Mario L. Gutierrez Abed
364009832
mlg3843@rit.edu

Problem Set 3
Computational Astrophysics

10-22-2021

Problem 1. *In spherical symmetry, the Laplacian takes on the form*

$$\nabla^2\psi = \frac{1}{r^2}\frac{\mathrm{d}}{\mathrm{d}r}\left(r^2\frac{\mathrm{d}\psi}{\mathrm{d}r}\right) = \frac{\mathrm{d}^2\psi}{\mathrm{d}r^2} + \frac{2}{r}\frac{\mathrm{d}\psi}{\mathrm{d}r}. \tag{1}$$

Using Gauss-Seidel iterations and second-order-accurate finite differencing, solve the following Poisson problem

$$\nabla^2\psi = -4\pi\rho, \qquad 1 < r < 10 \tag{2a}$$
$$\left.\frac{\mathrm{d}\psi}{\mathrm{d}r}\right|_{r=1} = 0 \tag{2b}$$
$$\psi(10) = 1 \tag{2c}$$
$$\rho(r) = \frac{1}{r^4}. \tag{2d}$$

Develop your own code to do this.

- *a) Starting with $n = 1024$ cells, plot the residual versus r after 100, 200, and 1000 iterations of Gauss-Seidel (with no over relaxation).*
- *b) Repeat the above with $\omega = 1.5$.*
- *c) This problem can be solved exactly. Using the exact solution, determine the L^∞-norm of the error in the approximate value of ψ over the grid (once the GS algorithm has converged) as a function of the number of cells.*

Solution. The centered discretization of the BVP (2a), given the Laplacian described by Eq. (1), is of the form

$$\begin{aligned}
&\frac{\Psi_{i+1} - 2\Psi_i + \Psi_{i-1}}{h^2} + \frac{2}{r_i}\frac{\Psi_{i+1} - \Psi_{i-1}}{2h} = -4\pi\rho_i \\
\implies\ & \Psi_{i+1} - 2\Psi_i + \Psi_{i-1} + \frac{h}{r_i}\left[\Psi_{i+1} - \Psi_{i-1}\right] = -4\pi h^2\frac{1}{r_i^4} \\
\implies\ & \left[1 - \frac{h}{r_i}\right]\Psi_{i-1} - 2\Psi_i + \left[1 + \frac{h}{r_i}\right]\Psi_{i+1} = -4\pi h^2\frac{1}{r_i^4} \qquad i = 1, \dots, n-1.
\end{aligned} \tag{3}$$

Using the notation

$$\begin{aligned}
\Theta_i^{\pm} &:= 1 \pm h/r_i \\
\varrho_i &:= -4\pi h^2\rho_i,
\end{aligned}$$

we cast (3) in matrix form:

$$\begin{bmatrix}
-2 & \Theta_1^+ & & & & \\
\Theta_2^- & -2 & \Theta_2^+ & & & \\
& \Theta_3^- & -2 & \Theta_3^+ & & \\
& & \ddots & \ddots & \ddots & \\
& & & \Theta_{n-2}^- & -2 & \Theta_{n-2}^+ \\
& & & & \Theta_{n-1}^- & -2
\end{bmatrix}
\begin{bmatrix} \Psi_1 \\ \Psi_2 \\ \vdots \\ \Psi_i \\ \vdots \\ \Psi_{n-1} \end{bmatrix}
=
\begin{bmatrix} \varrho_1 \\ \varrho_2 \\ \vdots \\ \varrho_i \\ \vdots \\ \varrho_{n-1} \end{bmatrix}.$$

This expression is not entirely accurate, however. We are missing what happens at the boundaries $\{r_0 = 1, r_n = 10\}$. From the

Neumann condition (2b), we get

$$\begin{aligned} \frac{\Psi_1 - \Psi_0}{h} &= 0 \\ \implies \Psi_1 &= \Psi_0. \end{aligned} \tag{4}$$

Thus, for $i = 1$,

$$\begin{aligned} &\Theta_1^- \Psi_0 - 2\Psi_1 + \Theta_1^+ \Psi_2 = \varrho_1 \\ \implies \quad &-\Theta_1^+ \Psi_1 + \Theta_1^+ \Psi_2 = \varrho_1. && \text{(By (4))} \end{aligned}$$

Hence, our matrix equation becomes

$$\underbrace{\begin{bmatrix} -\Theta_1^+ & \Theta_1^+ & & & & \\ \Theta_2^- & -2 & \Theta_2^+ & & & \\ & \Theta_3^- & -2 & \Theta_3^+ & & \\ & & \ddots & \ddots & \ddots & \\ & & & \Theta_{n-2}^- & -2 & \Theta_{n-2}^+ \\ & & & & \Theta_{n-1}^- & -2 \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} \Psi_1 \\ \Psi_2 \\ \vdots \\ \Psi_i \\ \vdots \\ \Psi_{n-1} \end{bmatrix}}_{\boldsymbol{\Psi}} = \underbrace{\begin{bmatrix} \varrho_1 \\ \varrho_2 \\ \vdots \\ \varrho_i \\ \vdots \\ \varrho_{n-1} - \Theta_{n-1}^+ \end{bmatrix}}_{\tilde{\varrho}}, \tag{5}$$

where the last entry in the $\tilde{\varrho}$ vector comes from the Dirichlet condition (2c). Thus we have ended up with the system

$$A\boldsymbol{\Psi} = \tilde{\varrho},$$

which we need to solve for $\boldsymbol{\Psi}$. We shall solve this system using both Gauß-Seidel and Successive Overrelaxation (SOR), as instructed.

Second-Order Accuracy of One-Sided Derivative

Eq. (4) introduces a problem: This one-sided derivative approximation is only first-order accurate, while we discretized the BVP (2) using a centered, second-order accurate scheme. We cannot use the usual centered scheme for the boundary at $r_0 = 1$, because that would not yield an expression for Ψ_0. Instead we shall derive a one-sided scheme that is second order accurate. Let us put

$$\psi'(r) \approx c_0\Psi(r) + c_1\Psi(r+h) + c_2\Psi(r+2h), \tag{6}$$

where the c_i are coefficients that we need to determine. Now, Taylor-expanding,

$$\begin{aligned} \Psi(r+h) &= \Psi(r) + \Psi'(r)h + \frac{1}{2}\Psi''(r)h^2 + \mathcal{O}(h^3) \\ \Psi(r+2h) &= \Psi(r) + 2\Psi'(r)h + 2\Psi''(r)h^2 + \mathcal{O}(h^3). \end{aligned}$$

Matching these expressions with the coefficients c_i on (6), we get

$$\begin{aligned} \Psi'(r) = &\left(c_0 + c_1 + c_2\right)\Psi(r) \\ &+ \left(c_1 + 2c_2\right) h\Psi'(r) \\ &+ \left(c_1 + 4c_2\right)\frac{1}{2}h^2\Psi''(r). \end{aligned}$$

In order for this expression to be compatible with Eq. (6), the following linear system must be satisfied:

$$\begin{aligned} c_0 + c_1 + c_2 &= 0 \\ c_1 + 2c_2 &= \frac{1}{h} \\ c_1 + 4c_2 &= 0. \end{aligned}$$

The solution is

$$\left\{ c_0 = -\frac{3}{2h}, \; c_1 = \frac{2}{h}, \; c_2 = -\frac{1}{2h} \right\}.$$

Plugging back into (6), we end up with

$$\psi'(r) \approx \frac{1}{2h}\left[-3\Psi(r) + 4\Psi(r+h) - \Psi(r+2h)\right]. \tag{7}$$

Hence, at r_0 we get

$$\begin{aligned} 0 = \psi'(r)\Big|_{r=1} &\approx \frac{1}{2h}\left[-3\Psi_0 + 4\Psi_1 - \Psi_2\right] \\ \implies \Psi_0 &= \frac{4\Psi_1 - \Psi_2}{3}. \end{aligned} \tag{8}$$

Thus, at $i = 1$,

$$\begin{aligned} \Theta_1^-\Psi_0 - 2\Psi_1 + \Theta_1^+\Psi_2 &= \varrho_1 \\ \implies \Theta_1^-\left[\frac{4}{3}\Psi_1 - \frac{1}{3}\Psi_2\right] - 2\Psi_1 + \Theta_1^+\Psi_2 &= \varrho_1 \\ \implies \left[\frac{4}{3}\Theta_1^- - 2\right]\Psi_1 + \left[\Theta_1^+ - \frac{1}{3}\Theta_1^-\right]\Psi_2 &= \varrho_1 \end{aligned}$$

Hence, in place of the matrix equation (5), we have

$$\underbrace{\begin{bmatrix} \Xi_1 & \Xi_2 & & & & \\ \Theta_2^- & -2 & \Theta_2^+ & & & \\ & \Theta_3^- & -2 & \Theta_3^+ & & \\ & & \ddots & \ddots & \ddots & \\ & & & \Theta_{n-2}^- & -2 & \Theta_{n-2}^+ \\ & & & & \Theta_{n-1}^- & -2 \end{bmatrix}}_{\hat{A}} \underbrace{\begin{bmatrix} \Psi_1 \\ \Psi_2 \\ \vdots \\ \Psi_i \\ \vdots \\ \Psi_{n-1} \end{bmatrix}}_{\boldsymbol{\Psi}} = \underbrace{\begin{bmatrix} \varrho_1 \\ \varrho_2 \\ \vdots \\ \varrho_i \\ \vdots \\ \varrho_{n-1} - \Theta_{n-1}^+ \end{bmatrix}}_{\tilde{\varrho}}, \tag{9}$$

where $\Xi_{1,2}$ are given by

$$\Xi_1 = \frac{4}{3}\Theta_1^- - 2, \qquad \Xi_2 = \Theta_1^+ - \frac{1}{3}\Theta_1^-.$$

We shall solve both this sytem and the one given by (5) and check if all this hassle was worth it!

The idea behind iterative methods such as Gauß-Seidel is that, if we have a large and sparse linear system $A\boldsymbol{\Psi} = \tilde{\varrho}$, we rewrite it in an equivalent form

$$\boldsymbol{\Psi} = B\boldsymbol{\Psi} + \mathbf{d}. \tag{10}$$

How the matrix B and the vector $\mathbf{d}$ are defined depends on which iterative method we use, as we shall soon see. Then, starting with an initial approximation $\boldsymbol{\Psi}^{(0)}$ of the solution vector $\boldsymbol{\Psi}$, we generate a sequence $\{\boldsymbol{\Psi}^{(k)}\}$ by the iterative scheme

$$\boldsymbol{\Psi}^{(k+1)} = B\boldsymbol{\Psi}^{(k)} + \mathbf{d} \qquad k = 0, 1, \ldots \tag{11}$$

We stop this algorithm either when the ***relative residual norm*** satisfies

$$\frac{\|\tilde{\varrho} - A\boldsymbol{\Psi}^{(k)}\|}{\|\tilde{\varrho}\|} \leq \epsilon \tag{12}$$

for some user-defined tolerance $\epsilon > 0$, or when the algorithm reaches a maximum number of iterations that the user is willing to allow. We shall use this relative residual norm as a stopping criterion on part *c)* of the problem. For parts *a)* and *b)* we are explictly asked to plot the residual as a function of r after a certain number of iterations, so we do not use the stopping criterion here.

The first order of business for all such iterative methods, then, is to rewrite the matrix A in the form $A = L + D + U$, where

$$\begin{aligned} L &= \text{lower triangular with zeroes on the diagonal;} \\ D &= \text{diagonal;} \\ U &= \text{upper triangular with zeroes on the diagonal.} \end{aligned}$$

That is,

$$\underbrace{\begin{pmatrix} \ddots & & & & U \\ & \ddots & & & \\ & & D & & \\ & & & \ddots & \\ L & & & & \ddots \end{pmatrix}}_{A} = \underbrace{\begin{pmatrix} 0 & & & & 0 \\ & \ddots & & & \\ & & 0 & & \\ & & & \ddots & \\ * & & & & 0 \end{pmatrix}}_{L} + \underbrace{\begin{pmatrix} a_{11} & & & & 0 \\ & \ddots & & & \\ & & a_{ii} & & \\ & & & \ddots & \\ 0 & & & & a_{nn} \end{pmatrix}}_{D} + \underbrace{\begin{pmatrix} 0 & & & & * \\ & \ddots & & & \\ & & 0 & & \\ & & & \ddots & \\ 0 & & & & 0 \end{pmatrix}}_{U}.$$

We can now implement our algorithm to find the solution $\boldsymbol{\Psi}$ to the system $A\boldsymbol{\Psi} = \tilde{\varrho}$, given some random initial guess, say, $\boldsymbol{\Psi}^{(0)} = [1, \ldots, 1]^\top$.[1] For the Gauß-Seidel method, we rewrite $(L + D + U)\boldsymbol{\Psi} = \tilde{\varrho}$ as

$$(L + D)\boldsymbol{\Psi} = \tilde{\varrho} - U\boldsymbol{\Psi},$$

which implies

$$\begin{aligned} \boldsymbol{\Psi} &= (L + D)^{-1}\left[\tilde{\varrho} - U\boldsymbol{\Psi}\right] \\ &= \underbrace{-(L + D)^{-1}U}_{B}\,\boldsymbol{\Psi} + \underbrace{(L + D)^{-1}\tilde{\varrho}}_{\mathbf{d}}. \end{aligned}$$

If we write $\boldsymbol{\Psi} = (L + D)^{-1}\left[\tilde{\varrho} - U\boldsymbol{\Psi}\right]$ in the iterative form

$$\boldsymbol{\Psi}^{(k+1)} = (L + D)^{-1}\left[\tilde{\varrho} - U\boldsymbol{\Psi}^{(k)}\right],$$

we see that this expression may also be written as

$$\boldsymbol{\Psi}^{(k+1)} = D^{-1}\left[\tilde{\varrho} - L\boldsymbol{\Psi}^{(k+1)} - U\boldsymbol{\Psi}^{(k)}\right]. \tag{13}$$

Thus, in terms of the individual entries, the Gauß-Seidel algorithm is given by[2]

$$\Psi_i^{(k+1)} = \frac{1}{a_{ii}}\left(\tilde{\varrho}_i - \sum_{\substack{j=0 \\ i>0}}^{i-1} a_{ij}\Psi_j^{(k+1)} - \sum_{j=i+1}^{n-2} a_{ij}\Psi_j^{(k)}\right), \qquad i = 0, \ldots, n-2. \tag{14}$$

The Gauß-Seidel method, however, can be slow to converge in some applications. SOR is a fairly minimal adjustment that can help performance tremendously. To implement the latter, we introduce a relaxation factor ω which is typically in the range $1 < \omega < 2$ (the case $\omega = 1$ reduces to Gauß-Seidel). In this recipe, the equation $\omega A\boldsymbol{\Psi} = \omega\tilde{\varrho}$ takes the form

$$(\omega L + \omega D + \omega U)\boldsymbol{\Psi} = \omega\tilde{\varrho},$$

which in turn implies

$$\begin{aligned} (D + \omega L)\boldsymbol{\Psi} &= \omega\tilde{\varrho} - \omega U\boldsymbol{\Psi} + (1 - \omega)D\boldsymbol{\Psi} \\ \Longrightarrow \quad \boldsymbol{\Psi} &= \underbrace{(D + \omega L)^{-1}\left[(1 - \omega)D - \omega U\right]}_{B}\boldsymbol{\Psi} + \underbrace{\omega(D + \omega L)^{-1}\tilde{\varrho}}_{\mathbf{d}}. \end{aligned}$$

This yields the SOR algorithm

$$\boxed{\Psi_i^{(k+1)} = \frac{\omega}{a_{ii}}\left(\tilde{\varrho}_i - \sum_{\substack{j=0 \\ i>0}}^{i-1} a_{ij}\Psi_j^{(k+1)} - \sum_{j=i+1}^{n-2} a_{ij}\Psi_j^{(k)}\right) + (1 - \omega)\Psi_i^{(k)}, \qquad i = 0, \ldots, n-2.} \tag{15}$$

[1] It turns out that the particular matrix we are dealing with in this problem is not very adequate for iterative methods, since it is neither strictly diagonally-dominant nor positive definite. The latter are conditions that guarantee convergence of Gauß-Seidel and SOR for *any* given initial guess…As a result, convergence for this particular system turns out to be quite problematic. It takes quite a bit of fine-tuning to get convergence in a reasonable number of iterations.

[2] Here and in the SOR algorithm I am switching the index of the $\tilde{\varrho}_i$ and Ψ_i by -1 in order accommodate for C++'s zero-base indexing. The entries of Ψ at the boundaries are appended after the algorithm is run.

Since Gauß-Seidel is just a special case of SOR, we shall only use the algorithm (15) in our code, and simply change ω accordingly. Without further ado, here is the content of our `main.cpp` file: [3]

```cpp
// main.cpp
// Successive Overrelaxation (SOR) applied to the Poisson equation
//    \nabla^2 \psi = - 4\pi\rho
// assuming spherical symmetry.
// Created by Mario L Gutierrez on 10/03/21.

#include <iostream>
#include <fstream>
#include <cmath>
#include <Eigen/Dense>
#include "functions.hpp"

using namespace std;
using namespace Eigen;

/*  Set this bool to 'false' for parts a) and b) of the problem,
    or set to 'true' for part c)   */
const bool N_TEST {false};
// const bool N_TEST {true};

/* -------------------------------------------------------------------------------*/
/* -------------------------------------------------------------------------------*/
// Start of main function
int main(int argc, const char * argv[]) {

    /* --------------------------------------------------------
    Parts a), b) of the Problem. Set N_TEST=false in global pars.
    --------------------------------------------------------*/
    if (!N_TEST){

        const int n {1024};
        const double h {(rn-r0)/n};

        // MatrixXd A   = A_mat(n);    // use either A or \hat{A}
        MatrixXd Ah  = Ah_mat(n);
        VectorXd rhs = rhs_vec(n);

        // set initial guess for the algorithm
        VectorXd guess = VectorXd::Ones(n-1);

        /* ------------------------------------------------------
        Save residuals after 100, 200, and 1000 iterations of GS
        (omega = 1)
        ------------------------------------------------------ */
        VectorXd residuals_100  = SOR_RES(Ah, rhs, guess, n, 100);
        VectorXd residuals_200  = SOR_RES(Ah, rhs, guess, n, 200);
        VectorXd residuals_1000 = SOR_RES(Ah, rhs, guess, n, 1000);

        ofstream res100file ("../Data/res100.csv");
            for (int j{0}; j < residuals_100.size(); j++)
                    res100file << abs(residuals_100(j)) << endl;
        res100file.close();

        ofstream res200file ("../Data/res200.csv");
            for (int j{0}; j < residuals_200.size(); j++)
                    res200file << abs(residuals_200(j)) << endl;
        res200file.close();

        ofstream res1000file ("../Data/res1000.csv");
            for (int j{0}; j < residuals_1000.size(); j++)
                    res1000file << abs(residuals_1000(j)) << endl;
        res1000file.close();

        /* -------------------------------------------------------
        Save residuals after 100, 200, and 1000 iterations of SOR_RES
        (omega = 1.5)
        -------------------------------------------------------*/
        VectorXd residuals_SOR_100  = SOR_RES(Ah, rhs, guess, n, 100, 1.5);
        VectorXd residuals_SOR_200  = SOR_RES(Ah, rhs, guess, n, 200, 1.5);
        VectorXd residuals_SOR_1000 = SOR_RES(Ah, rhs, guess, n, 1000, 1.5);

```

[3] The Eigen library is required for this code to compile.

```
        ofstream res100SORfile ("../Data/res_SOR_100.csv");
            for (int j{0}; j < residuals_SOR_100.size(); j++)
                    res100SORfile << abs(residuals_SOR_100(j)) << endl;
        res100SORfile.close();

        ofstream res200SORfile ("../Data/res_SOR_200.csv");
            for (int j{0}; j < residuals_SOR_200.size(); j++)
                    res200SORfile << abs(residuals_SOR_200(j)) << endl;
        res200SORfile.close();

        ofstream res1000SORfile ("../Data/res_SOR_1000.csv");
            for (int j{0}; j < residuals_SOR_1000.size(); j++)
                    res1000SORfile << abs(residuals_SOR_1000(j)) << endl;
        res1000SORfile.close();


        /* -------------------------------------------------------
        Save array with r-values
        -------------------------------------------------------*/
        VectorXd r_vals = VectorXd::LinSpaced(n-1,r0+h,rn-h);

        ofstream rfile ("../Data/r_vals.csv");
            for (int j{0}; j < r_vals.size(); j++)
                    rfile << r_vals(j) << endl;
        rfile.close();
    }
    // end of parts a) and b)


   /* -------------------------------------------------------
    Part c) of the Problem. Set N_TEST=true in global pars.
    -------------------------------------------------------*/
    if (N_TEST){

        // Do N_TEST for different values of n
        VectorXi n_array (13);
        n_array << 200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800;

        ofstream nfile ("../Data/Partc/nvals.csv");
        ofstream n_errors_file ("../Data/Partc/n_errors.csv");

        for (int N : n_array){
            // MatrixXd A     = A_mat(N);    // use either A or \hat{A}
            MatrixXd Ah    = Ah_mat(N);
            VectorXd rhs   = rhs_vec(N);
            VectorXd guess = VectorXd::Ones(N-1);

            // Save the full grid, this time including the endpoints as well
            VectorXd r_grid       = VectorXd::LinSpaced(N+1,r0,rn);
            VectorXd solution     = exact_sol(r_grid);
            VectorXd num_solution = SOR(Ah, rhs, guess, N, 100000, 1.9);
            double   n_error      = compare(num_solution, solution);

            // output numerical solution for N=800
            if (N==800){
                ofstream Psifile ("../Data/Partc/psi.csv");
                for (int j{0}; j < num_solution.size(); j++)
                        Psifile << num_solution(j) << endl;
                Psifile.close();
            }

            nfile << N << endl;
            n_errors_file << n_error << endl;
        }
        nfile.close();
        n_errors_file.close();
    }
    // end of part c)

}  //end of main

/* ----------------------------------------------------------------------------*/
/* ----------------------------------------------------------------------------*/
```

The routines implemented in `functions.cpp` are given here:

```cpp
#include "functions.hpp"
/* ------------------------------------------------------------------------------
                        FUNCTION IMPLEMENTATIONS
 ------------------------------------------------------------------------------*/

// Generate A matrix
MatrixXd A_mat(const int &dim){

    const double h {(rn-r0)/dim};
    MatrixXd A(dim-1,dim-1);

    double r_i {};
    double r_j {};
    double theta_p {};
    double theta_m {};

    // Generate (n-1)x(n-1) matrix A (from Eq.5)
    for (int i {0}; i < dim-1; i++) {

        r_i     = r0 + (i+1)*h;
        r_j     = r0 + (i+2)*h;
        theta_p = 1. + h/r_i;
        theta_m = 1. - h/r_j;

        if (i == 0)
            A(i,i) = -theta_p;
        else
            A(i,i) = -2.;

        if (i != dim-2){
            A(i,i+1) = theta_p;
            A(i+1,i) = theta_m;
        }
    }
    return A;
}

// Generate \hat{A} matrix
MatrixXd Ah_mat(const int &dim){

    const double h {(rn-r0)/dim};
    const double Xi_1 = 4./3. * (1. - h/(r0+h)) - 2.;           //\Xi_i from Eq. (8)
    const double Xi_2 = 1. + h/(r0+h) - 1./3. * (1. - h/(r0+h));
    double r_i {};
    double r_j {};
    double theta_p {};
    double theta_m {};

    MatrixXd Ah(dim-1,dim-1);

    // Generate (n-1)x(n-1) matrix \hat{A} (from Eq.9)
    for (int i {0}; i < dim-1; i++) {

        r_i     = r0 + (i+1)*h;
        r_j     = r0 + (i+2)*h;
        theta_p = 1. + h/r_i;
        theta_m = 1. - h/r_j;

        if (i == 0){
            Ah(i,i)   = Xi_1;
            Ah(i,i+1) = Xi_2;
            Ah(i+1,i) = theta_m;
        }
        else
            Ah(i,i) = -2.;

        if (i != dim-2 && i != 0){
            Ah(i,i+1) = theta_p;
            Ah(i+1,i) = theta_m;
        }
    }

    return Ah;
}
```

```
// Generate (\tilde{\varrho}}) vector
VectorXd rhs_vec(const int &dim){

    const double h {(rn-r0)/dim};
    VectorXd rhs(dim-1);
    double r_i {};
    double rho_i {};
    double theta_p {};

    // Generate (n-1)x1 rhs vector (\tilde{\varrho}}) (from Eq.5 or 9)
    for (int i {0}; i < dim-1; i++) {

        r_i     = r0 + (i+1)*h;
        rho_i   = - 4. * M_PI * h*h * 1./pow(r_i,4);
        theta_p = 1. + h/r_i;

        if (i == dim-2)
            rhs(i) = rho_i - theta_p;
        else
            rhs(i) = rho_i;
    }
    return rhs;
}

// SOR_RES function implementation
VectorXd SOR_RES(const MatrixXd &A, const VectorXd &b,
                 const VectorXd &x0, const int &dim,
                 const int max_it, const double omega){
    /*
    ** SOR_RES routine that returns the residual b-Ax from a system Ax=b
       at the end of SOR algorithm**
    INPUTS:
        A: (dim-1)x(dim-1) matrix
        b: (dim-1)-vector
        x0: initial guess for iterative SOR solver
        dim: num of cells in the grid
        max_it: max number of itartions allowed
        omega: relaxation parameter (omega=1 => Gauss-Seidel)
    OUTPUT:
        diff: the residuals b - Ax
    */

    const double h {(rn-r0)/dim};
    VectorXd Psi_new = x0;
    size_t it {0};

    do{

        VectorXd Psi_old = Psi_new;

        for (int i {0}; i < dim-1; i++){

            VectorXd ai2  = A.row(i)(seq(i+1,dim-2));
            VectorXd psi2 = Psi_old(seq(i+1,dim-2));
            double sum2   = ai2.dot(psi2);

            if (i == 0)
                Psi_new(i) = omega/A(i,i) * (b(i) - sum2) + (1. - omega) * Psi_old(i);
            else{
                VectorXd ai1  = A.row(i)(seq(0,i-1));
                VectorXd psi1 = Psi_new(seq(0,i-1));
                double sum1   = ai1.dot(psi1);

                Psi_new(i) = omega/A(i,i) * (b(i) - sum1 - sum2) + (1. - omega) * Psi_old(i);
            }
        }

        it+=1;
    } while (it <= max_it);

    // residuals to be output
    VectorXd diff = b - A * Psi_new;
    diff = diff/(h*h);

    return diff;
}
```

```

// SOR function implementation
VectorXd SOR(const MatrixXd &A, const VectorXd &b,
             const VectorXd &x0, const int &dim,
             const int max_it, const double omega,
             const double tol){
    /*
    ** SOR routine that returns the solution x to the system Ax=b at the end of SOR algorithm **
    INPUTS:
        A: (dim-1)x(dim-1) matrix
        b: (dim-1)-vector
        x0: initial guess for iterative SOR solver
        dim: num of cells in the grid
        max_it: max number of itartions allowed
        omega: relaxation parameter (omega=1 => Gauss-Seidel)
        tol: user-defined error tolerance
    OUTPUT:
        Psi_full: the solution itself
    */

    VectorXd Psi_new = x0;
    size_t it {0};
    double residual_norm {};
    const double h {(rn-r0)/dim};

    do{

        VectorXd Psi_old = Psi_new;

        for (int i {0}; i < dim-1; i++){

            VectorXd ai2  = A.row(i)(seq(i+1,dim-2));
            VectorXd psi2 = Psi_old(seq(i+1,dim-2));
            double sum2   = ai2.dot(psi2);

            if (i == 0)
                Psi_new(i) = omega/A(i,i) * (b(i) - sum2) + (1. - omega) * Psi_old(i);
            else{
                VectorXd ai1  = A.row(i)(seq(0,i-1));
                VectorXd psi1 = Psi_new(seq(0,i-1));
                double sum1   = ai1.dot(psi1);

                Psi_new(i)    = omega/A(i,i) * (b(i) - sum1 - sum2) + (1. - omega) * Psi_old(i);
            }
        }

        VectorXd diff = b - A * Psi_new;
        diff = diff/(h*h);
        residual_norm = diff.lpNorm<Infinity>()/b.lpNorm<Infinity>();

        cout << "\nresidual = " << residual_norm << " at iteration " << it << endl;

        if (residual_norm <= tol){
            cout << "The relative residual norm " << residual_norm <<
                    " has reached the derired tolerance. \
                     Convergence successful after " << it <<
                     " iterations!\n" << endl;
            break;
        }

        it+=1;
    } while (it <= max_it);

    // append solution at the boundaries
    VectorXd Psi_full(dim+1);
    // double Psi_0 = Psi_new(0);                              // if using Neumann condition (4)
    double Psi_0 = 1./3. * (4. * Psi_new(0) - Psi_new(1));    // if using Neumann condition (8)
    double Psi_n = 1.;
    Psi_full << Psi_0, Psi_new, Psi_n;

    return Psi_full;
}

// Evaluate exact solution (Eq. 17) on the grid
VectorXd exact_sol(const VectorXd &grid){
    VectorXd sol (grid.size());
    for (int i {0}; i < grid.size(); i++)
        sol(i) = - 2. * M_PI/(grid(i)*grid(i)) + 4. * M_PI/grid(i) + 1. - 19. * M_PI/50.;
    return sol;
}

// Compare L^{\infty} norm of the error between numerical and exact solutions
double compare(const VectorXd &numerical, const VectorXd &analytical){
    VectorXd diff = numerical - analytical;
    double error = diff.lpNorm<Infinity>();
    return error;
}
```

All the above routines are declared on the `function.hpp` header file:

```
#ifndef FUNCTIONS_HPP
#define FUNCTIONS_HPP

#include <cmath>
#include <iostream>
#include <Eigen/Dense>

using namespace std;
using namespace Eigen;

// Global parameters
const double r0 {1.};
const double rn {10.};

// Function prototypes
VectorXd SOR_RES    (const MatrixXd &A, const VectorXd &b,
                     const VectorXd &x0, const int &dim,
                     const int max_it = 100,   const double omega = 1.);
VectorXd SOR        (const MatrixXd &A, const VectorXd &b,
                     const VectorXd &x0, const int &dim,
                     const int max_it = 1000, const double omega = 1.,
                     const double tol = 1.e-10);
VectorXd exact_sol  (const VectorXd &x);
VectorXd rhs_vec    (const int &dim);
MatrixXd A_mat      (const int &dim);
MatrixXd Ah_mat     (const int &dim);
double compare      (const VectorXd &numerical, const VectorXd &analytical);

#endif
```

We import the files output by this code into Python for plotting purposes. Here are the residuals as a function of r for $\omega = 1$ and $\omega = 1.5$ for 100, 200, and 1000 iterations:

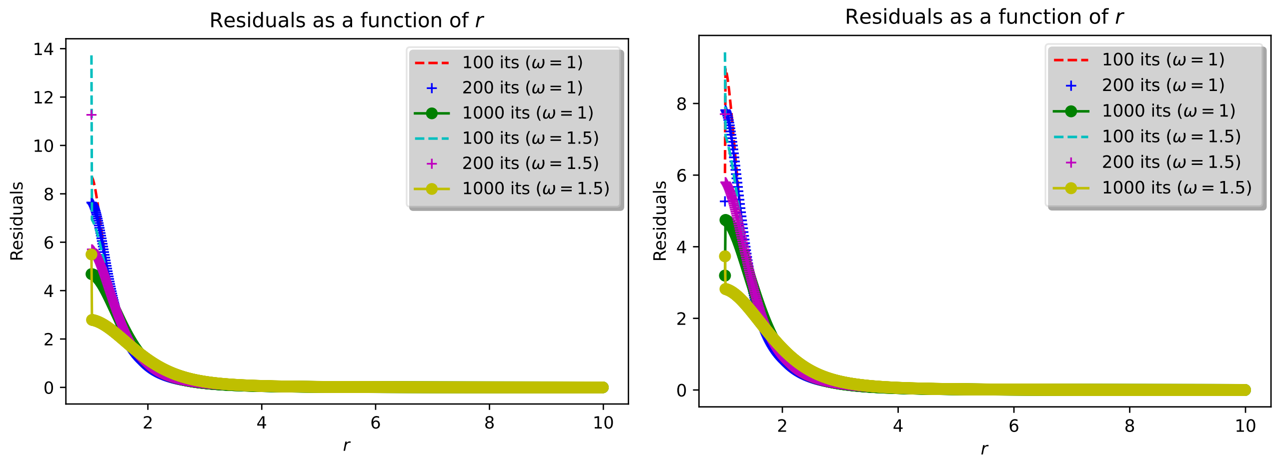

Figure 1: Residuals after 100, 200, and 1000 iterations of SOR, with $\omega \in \{1, 1.5\}$. On the left we have the results using the differential matrix A from Eq. (5), and on the right we have the results using $\hat{A}$ from Eq. (9). The latter shows, as expected, an improvement over the result obtained using A. Still, the residuals are quite large, especially on the left boundary, but this is due to the fact that we need many more iterations to yield an acceptable result, since the differential matrix for this particular problem is not perfectly suited for Gauß-Seidel (i.e., it is not strictly diagonally-dominant nor is it postive definite).

For the last part of the problem we need to use the analytical solution. Integrating Eq. (2a) twice with respect to r, we get the general solution

$$\psi(r) = -\frac{2\pi}{r^2} - \frac{c_0}{r} + c_1. \tag{16}$$

From the given boundary conditions, we get the coefficients $c_0 = -4\pi$ and $c_1 = 1 - 19\pi/50$. Thus our particular exact solution is

$$\boxed{\psi(r) = -\frac{2\pi}{r^2} + \frac{4\pi}{r} + 1 - \frac{19\pi}{50}.} \tag{17}$$

The errors we get are not as small as one would like. As we alluded to earlier, this could be attributed to the fact that this particular matrix we are dealing with is neither strictly diagonally-dominant nor positive definite. The routine was tested on a different matrix that does fullfill these conditions, and the convergence there was quite fast and accurate. We would need many more iterations to significantly lower the error.

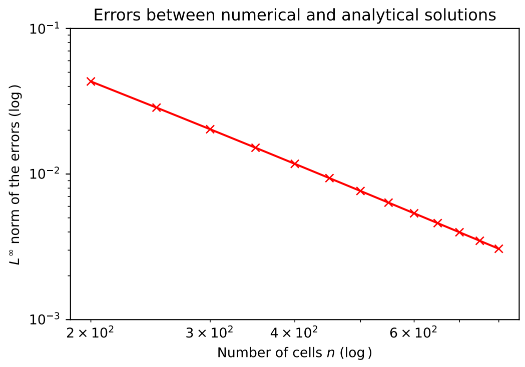

Figure 2: L^∞-norm of the errors for $n \in \{200, 250, 300, 350, 400, 450, 500, 550, 600, 650, 700, 750, 800\}$, in a $\log$-$\log$ scale.

The numerical and exact solutions are then plotted for $n = 800$:

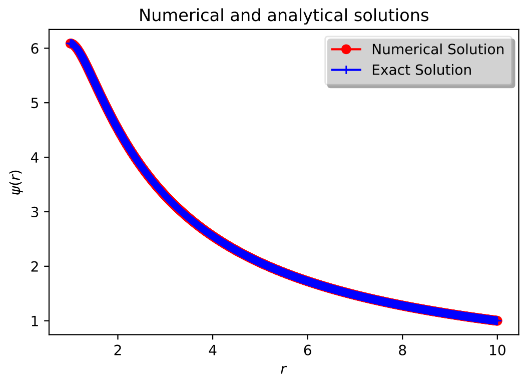

Figure 3: Comparison between analytical and numerical solutions. For the latter, 100000 iterations (or reaching a tolerance of 10^{-10}, whichever came first) were run using $\omega = 1.9$ and $n = 800$.

♠