Iterative Solution of the Poisson Equation

The purpose of this project is to develop an iterative solver for the Poisson equation on cartesian grids in multiple dimensions. In vector form this equation is given by:

$$\nabla^2 u = f \tag{1}$$

where $u(\mathbf{x})$ is the solution sought and $f(\mathbf{x})$ is a known function. Boundary conditions must be specified on the entire perimeter of the domain in order to determine a unique solution to the equation. For simplicity we will use so-called Dirichlet boundary conditions: the solution u is known on the boundary.

In cartesian coordinates the Poisson equation takes the following form

$$\nabla^2 u = \nabla \cdot (\nabla u) = \begin{cases} u_{xx} = f(x) & 1D \\ u_{xx} + u_{yy} = f(x,y) & 2D \\ u_{xx} + u_{yy} + u_{zz} = f(x,y,z) & 3D \end{cases} \tag{2}$$

In order to solve the problem numerically we need to replace the second order partial derivatives with second-order finite diffence approximations. For the one-dimensional case, for example, we would use:

$$u_{xx} = \frac{u_{i-1} - 2u_i + u_{i+1}}{\Delta x^2} + O(\Delta x^2). \tag{3}$$

The differential equations become a system of linear algebraic equations that we will solve using Jacobi iterations.

1 The 1D Poisson solver

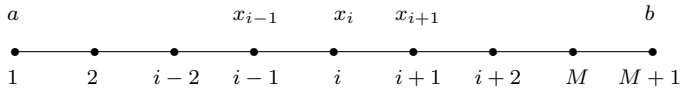

Figure 1: 1D finite difference grid for the interval $a \leq x \leq b$ using a uniform grid spacing $\Delta x = x_{i+1} - x_i = (b-a)/M$. The index of the nodes are the integers below the line, and the corresponding abcissa are shown above.

Figure 1 shows the discrete grid and the location of the nodes where the solution is sought. Applying the differential equation at each one of the interior nodes we obtain the algebraic system:

$$u_{i-1} - 2u_i + u_{i+1} = \Delta x^2 f_i, \text{ for } i = 2, 3, 4, \ldots, M \tag{4}$$

It is very important to remember that the system has only $(M-1)$ unknowns since u_1 and u_{M+1} are known from the boundary conditions; the unknowns u_i are located only at the interior nodes. The iterative solution proceeds by starting with a guess $u_i^{(0)}$ which, in general, will not satisfy equation 4. The guess is then updated to enforce the equality 4 at point i

$$u_i^{(n+1)} = \frac{u_{i+1}^{(n)} + u_{i+1}^{(n)} - \Delta x^2 f_i}{2} \text{ for } i = 2, 3, 4, \ldots, M \tag{5}$$

where the superscript n is the iteration index. This update is repeated until a convergence criterion is reached, for example the correction drops below a specified tolerance:

$$\max_{2 \leq i \leq M} \left| c_i^{(n)} \right| = \max_{2 \leq i \leq M} \left| u_i^{(n+1)} - u_i^{(n)} \right| < \epsilon \tag{6}$$

Theoretical analysis guarantees that the process will eventually reach a solution, and that the number of iteration needed to reach convergence scales as

$$K \geq \frac{\ln \epsilon}{\ln\left(1 - 2\sin^2 \frac{\pi}{2M}\right)} \sim -\frac{2M^2}{\pi^2} \ln \epsilon \tag{7}$$

The above can be used as a rough estimate to bound the iteration count.

The program design shown in 2-3 can be used as an initial design plan. The implied division of labor allows the main program to control the various tasks with a fair amount of flexibility. The code should be designed by stages, and you should leverage your previous efforts and re-use the software you have already built and tested to complete the assignment.

```
program poissonsolver
  use grid             ! geometry module
  use fileio           ! file i/o module
  use solver           ! new solver to be built
  use poissondata      ! BC, Forcing and 1st guess module
  implicit none
  integer, parameter :: M =8     ! number of intervals
  integer, parameter :: Mp=M+1   ! number of points
  real*8, parameter :: tol=1.d-9 ! error tolerance
  real*8  :: u(Mp)     ! solution
  real*8  :: f(Mp)     ! forcing function
  real*8  :: f(Mp)     ! x-coordinates
  real*8  :: enorm     ! error norms of the iteration process
  integer :: niter     ! number of iterations allowed/performed

  call SetGrid(xg,dx,smin,smax,M) ! Define the grid
  call WriteAsciiVector(xg,Mp)    ! save grid to file
  call DefineRhs(f,x,Mp)          ! Define forcing function
  call FirstGuess(u,x,Mp)         ! first Guess, can be simply u=0.
  call SetBC(u,x,Mp)              ! Set Boundary conditions

  call IterativeSolver(u,f,dx,tol,enorm,niter,Mp)  ! iterative solver

  call OutputSolution(u,Mp)       ! Save solution to a file

 stop
end program poissonsolver
```

Figure 2: Outline of main program. The main program sets-up the problem (geometry, forcing function, first guess and boundary conditions), calls the iterative solver, and outputs the solution to a file. The solver takes the tolerance

```
module solver
contains
subroutine IterativeSolver(u,f,dx,tol,enorm,niter,Mp)
  implicit none
  integer, intent(in) :: Mp       ! size of computational grid
  real*8, intent(in) :: f(Mp)     ! forcing function
  real*8, intent(in) :: dx        ! grid spacing
  real*8, intent(in) :: tol       ! tolerance
  real*8, intent(out) :: enorm    ! error norm reported
  integer, intent(inout) :: niter ! max number of iterations on input
  real*8, intent(inout) :: u(Mp)  ! solution
  integer :: nitermax,it
  nitermax = niter                ! max iterations allowed
  do it = 1,nitermax
    call JacobiIteration(u,f,dx,enorm,Mp) ! perform single iteration
    if (enorm < tol) then
      exit                   ! solution has converged exit loop
    endif
  enddo
  if (enorm > tol .and. it > nitermax)then
    print *,'Solution did not converge' ! error warning
  endif
  niterm = it
  return
end subroutine IterativeSolver

! JacobiIteration does a single update of solution u
! it returns an error norm measuring the maximum change:
! enorm = max_i |u_i^{n+1} - u_i|
subroutine JacobiIteration(u,f,dx,enorm,Mp) ! perform single iteration
  implicit none
  ...
  return
end subroutine JacobiIteration
end module solver
```

Figure 3: Outline of iterative solver. The iterative solver specified the tolerance desired, and the maximum number of iterations allowed. It returns, aside from the solution, an error measure of the iteration error, and the number of iterations necessary to reach the prescribed error level. If the iteration fails to converge within the alloted iterations, an error message is printed. The subroutine JacobiIteration performs a single update of the solution

1. **No forcing**
   During the development phase of the program, use $f = 0$, $u_1 = 1$, and $u_{M+1} = M + 1$. The solution is then simply a straight line and is given by $u_i = M(x_i - a) + 1$, and the numerical solution should yield this exact solution. Use the interval $-1 \leq x \leq 1$.

2. **Simple forcing**
   Once the code is working try the code on the following problem:

$$u(\pm 1) = \pm 6, \quad f = 6x \tag{8}$$

   whose solution is $u = 6x^3$.

3. **Non-trivial problem**
   Once the code is working try your program on the following case:

$$\begin{aligned} u(\pm 1) &= \cos\left(\pi e^{\pm 1}\right) \\ f &= -\pi e^x \left[e^x \cos(\pi e^x) + \sin(\pi e^x)\right] \\ u(x) &= \cos\left(\pi e^x\right) \end{aligned}$$

   The numerical solution is now only approximate. Try solving the equations using 25, 50, 100, 200, 400, and 800 points. For each case record the error committed, and the number of iterations required to achieve convergence. Verify that the method is indeed second order accurate.

2 The 2D Poisson solver

Figure 4 refers to the two-dimensional computational grid. The finite difference approximation takes the form:

$$\frac{u_{i-1,j} - 2u_{i,j} + u_{i+1,j}}{\Delta x^2} + \frac{u_{i,j-1} - 2u_{i,j} + u_{i,j+1}}{\Delta y^2} = f_{i,j}, \quad 2 \leq i \leq M, \quad 2 \leq j \leq N \tag{9}$$

Again the unknown are located strictly in the interior of grid since u is known at the boundaries from the boundary conditions, and hence there are $(M-1)(N-1)$ unknowns. The iterative update of the Jacobi iteration can now be written as:

$$u_{i,j}^{(n+1)} = \frac{\left(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)}\right)\Delta y^2 + \left(u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)}\right)\Delta x^2 - \Delta x^2 \Delta y^2 f_{i,j}}{2\left(\Delta x^2 + \Delta y^2\right)} \tag{10}$$

Again it can be shown that the number of iterations needed to reach a tolerance level scales as $K \sim \max(M^2, N^2) \ln \epsilon$.

The programming tasks required is now to upgrade your one-dimensional iterative solver to two-dimensions following the same steps outlined before.

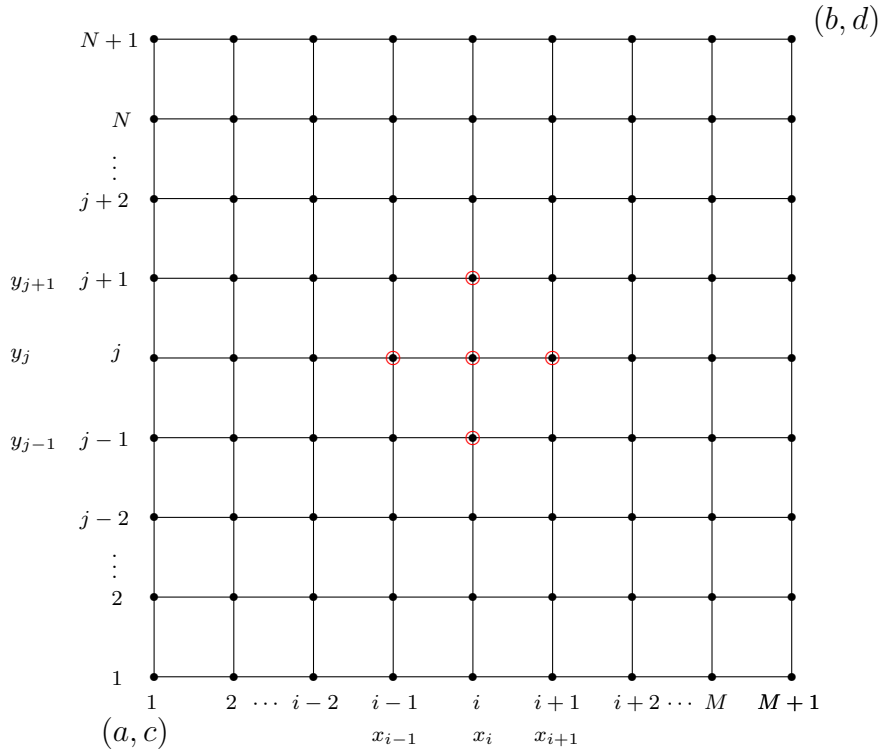

Figure 4: 2D finite difference grid for the interval $a \leq x \leq b$ using a uniform grid spacing $\Delta x = x_{i+1} - x_i = (b-a)/M$, and $\Delta y = y_{j+1} - y_j = (d-c)/N$. The nodes must now be referred to with a 2-integer index (i, j). The stencil for one of the computational grids is shown in red.

- **Development phase**
  Use a trivial case where the exact solution is just linear in each of the coordinate, say $u = x + 2 * y - 3z$, $f = 0$, to test your code during development. Another interesting solution is $u = x^2 - y^2$. Use the domain $|x| \leq 2$, and $|y| \leq 1$ with $M = 8$ and $N = 5$. The boundary conditions can be lifted from the exact solution.

- **Experimentation phase**
  For this case the problem's data, including the exact solution, is given by

$$u = \cos\left(\frac{\pi}{2}e^{(y-x)}\right) + \sin\left(\frac{\pi}{2}e^{(x+y)}\right) \tag{11}$$

$$\begin{aligned} f &= \pi e^{(x+y)}\left[\cos\left(\frac{\pi}{2}e^{(x+y)}\right) - \frac{\pi}{2}e^{(x+y)}\sin\left(\frac{\pi}{2}e^{(x+y)}\right)\right] \\ &- \pi e^{(y-x)}\left[\sin\left(\frac{\pi}{2}e^{(y-x)}\right) + \frac{\pi}{2}e^{(y-x)}\cos\left(\frac{\pi}{2}e^{(y-x)}\right)\right] \end{aligned} \tag{12}$$

  Use the same geometry as for the earlier case, but experiment with changing the number of points. This is somewhat of a demanding problem as the solution starts oscillating fast as one approaches the north-eastern corner of

the domain, one can anticipate that to model a wavelength accurately at least 8 points are needed, and hence $\Delta y > 0.025$. Again, confirm the second order convergence rate using (100×50), (200×100), (400×200), (800×400) and (1600×800) cells. Record the number of iterations needed to achieve convergence, and the CPU-time consumed.

3 The 3D Poisson solver

This part is optional, and does not require much work given that its a simple extension of the 2D code. It however drives home the message that the number of unknowns, and hence the time to solution, grows very fast in three-dimensions. The 3D stencil for the finite difference approximation of the Laplace operator, ∇^2, include 7 points on the computational grid, and is given by:

$$\begin{aligned} \frac{u_{i-1,j,k} - 2u_{i,j,k} + u_{i+1,j,k}}{\Delta x^2} + \frac{u_{i,j-1,k} - 2u_{i,j,k} + u_{i,j+1,k}}{\Delta y^2} \quad & + \\ \frac{u_{i,j,k-1} - 2u_{i,j,k} + u_{i,j,k+1}}{\Delta z^2} \quad &= \quad f_{i,j,k} \\ (2,2,2) \leq (i,j,k) \leq (M,N,P) & \end{aligned} \tag{13}$$

where P is the number of intervals in the z-direction. Again we have assumed that Dirichlet boundary conditions are applied on all boundaries $i = 1, M$, $j = 1, N$, and $k = 1, P$. The total number of unknowns is then $(M-1)(N-1)(P-1)$. Notice that the number of unknowns grows very quickly in 3D; for example, a $25 \times 25 \times 25$ cell discretization will have $(25-1)^3$ unknowns or 13,824 unknowns. The Jacobi iterative solution takes the form:

$$\begin{aligned} u_{i,j,k}^{(n+1)} &= a_x \left(u_{i-1,j,k}^{(n)} + u_{i+1,j,k}^{(n)} \right) + a_y \left(u_{i,j-1,k}^{(n)} + u_{i,j+1,k}^{(n)} \right) + a_z \left(u_{i,j,k-1}^{(n)} + u_{i,j,k+1}^{(n)} \right) \\ &- a_f f_{i,j,k} \end{aligned} \tag{14}$$

$$a_x = \frac{\Delta y^2 \Delta z^2}{2\left(\Delta x^2 + \Delta y^2 + \Delta z^2\right)} \tag{15}$$

$$a_y = \frac{\Delta z^2 \Delta x^2}{2\left(\Delta x^2 + \Delta y^2 + \Delta z^2\right)} \tag{16}$$

$$a_z = \frac{\Delta x^2 \Delta y^2}{2\left(\Delta x^2 + \Delta y^2 + \Delta z^2\right)} \tag{17}$$

$$a_f = \frac{\Delta x^2 \Delta y^2 \Delta z^2}{2\left(\Delta x^2 + \Delta y^2 + \Delta z^2\right)} \tag{18}$$