

Multigrid Methods

Practical multigrids methods were first introduced by Brandt, and independently by Hackbusch, in the 1970s. They can solve elliptic partial differential equations discretized on a lattice with N points in $\mathcal{O}(N)$ operations. They are extremely efficient compared with conventional method: for example, the classic Gauss-Seidel iteration method takes $\mathcal{O}(N^2)$ operations in two dimensions!

Multigrid methods are essentially iterative methods in which the grid spacing is adapted at each step, for example by increasing or reducing the number of points in each dimension by a factor of 2.

Multigrid methods have been used extensively in solving in solving parabolic equations, eigenvalue problems and integral equations, in addition to elliptic PDEs. They have also been combined with Monte Carlo methods to solve lattice problems in statistical physics and quantum field theory.

Elliptic Partial Differential Equations

Partial differential equations are typically used to describe the behavior of *continuous* media such as fluids and electromagnetic fields.

Poisson's equation in electrostatics

The electric field $\mathbf{E}$ due to a static charge density distribution $\rho(x, y, z)$ is determined by Gauss' Law:

$$\nabla \cdot \mathbf{E} = \frac{\rho(x, y, z)}{\epsilon_0} ,$$

where ϵ_0 is the permittivity constant. A static electric field can be derived from a scalar potential $\psi(x, y, z)$

$$\mathbf{E} = -\nabla\psi ,$$

which obeys Poisson's equation

$$\nabla^2\psi = \left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \psi = -\frac{\rho(x, y, x)}{\epsilon_0} .$$

This is an example of an *elliptic partial differential equation*.

Boundary conditions for a unique solution

Elliptic partial differential equations like Poisson's equation have unique solutions inside a region of space (which can be multiply connected) if the following types of boundary conditions are specified on the *closed* boundary of the region:

- the value of ψ on the bounding surface is specified (Dirichlet conditions), or
- the value of the normal derivative $\hat{\mathbf{n}} \cdot \nabla \psi$ on the bounding surface is specified (Neumann conditions), or
- periodic boundary conditions are imposed.

Discretization in two dimensions

Consider the 2-D form of Poisson's equation:

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \psi(x, y) = -\rho(x, y) ,$$

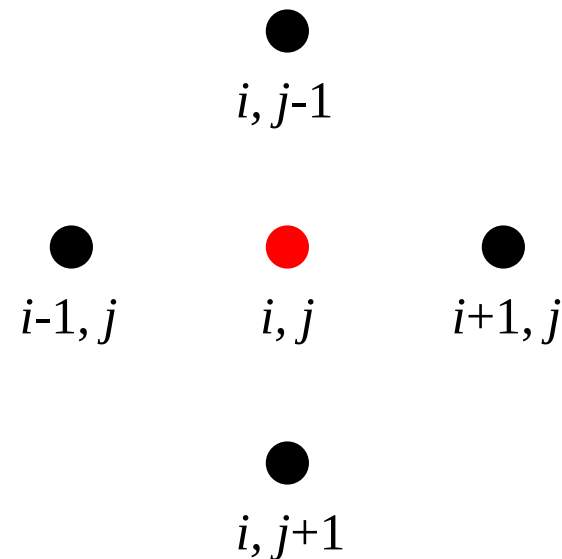
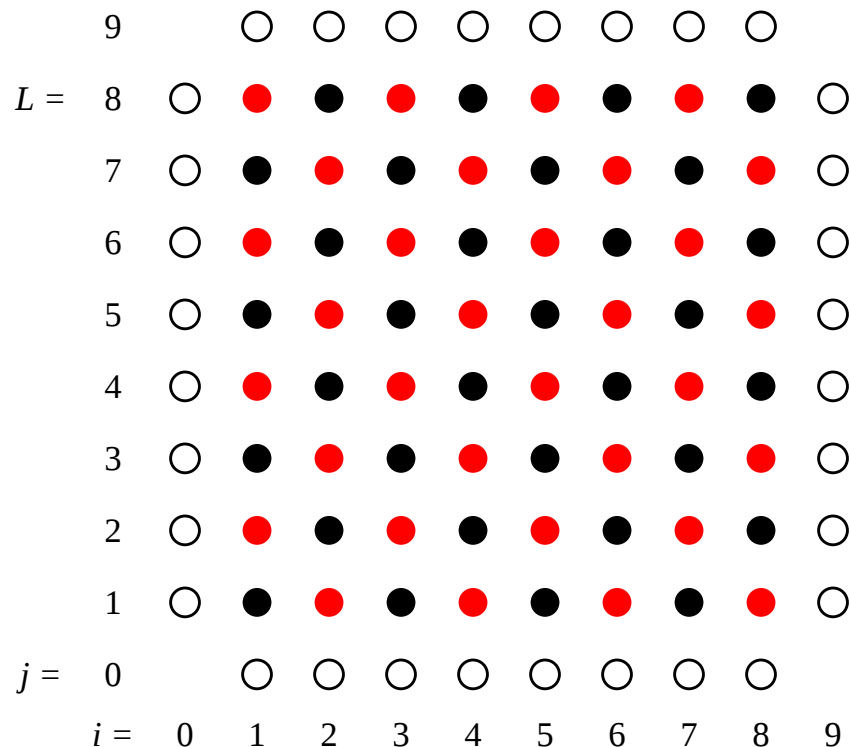
where we set $\epsilon_0 = 1$ for convenience. Let's solve this equation in the interior of a square region of unit side $0 \leq x, y \leq 1$. Introduce a grid of points separated by a lattice spacing h in the x and y directions:

$$x_i = ih , \quad i = 0, 1, \dots, L, L+1 , \quad y_j = jh , \quad j = 0, 1, \dots, L, L+1 .$$

There are L^2 *interior* points as shown in the figure, and the lattice spacing $h = 1/(L+1)$. The value of $\psi(x_i, y_j)$ at a lattice point is abbreviated $\psi_{i,j}$, and $\rho(x_i, y_j) = \rho_{i,j}$.

We next need a discrete form for the Laplacian operator. There are many possibilities, but the simplest symmetric finite-difference form is

$$\left(\frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \right) \psi(x_i, y_j) \simeq \frac{1}{h^2} [\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1} - 4\psi_{i,j}] = -\rho_{i,j} .$$



Note the following properties of this discretization:

- Each lattice site is connected to its 4 nearest neighbors.
- For later convenience, the interior sites are labeled even or odd depending on whether $i+j$ is even or odd. Even sites are colored black, and odd sites are colored red. Note that the 4 neighbors of a red site are black, and the 4 neighbors of a black site are red.
- The boundary sites are indicated using open circles. With the discretization formula above for the Laplacian operator, only those boundary sites that are nearest neighbors of an interior site need to be taken into consideration.

Jacobi's iterative method

Suppose we have found a solution of the discretized equation, then at each lattice site,

$$\psi_{i,j} = \frac{1}{4} [\psi_{i+1,j} + \psi_{i-1,j} + \psi_{i,j+1} + \psi_{i,j-1} + h^2 \rho_{i,j}] .$$

If we knew the right hand side, then we could compute the left hand side. Unfortunately, the right hand side involves ψ at the 4 neighboring points! Thus to find the solution, the L^2 equations must be solved simultaneously. Solving Poisson's equation is essentially a problem in linear algebra.

Jacobi's iterative method for solving these simultaneous equations is to start with a guess $\psi_{i,j}^0$ for the solution at the interior lattice points. Plugging this guess into the right hand sides yields $\psi_{i,j}^1$ at all lattice points. This procedure

$$\psi_{i,j}^{n+1} = \frac{1}{4} [\psi_{i+1,j}^n + \psi_{i-1,j}^n + \psi_{i,j+1}^n + \psi_{i,j-1}^n + h^2 \rho_{i,j}] , \quad n = 0, 1, 2, \dots$$

is repeated until (hopefully) the process converges to the correct solution.

Program for solving Poisson's equation in 2-D

The following program solves Poisson's equation with a charge q placed at the center of the square region.

```
// Poisson's equation in 2-D using Jacobi, Gauss-Seidel,  
//      or Successive Over Relaxation  
  
#include <cmath>  
#include <cstdlib>  
#include <iostream>  
#include <fstream>  
  
int L;                               // number of interior points in x and y
```

```
double **psi;           // potential to be found
double **rho;           // given charge density

double h;               // lattice spacing
double **psiNew;        // new potential after each step
int steps;              // number of iteration steps
double accuracy;        // desired accuracy in solution
double omega;           // overrelaxation parameter

void allocateStorage() {

    // allocate storage for lattice arrays
    int N = L + 2;       // interior points + 2 boundary points
    psi = new double* [N];
    psiNew = new double* [N];
    rho = new double* [N];
    for (int i = 0; i < N; i++) {
        psi[i] = new double [N];
        psiNew[i] = new double [N];
        rho[i] = new double [N];
    }
}

void initialize() {

    // zero all arrays
    for (int i = 0; i < L + 2; i++)
        for (int j = 0; j < L + 2; j++)
            psi[i][j] = psiNew[i][j] = rho[i][j] = 0;
```

```
h = 1 / double(L + 1);    // assume physical size in x and y = 1
double q = 10;             // point charge
int i = L / 2;             // center of lattice
rho[i][i] = q / (h * h);   // charge density

steps = 0;
}

void Jacobi() {

    // Jacobi algorithm for a single iterative step
    for (int i = 1; i <= L; i++)
        for (int j = 1; j <= L; j++)
            psiNew[i][j] = 0.25 * (psi[i - 1][j] + psi[i + 1][j] +
                                   psi[i][j - 1] + psi[i][j + 1] +
                                   h * h * rho[i][j]);
}
```

Convergence of the iteration

We need to decide when the solution has converged sufficiently. Since we presumably do not know the exact solution, one criterion is to ask that the approximate solution ceases to change significantly from one iteration to the next. The following function estimates the relative error as the average over all lattice sites of non-zero magnitudes of change.

```
double relativeError() {

    double error = 0;        // average relative error per lattice point
```

```

int n = 0;                                // number of non-zero differences

for (int i = 1; i <= L; i++)
for (int j = 1; j <= L; j++) {
    if (psiNew[i][j] != 0)
    if (psiNew[i][j] != psi[i][j]) {
        error += abs(1 - psi[i][j] / psiNew[i][j]);
        ++n;
    }
}
if (n != 0)
    error /= n;

return error;
}

```

Gauss-Seidel method

This is a modification of the Jacobi method, which can be shown to converge a little faster. Suppose we are sweeping the lattice in order of increasing i and j , then the left and lower neighbors of each lattice site will have been updated: why not use these (presumably) more accurate values in Jacobi's formula? This results in one form of the Gauss-Seidel algorithm:

$$\psi_{i,j}^{n+1} = \frac{1}{4} [\psi_{i+1,j}^n + \psi_{i-1,j}^{n+1} + \psi_{i,j+1}^n + \psi_{i,j-1}^{n+1} + h^2 \rho_{i,j}] \quad , \quad n = 0, 1, 2, \dots$$

```

void GaussSeidel() {

    // copy psi to psiNew

```

```

for (int i = 1; i <= L; i++)
for (int j = 1; j <= L; j++)
    psiNew[i][j] = psi[i][j];

// Gauss-Seidel update
for (int i = 1; i <= L; i++)
for (int j = 1; j <= L; j++)
    psiNew[i][j] = 0.25 * (psiNew[i - 1][j] + psiNew[i + 1][j] +
                          psiNew[i][j - 1] + psiNew[i][j + 1] +
                          h * h * rho[i][j]);
}

```

The Successive Over Relaxation (SOR) method

The Jacobi and Gauss-Seidel methods do not use the value of $\psi_{i,j}$ at the same lattice point in updating $\psi_{i,j}$. It turns out that the convergence of the iteration can be improved considerably by using a linear combination of the new and old solutions as follows:

$$\psi_{i,j}^{n+1} = (1 - \omega)\psi_{i,j}^n + \frac{\omega}{4} [\psi_{i+1,j}^n + \psi_{i-1,j}^{n+1} + \psi_{i,j+1}^n + \psi_{i,j-1}^{n+1} + h^2 \rho_{i,j}] \quad , \quad n = 0, 1, 2, \dots$$

The *over-relaxation parameter* ω can be tuned to optimize the convergence. It can be shown that:

- The method is convergent only for $0 < \omega < 2$.
- It is faster than Gauss-Seidel only if $1 < \omega < 2$.
- It converges fastest for a square lattice if

$$\omega \simeq \frac{2}{1 + \frac{\pi}{L}} \quad ,$$

where L is the number of lattice points in the x or y directions.

Note that SOR could also be implemented using the Jacobi prescription with only ψ^n values on the right hand side.

Checkerboard (red-black) updating

We will implement SOR with a variant of the Gauss-Seidel method. Since even sites only have odd neighbors, and odd sites only have even neighbors, we can update all of the even sites first. The updated even-site values are then used to update all of the odd sites.

```
void SuccessiveOverRelaxation() {  
  
    // update even sites  
    for (int i = 1; i <= L; i++)  
        for (int j = 1; j <= L; j++)  
            if ((i + j) % 2 == 0)  
                psiNew[i][j] = (1 - omega) * psi[i][j] + omega / 4 *  
                    (psi[i - 1][j] + psi[i + 1][j] +  
                     psi[i][j - 1] + psi[i][j + 1] +  
                     h * h * rho[i][j]);  
  
    // update odd sites  
    for (int i = 1; i <= L; i++)  
        for (int j = 1; j <= L; j++)  
            if ((i + j) % 2 != 0)  
                psiNew[i][j] = (1 - omega) * psi[i][j] + omega / 4 *  
                    (psiNew[i - 1][j] + psiNew[i + 1][j] +  
                     psiNew[i][j - 1] + psiNew[i][j + 1] +  
                     h * h * rho[i][j]);  
}
```

```
void iterate(void (*method)()) {

    while (true) {
        method();
        ++steps;
        double error = relativeError();
        if (error < accuracy)
            break;
        double **swap = psi;
        psi = psiNew;
        psiNew = swap;
    }
    cout << " Number of steps = " << steps << endl;
}

int main() {

    cout << " Iterative solution of Poisson's equation\n"
         << " -----\n";
    cout << " Enter number of interior points in x or y: ";
    cin >> L;

    allocateStorage();
    initialize();

    cout << " Enter desired accuracy in solution: ";
    cin >> accuracy;

    cout << " Enter 1 for Jacobi, 2 for Gauss Seidel, 3 for SOR: ";
```

```
int choice;
cin >> choice;
switch (choice) {
case 1:
    iterate(Jacobi);
    break;
case 2:
    iterate(GaussSeidel);
    break;
case 3:
    cout << " Enter overrelaxation parameter omega: ";
    cin >> omega;
    iterate(SuccessiveOverRelaxation);
    break;
default:
    cout << " Jacobi: " << endl;
    iterate(Jacobi);
    cout << " Gauss-Seidel: " << endl;
    initialize();
    iterate(GaussSeidel);
    omega = 2 / (1 + 4 * atan(1.0) / double(L));
    cout << " Successive Over Relaxation with theoretical optimum omega = "
        << omega << endl;
    initialize();
    iterate(SuccessiveOverRelaxation);
    break;
}

// write potential to file
```

```
cout << " Potential in file poisson.data" << endl;
ofstream file("poisson.data");
for (int i = 0; i < L + 2; i++) {
    double x = i * h;
    for (int j = 0; j < L + 2; j++) {
        double y = j * h;
        file << x << '\t' << y << '\t' << psi[i][j] << '\n';
    }
    file << '\n';
}
file.close();
}
```

Results from running the program

Iterative solution of Poisson's equation

Enter number of interior points in x or y: 50

Enter desired accuracy in solution: 0.001

Enter 1 for Jacobi, 2 for Gauss Seidel, 3 for SOR: 0

Jacobi:

Number of steps = 1012

Gauss-Seidel:

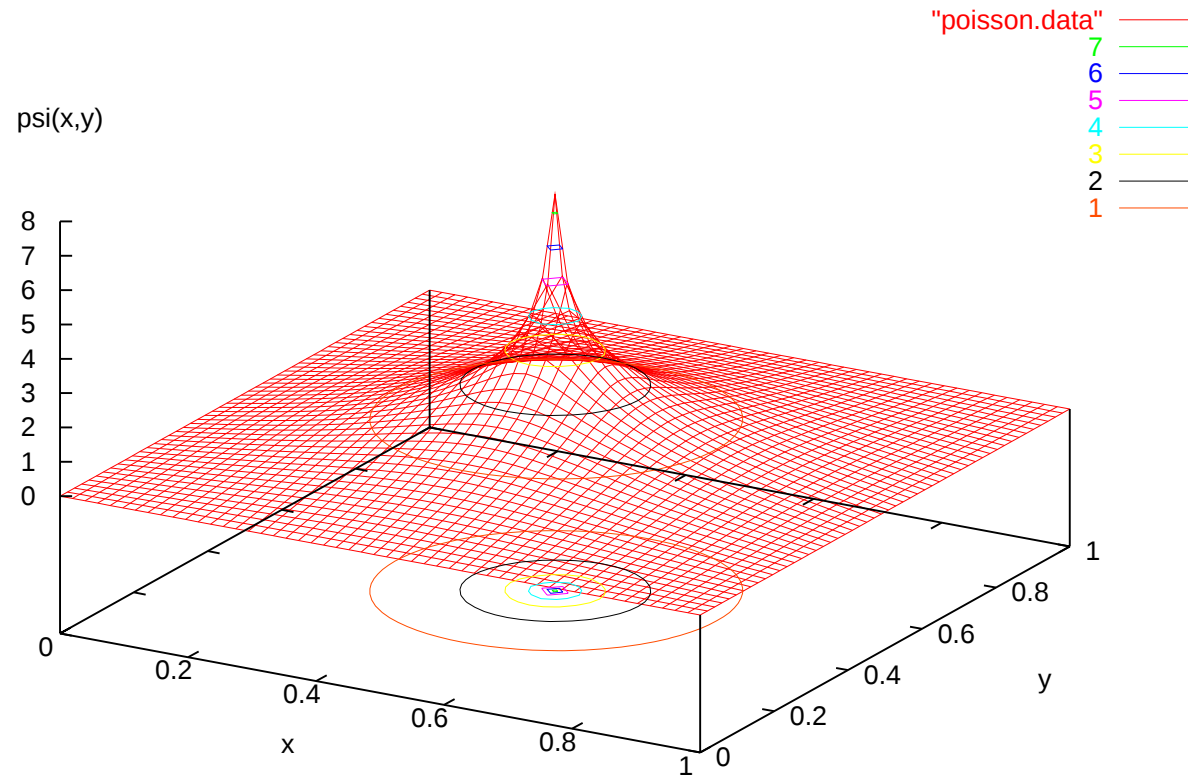
Number of steps = 506

Successive Over Relaxation with theoretical optimum $\omega = 1.88177$

Number of steps = 62

Potential in file poisson.data

Poisson's equation: $L = 50$, Point charge $q = 10$



According to *Numerical Recipes* §19.5, the number of iterations r required to reduce the overall error by a factor of 10^{-p} for Laplace's equation in 2-D is given by

$$r \simeq \begin{cases} \frac{1}{2}pL^2 & \text{for Jacobi's method} \\ \frac{1}{4}pL^2 & \text{for the Gauss-Seidel method} \\ \frac{1}{3}pL & \text{for SOR with } \omega \simeq 2/(1 + \pi/L) \end{cases} \quad \left(\begin{array}{l} \frac{1}{2} \times 3 \times 50^2 = 3,750 \\ \frac{1}{4} \times 3 \times 50^2 = 1,875 \\ \frac{1}{3} \times 3 \times 50 = 50 \end{array} \right)$$