

Multigrid program for Poisson's equation in 2-D

The following program `poisson-mg.cpp` implements a simple V-cycle multigrid method to find the electrostatic potential due to a point charge by solving Poisson's equation.

```
// Multigrid program for Poisson's equation in 2-D

#include <cmath>
#include <cstdio>
#include <fstream>
#include <iostream>

double accuracy;          // desired relative accuracy in solution
int L;                    // number of interior points in each dimension
```

A simple matrix class to simplify the programming

To code the multigrid algorithm, we need to compute and store various matrices representing the solution, the source function, the residual, and the correction at each level of coarser lattices with

$$2^{\ell-1} \rightarrow 2^{\ell-2} \rightarrow \dots \rightarrow 2^0 = 1$$

interior points in each direction.

To simplify allocating and freeing memory for all of these object, we define a simple C++ `Matrix` class as follows:

```
class Matrix {

public:
```

First define a set of three constructors:

- the default constructor constructs an empty matrix with no elements,
- the constructor `Matrix(int N)` constructs an $N \times N$ matrix,
- and, the *copy constructor* `Matrix(const Matrix& mat)` is defined so memory is properly allocated and freed when temporary objects are constructed by the compiler (for example in function arguments).

Also defined is a destructor which correctly frees memory allocated by any constructor.

```
Matrix() { N = 0; m = 0; }    // default constructor

Matrix(int N) {                // allocate new NxN matrix
    this->N = N;
    allocateStorage();
}

Matrix(const Matrix& mat) {    // copy constructor
    N = mat.N;
    allocateStorage();
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            m[i][j] = mat[i][j];
}

~Matrix() {                    // destructor
    freeStorage();
}
```

Next, a convenient operator is defined which correctly manages memory when one `Matrix` is assigned to

another. Note that nothing happens with self-assignment, and the `Matrix` is re-sized if the dimensions do not agree.

```
Matrix& operator=(const Matrix& mat) {
    if (this != &mat) {
        if (N != mat.N) {
            freeStorage();
            N = mat.N;
            allocateStorage();
        }
        for (int i = 0; i < N; i++)
            for (int j = 0; j < N; j++)
                m[i][j] = mat[i][j];
    }
    return *this;
}
```

Next, define a couple of convenience functions:

```
void zero() {                // set matrix elements to zero
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            m[i][j] = 0;
}

int rowsCols() {             // return the number of rows or columns
    return N;
}
```

The following two operators allow access to `Matrix` elements using array notation `[i][j]`.

```
double* operator[] (const int i) { return m[i]; }
const double* operator[] (const int i) const { return m[i]; }
```

The private section of the class holds the number of rows or columns N and a 1-D array of N pointers ****m**. The actual matrix elements are held in a 1-D array of $N * N$ doubles, with **m[i]** pointing to the start of the i -th row.

```
private:
    int N;                // matrix is NxN
    double **m;           // the matrix elements

    void allocateStorage() {
        m = new double* [N];
        m[0] = new double [N * N];
        for (int i = 1; i < N; i++)
            m[i] = m[i - 1] + N;
    }

    void freeStorage() {
        if (m != 0) {
            delete [] m[0];
            delete [] m;
        }
    }
};
```

Continuing with the program

The **Matrix** class can now be used to declare **Matrix** instances. This declaration produces zero matrices, which will later be assigned to $(L + 2) \times (L + 2)$ matrices.

```
Matrix psi;           // solution to be found
Matrix psiNew;        // approximate solution after 1 iteration
Matrix rho;           // given source function

double h;             // step size
int steps;            // number of iteration steps

void initialize() {

    // check that L is a power of 2 as required by multigrid
    int pow2 = 1;
    while (pow2 < L)
        pow2 *= 2;
    if (pow2 != L) {
        L = pow2;
        cout << " Setting L = " << L << " (must be a power of 2)" << endl;
    }

    // create (L+2)x(L+2) matrices and zero them
    psi = Matrix(L + 2);
    psi.zero();
    psiNew = rho = psi;

    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;
```

```
}
```

Here is an application of `Matrix` objects passed by reference to a function: this avoids creating local copies of the objects. The function implements checkerboard updating for the pre- and post-smoothing multigrid steps.

```
void GaussSeidel(double h, Matrix& u, const Matrix& f) {  
  
    int L = u.rowsCols() - 2;  
  
    // use checkerboard updating  
    for (int color = 0; color < 2; color++)  
        for (int i = 1; i <= L; i++)  
            for (int j = 1; j <= L; j++)  
                if ((i + j) % 2 == color)  
                    u[i][j] = 0.25 * (u[i - 1][j] + u[i + 1][j] +  
                                     u[i][j - 1] + u[i][j + 1] +  
                                     h * h * f[i][j]);  
}
```

Implementing the recursive two-grid algorithm

The following function implements the recursive two-grid algorithm discussed in the previous lecture. The recursion results in a V-cycle.

```
void twoGrid(double h, Matrix& u, Matrix& f) {  
  
    // solve exactly if there is only one interior point  
    int L = u.rowsCols() - 2;  
    if (L == 1) {
```

```

    u[1][1] = 0.25 * (u[0][1] + u[2][1] + u[1][0] + u[1][2] +
                      h * h * f[1][1]);
    return;
}

// do a few pre-smoothing Gauss-Seidel steps
int nPreSmooth = 3;
for (int i = 0; i < nPreSmooth; i++)
    GaussSeidel(h, u, f);

```

Recall the definition of the *residual*:

$$r = \nabla^2 u + f .$$

```

// find the residual
Matrix r = f;
for (int i = 1; i <= L; i++)
    for (int j = 1; j <= L; j++)
        r[i][j] += (u[i + 1][j] + u[i - 1][j] +
                     u[i][j + 1] + u[i][j - 1] - 4 * u[i][j]) / (h * h);

```

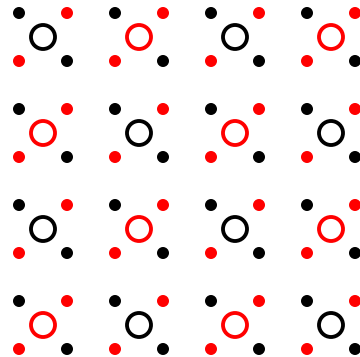
A simple cell-centered coarsening is used to restrict the residual to the coarser grid:

$$R_{I,J} = \frac{1}{4} [r_{i,j} + r_{i+1,j} + r_{i,j+1} + r_{i+1,j+1}] , \quad i = 2(I-1) , \quad j = 2(J-1) .$$

```

// restrict residual to coarser grid
int L2 = L / 2;
Matrix R(L2 + 2);

```



```

for (int I = 1; I <= L2; I++) {
    int i = 2 * I - 1;
    for (int J = 1; J <= L2; J++) {
        int j = 2 * J - 1;
        R[I][J] = 0.25 * (r[i][j] + r[i + 1][j] + r[i][j + 1] +
                           r[i + 1][j + 1]);
    }
}

```

The residual is the source function to compute the correction on the coarser grid by solving

$$\nabla^2 V = -R(x, y) ,$$

with the initial guess $V(x, y) = 0$. This is done by calling `twoGrid` recursively.

```

// initialize correction V on coarse grid to zero
Matrix V(L2 + 2);
V.zero();

// call twoGrid recursively

```

```
twoGrid(2 * h, V, R);
```

Prolongation of the correction V from the coarser grid to the finer is done using simple injection

$$v_{i,j} = v_{i+1,j} = v_{i,j+1} = v_{i+1,j+1} = V_{I,J} , \quad i = 2(I - 1) , \quad j = 2(J - 1) .$$

```
// prolongate V to fine grid using simple injection
Matrix v(L + 2);
v.zero();
for (int I = 1; I <= L2; I++) {
    int i = 2 * I - 1;
    for (int J = 1; J <= L2; J++) {
        int j = 2 * J - 1;
        v[i][j] = v[i + 1][j] = v[i][j + 1] = v[i + 1][j + 1] = V[I][J];
    }
}
```

Finally, the correction is applied, and followed by a few post-smoothing Gauss-Seidel steps.

```
// correct u
for (int i = 1; i <= L; i++)
for (int j = 1; j <= L; j++)
    u[i][j] += v[i][j];

// do a few post-smoothing Gauss-Seidel steps
int postSmooth = 3;
for (int i = 0; i < postSmooth; i++)
    GaussSeidel(h, u, f);
```

```
}
```

Steering the calculation

A relative error is defined as in the program `poisson.cpp`, and the main function is defined in a similar way.

```
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;  
}  
  
int main() {  
  
    cout << " Multigrid solution of Poisson's equation\n"  
         << " -----\n";
```

```
cout << " Enter desired accuracy in the solution: ";
cin >> accuracy;
cout << " Enter number of interior points in x or y: ";
cin >> L;

initialize();
while (true) {
    psiNew = psi;
    twoGrid(h, psi, rho);
    ++steps;
    double error = relativeError();
    cout << " Step No.  " << steps << "\tError = " << error << endl;
    if (steps > 1 && error < accuracy)
        break;
}

// write potential to file
cout << " Potential in file poisson-mg.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

Multigrid solution of Poisson's equation

Enter desired accuracy in the solution: 0.001

Enter number of interior points in x or y: 50

Setting L = 64 (must be a power of 2)

Step No. 15 Error = 0.000672391

Potential in file poisson-mg.data

These results can be compared with the methods used in `poisson.cpp`:

Iterative solution of Poisson's equation

Enter number of interior points in x or y: 64

Enter desired accuracy in solution: 0.001

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

Jacobi:

Number of steps = 1336

Gauss-Seidel:

Number of steps = 668

Successive Over Relaxation with theoretical optimum $\omega = 1.90642$

Number of steps = 76

Potential in file poisson.data