Sample Code - Multigrid Algorithm

Junlin Wu

The C++ code below is an implementation of the multigrid algorithm, which corresponds to the project that I was working on part-time. The goal of this code is to incorporate into the fluid simulation in paper *Real-Time Fluid Dynamics for Games* for solving the large scale Poisson problem. Since classical iterative method that is used in the original paper is inefficient in removing low frequency errors, multigrid algorithm makes an improvement through using multilevel method to transfer low frequency errors to a coarser level, where they become high frequency.

Listing 1: Multigrid Class

```
#include <iostream>
#include <vector>
using namespace std;
#define IX(i,j) ((i)+(N+1)*(j))
#define IXC(i,j) ((i)+(N/2 +1)*(j))
#define FOR_EACH_CELL for(i=1; i<=N-1; i++) {for(j=1; j<=N-1; j++){
#define END_FOR }}

/*
This code solves the Poisson problem -(u_xx + u_yy) = f in a unit square with boundaries 0;
The problem is solved on a (N+1) * (N+1) grid;
alpha_down is the number of steps for pre-smoothing;
alpha_c is the number of smoothing steps for the coarsest level;
alpha_up is the number of steps for post-smoothing;
num_lev is the preset number of levels for multigrid algorithm
*/

class MultiGrid_Class
{
    public:
        void Smooth_relax(vector <float> &u, vector <float> &f, int mu, int N);
        vector <float> Operator_A(vector <float> &u, int N);
        vector <float> Restriction(vector <float> &u, int N);
        vector <float> Interpolation(vector <float> &uc, int N);
        vector <float> MultiGrid(vector <float> &f, int alpha_down, int alpha_up, int
            alpha_c, int num_lev, int N);
};
```

Listing 2: Smooth_relax function

```
void MultiGrid_Class::Smooth_relax(vector <float> &u, vector <float> &f, int mu, int N){
    /*2D weighted Jacobi, weight = 0.8*/
    int i,j,k;
    float w = 0.8;
    float h = 1.0/N;
    for(k = 1; k <= mu; k++){
        FOR_EACH_CELL
            u[IX(i,j)] = (1-w) * u[IX(i,j)] + (w/4) * (u[IX(i-1,j)] + u[IX(i+1,j)] + u[IX(i,
                j-1)] + u[IX(i,j+1)] + h*h*f[IX(i,j)]);
        END_FOR
    }
```

```
}
```

Listing 3: Operator_A function

```
vector <float> MultiGrid_Class::Operator_A(vector <float> &u, int N){
    int i,j;
    float h = 1.0/N;
    vector <float> y(u.size(), 0.0);
    FOR_EACH_CELL
        y[IX(i,j)] = ((-u[IX(i-1,j)] + 2*u[IX(i,j)] - u[IX(i+1,j)]) + (-u[IX(i,j-1)] + 2*u[
            IX(i,j)] - u[IX(i,j+1)]))/(h*h);
    END_FOR
    return y;
}
```

Listing 4: Restriction function

```
vector <float> MultiGrid_Class::Restriction(vector <float> &u, int N){
    int i,j;
    int size_c = (N/2 +1) * (N/2 + 1);
    vector <float> uc(size_c, 0.0);
    for(i = 0; i <= N/2; i++){
        for (j = 0; j <= N/2; j++){
            uc[IXC(i,j)] = u[IX(2*i, 2*j)];
        }
    }
    return uc;
}
```

Listing 5: Interpolation function

```
vector <float> MultiGrid_Class::Interpolation(vector <float> &uc, int N){
    int size = (N+1) * (N+1);
    int i,j;
    vector <float> u(size, 0.0);
    for(i = 0; i<= N/2 -1; i++){
        for(j = 0; j<= N/2 -1; j++){
            u[IX(2*i, 2*j)] = uc[IXC(i,j)];
            u[IX(2*i+1, 2*j)] = (uc[IXC(i,j)] + uc[IXC(i+1,j)])/2;
            u[IX(2*i, 2*j+1)] = (uc[IXC(i,j)] + uc[IXC(i,j+1)])/2;
            u[IX(2*i+1, 2*j+1)] = (uc[IXC(i,j)] + uc[IXC(i+1,j)] +uc[IXC(i,j+1)] + uc[IXC(i
                +1,j+1)])/4;
        }
    }
    return u;
}
```

Listing 6: MultiGrid function

```
vector <float> MultiGrid_Class::MultiGrid(vector <float> &f, int alpha_down, int alpha_up,
    int alpha_c, int num_lev, int N){
    int i,j;
    /*Set initialize solution guess as zero*/
    vector <float> u(f.size(), 0.0);
    /*Coarest level, approximate solution through classical iterative method*/
    if(num_lev == 1){
        Smooth_relax(u, f, alpha_c, N);
        return u;
    }
    /*Pre-smoothing, remove high frequency errors*/
    Smooth_relax(u, f, alpha_down, N);
    /*Apply operator A on u*/
    vector <float> Au = Operator_A(u, N);
    /*Calculate residual*/
    vector <float> r(f.size(),0.0);
```

```
16      FOR_EACH_CELL
17          r[IX(i,j)] = f[IX(i,j)] - Au[IX(i,j)];
18      END_FOR
19      /*Restriction (fine grid -> coarse grid)*/
20      vector <float> rc = Restriction(r, N);
21      /*Recursive, solve Ae = r*/
22      vector <float> ec = MultiGrid(rc, alpha_down, alpha_up, alpha_c, num_lev -1, N/2);
23      /*Interpolation (coarse grid -> fine grid)*/
24      vector <float> e = Interpolation(ec, N);
25      /*Fine grid solution correction*/
26      FOR_EACH_CELL
27          u[IX(i,j)] = u[IX(i,j)] + e[IX(i,j)];
28      END_FOR
29      /*Post-smoothing, remove high frequency errors*/
30      Smooth_relax(u, f, alpha_up, N);
31      return u;
32  }
```