{i_L})\right)\right] c^L_n = -4\pi\rho\left(r^L_{i_L}\right) \qquad \text{(Left grid, Interior)}$$
$$\sum_{n=0}^{N^L_{\max}} T_n(1)\, c^L_n - \sum_{n=0}^{N^R_{\max}} T_n(-1)\, c^R_n = 0 \qquad \text{(Interface, Eq. (15a))}$$
$$\frac{2}{r^L_{\max} - r^L_{\min}} \sum_{n=0}^{N^L_{\max}} T'_n(1)\, c^L_n - \frac{2}{r^R_{\max} - r^R_{\min}} \sum_{n=0}^{N^R_{\max}} T'_n(-1)\, c^R_n = 0 \qquad \text{(Interface, Eq. (15b))}$$
$$\frac{4}{r^R_{\max} - r^R_{\min}} \sum_{n=0}^{N^R_{\max}} \left[\frac{1}{r^R_{\max} - r^R_{\min}} T''_n\left(x_R(r^R_{i_R})\right) + \frac{1}{r^R_{i_R}} T'_n\left(x_R(r^R_{i_R})\right)\right] c^R_n = -4\pi\rho\left(r^R_{i_R}\right) \qquad \text{(Right grid, Interior)}$$
$$\sum_{n=0}^{N^R_{\max}} \left[\frac{2\, r^R_{\max}}{r^R_{\max} - r^R_{\min}} T'_n(1) + T_n(1)\right] c^R_n = 0. \qquad \text{(Right grid, Right boundary)}$$

where $i_L = 1, \ldots, N^L_{\max} - 1$ and $i_R = 1, \ldots, N^R_{\max} - 1$. The matrix equation now has the form

$$\underbrace{\left[\begin{array}{c:c} \mathbf{T}_L & \mathbf{0} \\ \hdashline \mathbf{0} & \mathbf{T}_R \end{array}\right]}_{\mathbf{T}} \underbrace{\begin{bmatrix} c^L_0 \\ \vdots \\ \vdots \\ c_{N^L_{\max}} \\ c^R_0 \\ \vdots \\ \vdots \\ c_{N^R_{\max}} \end{bmatrix}}_{\mathbf{c}} = \underbrace{\begin{bmatrix} 0 \\ \tilde{\varrho}^L_1 \\ \vdots \\ \tilde{\varrho}_{N^L_{\max}-1} \\ 0 \\ 0 \\ \tilde{\varrho}^R_1 \\ \vdots \\ \tilde{\varrho}_{N^R_{\max}-1} \\ 0 \end{bmatrix}}_{\tilde{\varrho}}, \tag{17}$$

where the blue line represents the leftmost boundary condition, the magenta line the rightmost condition, and the dashed lines are the interface equations. Thus we have a dense $(N^L_{\max} + N^R_{\max} + 2) \times (N^L_{\max} + N^R_{\max} + 2)$ system,

$$\mathbf{Tc} = \tilde{\varrho}.$$

which we solve using a child class from the `SpectralSystem` parent class (found on the `spectral_two_grid.py` file): [1]

[1] While it is not necessary to create a separate child class, it certainly keeps the class's body more tidy.

```python
import numpy as np
import chebyshev as ch
import spectral_solver as spec

''' ---------------------------------------------
         Create Spectral Solver Child Class
    (splits the system into two adjacent intervals)
    ----------------------------------------------- '''

# some further default parameters
N_MAX_RIGHT = 20
M_MAX_RIGHT = 500
R_MIN_LEFT  = 0.
R_MAX_LEFT  = 1.
R_MIN_RIGHT = 1.
R_MAX_RIGHT = 5.

class SpectralSystemTwoDomains(spec.SpectralSystem):

    '''   Constructor  '''
    def __init__(self, rho,
                       N_max = spec.N_MAX_DEFAULT,
                       M_max = spec.M_MAX_DEFAULT,
                       r_min = R_MIN_LEFT,
                       r_max = R_MAX_LEFT,
                       N_max_right = N_MAX_RIGHT,
                       M_max_right = M_MAX_RIGHT,
                       r_min_right = R_MIN_RIGHT,
                       r_max_right = R_MAX_RIGHT
                ):
        spec.SpectralSystem.__init__(self, rho, N_max, M_max, r_min, r_max)
        self.N_max_right  = N_max_right
        self.N_max_total  = self.N_max + self.N_max_right + 2
        self.M_max_right  = M_max_right
        self.r_min_right  = r_min_right
        self.r_max_right  = r_max_right
        self.dxdr_right   = 2. / (self.r_max_right - self.r_min_right)
        self.x_right      = self.gauss_lobatto_grid(self.N_max_right)
        self.r_right      = self.r_of_x(self.x_right, self.r_min_right, self.r_max_right)
        self.source       = self.source_func2(self.r, self.r_right)
        self.coeffs       = self.solve(self.spectral_matrix2(), self.source)

        self.r_test_left  = np.linspace(self.r_min + 1.0e-4, self.r_max - 1e-4, self.N_max+1)
        self.r_test_right = np.linspace(self.r_min_right + 1.0e-4,
                            self.r_max_right, self.N_max_right+1)
        self.x_test_left  = self.x_of_r(self.r_test_left, self.r_min, self.r_max)
        self.x_test_right = self.x_of_r(self.r_test_right, self.r_min_right, self.r_max_right)

        self.r            = np.concatenate((self.r_test_left, self.r_test_right))
        self.source_test  = self.source_func2(self.r_test_left, self.r_test_right)

        self.y_left       = self.y_func(self.x_test_left, self.N_max, self.coeffs[:self.N_max + 1])
        self.y_right      = self.y_func(self.x_test_right, self.N_max_right,
                            self.coeffs[N_max + 1:])
        self.y            = np.concatenate((self.y_left, self.y_right))

        self.y_r_left     = self.y_r_func(self.x_test_left, self.N_max, self.dxdr,
                            self.coeffs[:self.N_max + 1])
        self.y_r_right    = self.y_r_func(self.x_test_right, self.N_max_right,
                            self.dxdr_right, self.coeffs[N_max + 1:])
        self.y_r          = np.concatenate((self.y_r_left, self.y_r_right))

        self.y_rr_left    = self.y_rr_func(self.x_test_left, self.N_max,
                            self.dxdr, self.coeffs[:self.N_max + 1])
        self.y_rr_right   = self.y_rr_func(self.x_test_right, self.N_max_right,
                            self.dxdr_right, self.coeffs[N_max + 1:])
        self.y_rr         = np.concatenate((self.y_rr_left, self.y_rr_right))


    ''' Rebuild the source vector '''
    def source_func2(self, rleft, rright):

        r_left_len  = self.N_max + 1
        r_right_len = self.N_max_right + 1
        r_len       = r_left_len + r_right_len
        source_vec  = np.zeros(r_len)

        for i in range(1, self.N_max):
            source_vec[i] = 4. * np.pi * self.rho(rleft[i])

        for i in range(1, self.N_max_right ):
            k = i + r_left_len
            source_vec[k] = 4. * np.pi * self.rho(rright[i])

        return source_vec


    ''' Rebuild spectral matrix '''
    def spectral_matrix2(self):

        r_left_len  = self.N_max + 1
        r_right_len = self.N_max_right + 1
        r_len       = r_left_len + r_right_len
        T           = np.zeros((r_len, r_len))


        # Top part of the matrix
        for j in range(r_left_len):
            T[0,j] = self.dxdr * ch.Cheby_poly_x(j, self.x[0])

        for i in range(1, r_left_len - 1):
            for j in range(r_left_len):
                inner  = 0.5 * self.dxdr * ch.Cheby_poly_xx(j, self.x[i]) + \
                         1./self.r[i] * ch.Cheby_poly_x(j, self.x[i])
                T[i,j] = 2. * self.dxdr * inner

        # Interface
        for j in range(r_left_len):
            T[self.N_max,j]   = ch.Cheby_poly(j, self.x[-1])
            T[self.N_max+1,j] = self.dxdr * ch.Cheby_poly_x(j, self.x[-1])

        for j in range(r_right_len):
            k = j + r_left_len
            T[self.N_max,k]   = - ch.Cheby_poly(j, self.x[0])
            T[self.N_max+1,k] = - self.dxdr_right * ch.Cheby_poly_x(j, self.x[0])


        # Bottom part of the matrix
        for j in range(r_right_len):
            k = j + r_left_len
            T[-1,k] = self.r_max_right * self.dxdr_right * \
                            ch.Cheby_poly_x(j, self.x_right[-1]) + \
                            ch.Cheby_poly(j, self.x_right[-1])

        for i in range(1, r_right_len - 1):
            for j in range(r_right_len):

                row = i + r_left_len
                col = j + r_left_len

                inner_right = 0.5 * self.dxdr_right * ch.Cheby_poly_xx(j, self.x_right[i]) + \
                              1./self.r_right[i] * ch.Cheby_poly_x(j, self.x_right[i])
                T[row,col]  = 2. * self.dxdr_right * inner_right

        return T
```

We re-calculate the L^∞- and L^2-norms of the residuals, using the two-grid approach:

```python
import spectral_two_grid as spec2
from IPython.display import display, Latex

system_2_improved = spec2.SpectralSystemTwoDomains(rho = rho_2,
                       N_max = 499,
                       M_max = 500,
                       r_min = spec2.R_MIN_LEFT,
                       r_max = spec2.R_MAX_LEFT,
                       N_max_right = 499,
                       M_max_right = 100,
                       r_min_right = spec2.R_MIN_RIGHT,
                       r_max_right = spec2.R_MAX_RIGHT
                )
```

```python

psi_r       = system_2_improved.y_r
psi_rr      = system_2_improved.y_rr
r           = system_2_improved.r
r[0]        = r[0] + 1e-8     #avoid dividing by zero
test_source = system_2_improved.source_test

res      = psi_rr + 2./r * psi_r + test_source
res[0]   = 0.
res[499] = 0.   #avoid the discontinuity
res_linf = np.linalg.norm(res, ord=np.inf)
res_l2   = np.linalg.norm(res)

display(Latex(f'The $L^{{\infty}}$ norm is {res_linf}'))
display(Latex(f'The $L^2$ norm is {res_l2}'))
```

The errors now much smaller. The resulting L^{∞}-norm is 4.5484×10^{-11}, while the L^2-norm is 4.5539×10^{-11}. To conclude, we do a similar N-test as the one above:

```python
'''
   Do an N test, calculating the L^2 amd L^\infty norms
   of the residuals for some N values
'''

import spectral_two_grid as spec2

N_test_2       = np.array((100, 200, 300, 400, 500))
residuals_linf = np.zeros_like(N_test_2,  dtype=np.float64)
residuals_l2   = np.zeros_like(N_test_2,  dtype=np.float64)

for N in N_test_2:

    system_2_improved = spec2.SpectralSystemTwoDomains(rho = rho_2,
                           N_max = N,
                           M_max = 500,
                           r_min = spec2.R_MIN_LEFT,
                           r_max = spec2.R_MAX_LEFT,
                           N_max_right = N,
                           M_max_right = 100,
                           r_min_right = spec2.R_MIN_RIGHT,
                           r_max_right = spec2.R_MAX_RIGHT
                )

    psi_r       = system_2_improved.y_r
    psi_rr      = system_2_improved.y_rr
    r           = system_2_improved.r
    r[0]        = r[0] + 1e-8     #avoid dividing by zero
    test_source = system_2_improved.source_test

    res      = psi_rr + 2./r * psi_r + test_source
    res[0]   = 0.

    #avoid the discontinuity
    discontinuity = np.where(np.abs(res) > 1.)[0][0]
    res[discontinuity] = 0.

    res_linf = np.linalg.norm(res, ord=np.inf)
    res_l2   = np.linalg.norm(res)
    ind      = np.where(N_test_2 == N)[0][0]

    residuals_linf[ind] = res_linf
    residuals_l2[ind]   = res_l2
```

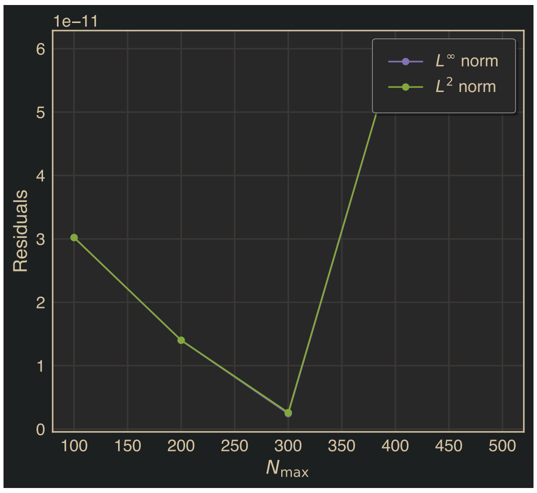

Figure 5: L^{∞}- and L^{2}-norms of the residuals, for $N_{\max} \in \{100, 200, 300, 400, 500\}$. We see much better results now. At around $N_{\max} = 300$ we are hitting round-off.

♠