5 Poisson's Equation

5.1 Introduction

In this section, we shall discuss some simple numerical techniques for solving Poisson's equation:

$$\nabla^2 u(\mathbf{r}) = v(\mathbf{r}). \tag{5.1}$$

Here, $u(\mathbf{r})$ is usually some sort of potential, whereas $v(\mathbf{r})$ is a source term. The solution to the above equation is generally required in some simply-connected volume V bounded by a closed surface S. There are two main types of boundary conditions to Poisson's equation. In so-called *Dirichlet* boundary conditions, the potential u is specified on the bounding surface S. In so-called *Neumann* boundary conditions, the normal gradient of the potential $\nabla u \cdot d\mathbf{S}$ is specified on the bounding surface.

Poisson's equation is of particular importance in electrostatics and Newtonian gravity. In electrostatics, we can write the electric field $\mathbf{E}$ in terms of an electric potential ϕ:

$$\mathbf{E} = -\nabla \phi. \tag{5.2}$$

The potential itself satisfies Poisson's equation:

$$\nabla^2 \phi = -\frac{\rho}{\epsilon_0}, \tag{5.3}$$

where $\rho(\mathbf{r})$ is the charge density, and ϵ_0 the permittivity of free-space. In Newtonian gravity, we can write the force $\mathbf{f}$ exerted on a unit test mass in terms of a gravitational potential ϕ:

$$\mathbf{f} = -\nabla \phi. \tag{5.4}$$

The potential satisfies Poisson's equation:

$$\nabla^2 \phi = 4\pi^2 \, G \, \rho, \tag{5.5}$$

where $\rho(\mathbf{r})$ is the mass density, and G the universal gravitational constant.

5.2 1-D Problem with Dirichlet Boundary Conditions

As a simple test case, let us consider the solution of Poisson's equation in one dimension. Suppose that

$$\frac{d^2u(x)}{dx^2} = v(x), \tag{5.6}$$

for $x_l \leq x \leq x_h$, subject to the Dirichlet boundary conditions $u(x_l) = u_l$ and $u(x_h) = u_h$.

As a first step, we divide the domain $x_l \leq x \leq x_h$ into equal segments whose vertices are located at the grid-points

$$x_i = x_l + \frac{i\,(x_h - x_l)}{N+1}, \tag{5.7}$$

for $i = 1, N$. The boundaries, x_l and x_h, correspond to $i = 0$ and $i = N+1$, respectively.

Next, we discretize the differential term d^2u/dx^2 on the grid-points. The most straightforward discretization is

$$\frac{d^2u(x_i)}{dx^2} = \frac{u_{i-1} - 2\,u_i + u_{i+1}}{(\delta x)^2} + O(\delta x)^2. \tag{5.8}$$

Here, $\delta x = (x_h - x_l)/(N+1)$, and $u_i \equiv u(x_i)$. This type of discretization is termed a *second-order, central difference* scheme. It is "second-order" because the truncation error is $O(\delta x)^2$, as can easily be demonstrated via Taylor expansion. Of course, an nth order scheme would have a truncation error which is $O(\delta x)^n$. It is a "central difference" scheme because it is symmetric about the central grid-point, x_i. Our discretized version of Poisson's equation takes the form

$$u_{i-1} - 2\,u_i + u_{i+1} = v_i\,(\delta x)^2, \tag{5.9}$$

for $i = 1, N$, where $v_i \equiv v(x_i)$. Furthermore, $u_0 = u_l$ and $u_{N+1} = u_h$.

It is helpful to regard the above set of discretized equations as a matrix equation. Let $\mathbf{u} = (u_1, u_2, \cdots, u_N)$ be the vector of the u-values, and let

$$\mathbf{w} = [v_1\,(\delta x)^2 - u_l,\, v_2\,(\delta x)^2,\, v_3\,(\delta x)^2,\, \cdots,\, v_{N-1}\,(\delta x)^2,\, v_N\,(\delta x)^2 - u_h] \tag{5.10}$$

be the vector of the source terms. The discretized equations can be written as:

$$\mathbf{M}\,\mathbf{u} = \mathbf{w}. \tag{5.11}$$

The matrix $\mathbf{M}$ takes the form

$$\mathbf{M} = \begin{pmatrix} -2 & 1 & 0 & 0 & 0 & 0 \\ 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \\ 0 & 0 & 0 & 0 & 1 & -2 \end{pmatrix} \tag{5.12}$$

for $N = 6$. The generalization to other N values is fairly obvious. Matrix $\mathbf{M}$ is termed a *tridiagonal* matrix, since only those elements which lie on the three leading diagonals are non-zero.

The formal solution to Eq. (5.11) is

$$\mathbf{u} = \mathbf{M}^{-1}\,\mathbf{w}, \tag{5.13}$$

where $\mathbf{M}^{-1}$ is the inverse matrix to $\mathbf{M}$. Unfortunately, the most efficient general purpose algorithm for inverting an $N \times N$ matrix—namely, Gauss-Jordan elimination with partial pivoting—requires $O(N^3)$ arithmetic operations. It is fairly clear that this is a disastrous scaling for finite-difference solutions of Poisson's equation. Every time we doubled the resolution (*i.e.*, doubled the number of grid-points) the required cpu time would increase by a factor of about eight. Consequently, adding a second dimension (which effectively requires the number of grid-points to be squared) would be prohibitively expensive in terms of cpu time. Fortunately, there is a well-known trick for inverting an $N \times N$ *tridiagonal* matrix which only requires $O(N)$ arithmetic operations.

Consider a general $N \times N$ tridiagonal matrix equation $\mathbf{M}\,\mathbf{u} = \mathbf{w}$. Let $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ be the vectors of the left, center and right diagonal elements of the matrix, respectively Note that a_1 and c_N are undefined, and can be conveniently set to zero. Our matrix equation can now be written

$$a_i\,u_{i-1} + b_i\,u_i + c_i\,u_{i+1} = w_i, \tag{5.14}$$

for $i = 1, N$. Let us search for a solution of the form

$$u_{i+1} = x_i\, u_i + y_i. \tag{5.15}$$

Substitution into Eq. (5.14) yields

$$a_i\, u_{i-1} + b_i\, u_i + c_i\, (x_i\, u_i + y_i) = w_i, \tag{5.16}$$

which can be rearranged to give

$$u_i = -\frac{a_i\, u_{i-1}}{b_i + c_i\, x_i} + \frac{w_i - c_i\, y_i}{b_i + c_i\, x_i}. \tag{5.17}$$

However, if Eq. (5.15) is general then we can write $u_i = x_{i-1}\, u_{i-1} + y_{i-1}$. Comparison with the previous equation yields

$$x_{i-1} = -\frac{a_i}{b_i + c_i\, x_i}, \tag{5.18}$$

and

$$y_{i-1} = \frac{w_i - c_i\, y_i}{b_i + c_i\, x_i}. \tag{5.19}$$

We can now solve our tridiagonal matrix equation in two stages. In the first stage, we scan *up* the leading diagonal from $i = N$ to 1 using Eqs. (5.18) and (5.19). Thus,

$$x_{N-1} = -\frac{a_N}{b_N}, \qquad y_{N-1} = \frac{w_N}{b_N}, \tag{5.20}$$

since $c_N = 0$. Furthermore,

$$x_i = -\frac{a_{i+1}}{b_{i+1} + c_{i+1}\, x_{i+1}}, \qquad y_i = \frac{w_{i+1} - c_{i+1}\, y_{i+1}}{b_{i+1} + c_{i+1}\, x_{i+1}} \tag{5.21}$$

for $i = N-2, 1$. Finally,

$$x_0 = 0, \qquad y_0 = \frac{w_1 - c_1\, y_1}{b_1 + c_1\, x_1}, \tag{5.22}$$

since $a_1 = 0$. We have now defined all of the x_i and y_i. In the second stage, we scan *down* the leading diagonal from $i = 0$ to $N-1$ using Eq. (5.15). Thus,

$$u_1 = y_0, \tag{5.23}$$

since $x_0 = 0$, and

$$u_{i+1} = x_i\, u_i + y_i \tag{5.24}$$

for $i = 1, N-1$. We have now inverted our tridiagonal matrix equation using $O(N)$ arithmetic operations.

Clearly, we can use the above algorithm to invert Eq. (5.11), with the source terms specified in Eq. (5.10), and the diagonals of matrix $\mathbf{M}$ given by $a_i = 1$ for $i = 2, N$, plus $b_i = -2$ for $i = 1, N$, and $c_i = 1$ for $i = 1, N-1$.

5.3 An Example Tridiagonal Matrix Solving Routine

Listed below is an example tridiagonal matrix solving routine which utilizes the `Blitz++` library (see Sect. 2.20).

```
// Tridiagonal.cpp

// Function to invert tridiagonal matrix equation.
// Left, centre, and right diagonal elements of matrix
// stored in arrays a, b, c, respectively.
// Right-hand side stored in array w.
// Solution written to array u.

// Matrix is NxN. Arrays a, b, c, w, u assumed to be of extent N+2,
// with redundant 0 and N+1 elements.

#include <blitz/array.h>

using namespace blitz;

void Tridiagonal (Array<double,1> a, Array<double,1> b, Array<double,1> c,
                  Array<double,1> w, Array<double,1>& u)
{
  // Find N. Declare local arrays.
  int N = a.extent(0) - 2;
  Array<double,1> x(N), y(N);

  // Scan up diagonal from i = N to 1
  x(N-1) = - a(N) / b(N);
  y(N-1) = w(N) / b(N);
  for (int i = N-2; i > 0; i--)
```

```
    {
      x(i) = - a(i+1) / (b(i+1) + c(i+1) * x(i+1));
      y(i) = (w(i+1) - c(i+1) * y(i+1)) / (b(i+1) + c(i+1) * x(i+1));
    }
  x(0) = 0.;
  y(0) = (w(1) - c(1) * y(1)) / (b(1) + c(1) * x(1));

  // Scan down diagonal from i = 0 to N-1
  u(1) = y(0);
  for (int i = 1; i < N; i++)
    u(i+1) = x(i) * u(i) + y(i);
}
```

5.4 1-D problem with Mixed Boundary Conditions

Previously, we solved Poisson's equation in one dimension subject to Dirichlet boundary conditions, which are the simplest conceivable boundary conditions. Let us now consider the following much more general set of boundary conditions:

$$\alpha_l\, u(x) + \beta_l \frac{du(x)}{dx} = \gamma_l, \tag{5.25}$$

at $x = x_l$, and

$$\alpha_h\, u(x) + \beta_h \frac{du(x)}{dx} = \gamma_h, \tag{5.26}$$

at $x = x_h$. Here, α_l, β_l, *etc.*, are known constants. The above boundary conditions are termed *mixed*, since they are a mixture of Dirichlet and Neumann boundary conditions.

Using the previous notation, the discretized versions of Eq. (5.25) and (5.26) are:

$$\alpha_l\, u_0 + \beta_l \frac{u_1 - u_0}{\delta x} = \gamma_l, \tag{5.27}$$

$$\alpha_h\, u_{N+1} + \beta_h \frac{u_{N+1} - u_N}{\delta x} = \gamma_h, \tag{5.28}$$

respectively. The above expressions can be rearranged to give

$$u_0 = \frac{\gamma_l\, \delta x - \beta_l\, u_1}{\alpha_l\, \delta x - \beta_l}, \tag{5.29}$$

$$u_{N+1} = \frac{\gamma_h \, \delta x + \beta_h \, u_N}{\alpha_h \, \delta x + \beta_h}. \tag{5.30}$$

Using Eqs. (5.8), (5.29), and (5.30), the problem can be reduced to a tridiagonal matrix equation $\mathbf{M}\,\mathbf{u} = \mathbf{w}$, where the left, center, and right diagonals of $\mathbf{M}$ possess the elements $a_i = 1$ for $i = 2, N$, with

$$b_1 = -2 - \frac{\beta_l}{\alpha_l \, \delta x - \beta_l}, \tag{5.31}$$

and $b_i = -2$ for $i = 2, N-1$, plus

$$b_N = -2 + \frac{\beta_h}{\alpha_h \, \delta x + \beta_h}, \tag{5.32}$$

and $c_i = 1$ for $i = 1, N-1$, respectively. The elements of the right-hand side are

$$w_1 = v_1 \, (\delta x)^2 - \frac{\gamma_l \, \delta x}{\alpha_l \, \delta x - \beta_l}, \tag{5.33}$$

with $w_i = v_i \, (\delta x)^2$ for $i = 2, N-1$, and

$$w_N = v_N \, (\delta x)^2 - \frac{\gamma_h \, \delta x}{\alpha_h \, \delta x + \beta_h}. \tag{5.34}$$

Our tridiagonal matrix equation can be inverted using the algorithm discussed previously.

5.5 An Example 1-D Poisson Solving Routine

Listed below is an example 1-d Poisson solving routine which utilizes the previously listed tridiagonal matrix solver and the `Blitz++` library (see Sect. 2.20).

```
// Poisson1D.cpp

// Function to solve Poisson's equation in 1-d:

//  d^2 u / dx^2 = v  for  xl <= x <= xh
```