Fast Iterative Solvers: Project 2

Jaeyong Jung (359804)

November 2, 2016

1. Plot the convergence using the measure $||r^m||_\infty/||r^0||_\infty$ against multigrid iterations m for meshes with n = 4, n = 7 (resulting in N = 16, and N = 128).

To begin with, $n = 4$ is tried to see the convergence. Simulation is done for two cases by varying ν as described in Figure 1. The maximum number of iteration is 30.

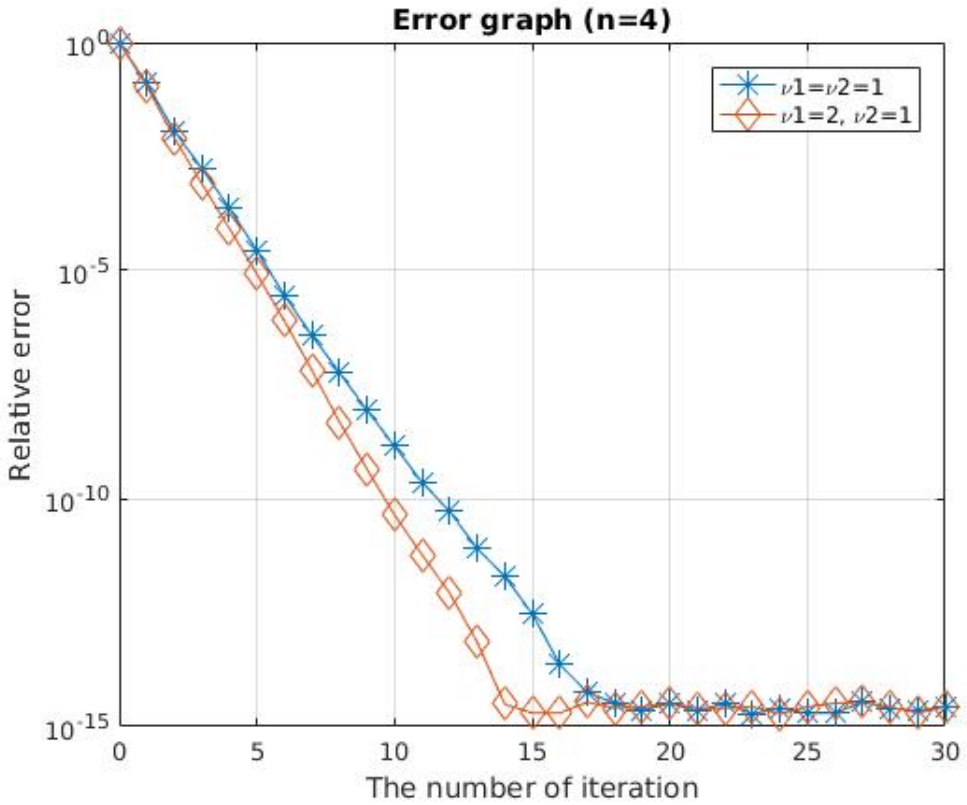

Figure 1: Error analysis when n is 4

For the first case, $\nu 1 = \nu 2 = 1$, The relative error decreases drastically until **18-th** iteration step. At that **18-th** point, the relative error is 3.239×10^{-15}. Then it reaches 2.699×10^{-15} at last. On the other hand, the later case, $\nu 1 = 2, \nu 2 = 1$, convergency is attained much faster. Untill **15-th** iteration, the relative error diminishes rapidly. At that point, relative error is 1.979×10^{-15}, which is even less than the first case's error at **18-th** point. Finally, the error at **30-th** step is 2.699×10^{-15} and it is identical to that of the first case. Two cases' final value is the same. Nevertheless, the second case shows oscilating behavior. At the **15-th** the error is 1.979×10^{-15} but the last error is bigger, 2.699×10^{-15}. It is guessed that coarsness of the mesh yield the oscillating bahavior. By comparing this result with numerical simulation with finer mesh, this guess will be verified.

Computational experiment with $n = 7$ is done to see if the rate of convergence is higher compared to coarser mesh. The result is shown on Figure 2.

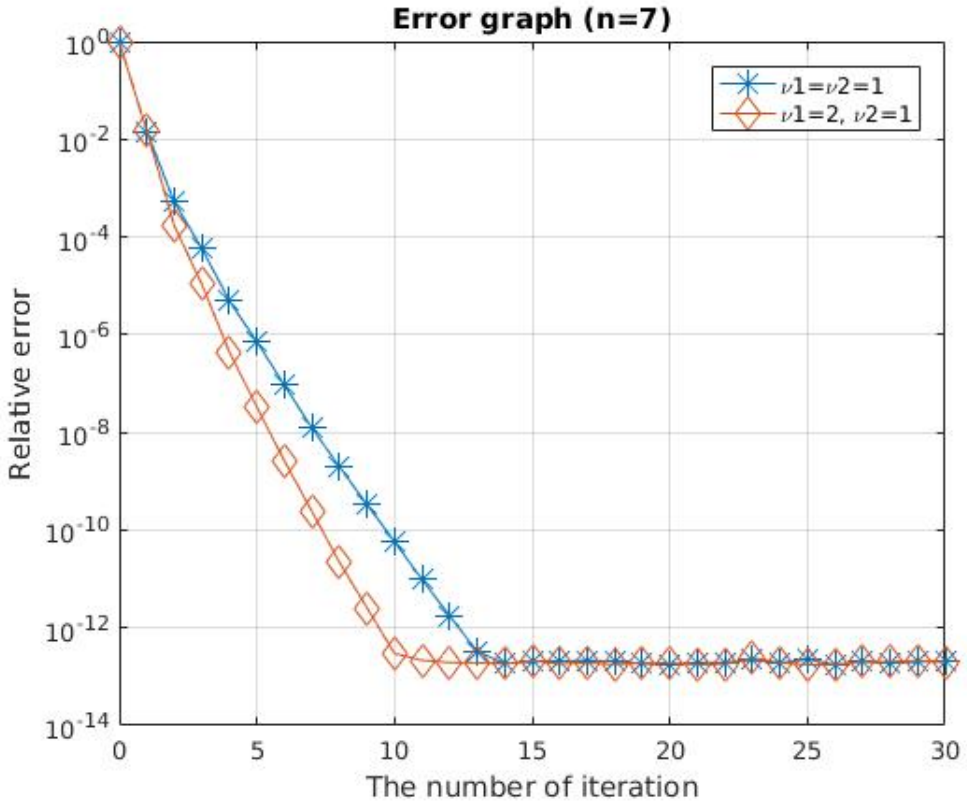

Figure 2: Error analysis when n is 7

When $\nu 1 = \nu 2 = 1$, the relative error decreases drastically until **13-th** iteration step. At that **13-th** point, the relative error is 3.288×10^{-13}. Then it reaches 2.109×10^{-13} at last, **30-th** step. On the other hand, the later case, $\nu 1 = 2, \nu 2 = 1$, convergency is attained much faster. Untill **10-th** iteration, the relative error diminishes rapidly. At that point, relative error is 2.976×10^{-13}, which is even less than the first case's error at **13-th** point. Finally, the error at **30-th** step is 1.951×10^{-13} and it is smaller as well.

In summary, it takes less iteration step for finer mesh to get convergence. Plus, finer mesh does hot have oscillating behavior. For both of them, more Gauss-Seidel pre-smoothing step makes numerical solution to converge with less iterations. However, the final iteration's value of error is smaller in coarser mesh. That is because the size of mesh is closer to the lowest level, where the equation is perfectly solved, for coarser mesh.

2. Further consideration about the effect of pre-Gauss-Seidel smoothing operation.)

The comparision of numerical experiments showed that pre-Gauss-Seidel smoothing will make the multigrid process to require less iteration for convergence. Nevertheless, it does not necessarily mean that more smoothing will take less time for convergence as well. Thus, it is necessary to conduct numerical experiment to compare runtimes. Before the comparision is done, other parameters are defined. $n = 7$ and $\nu 2 = 1$. $\nu 1$ is changed every time to see

difference in runtime.

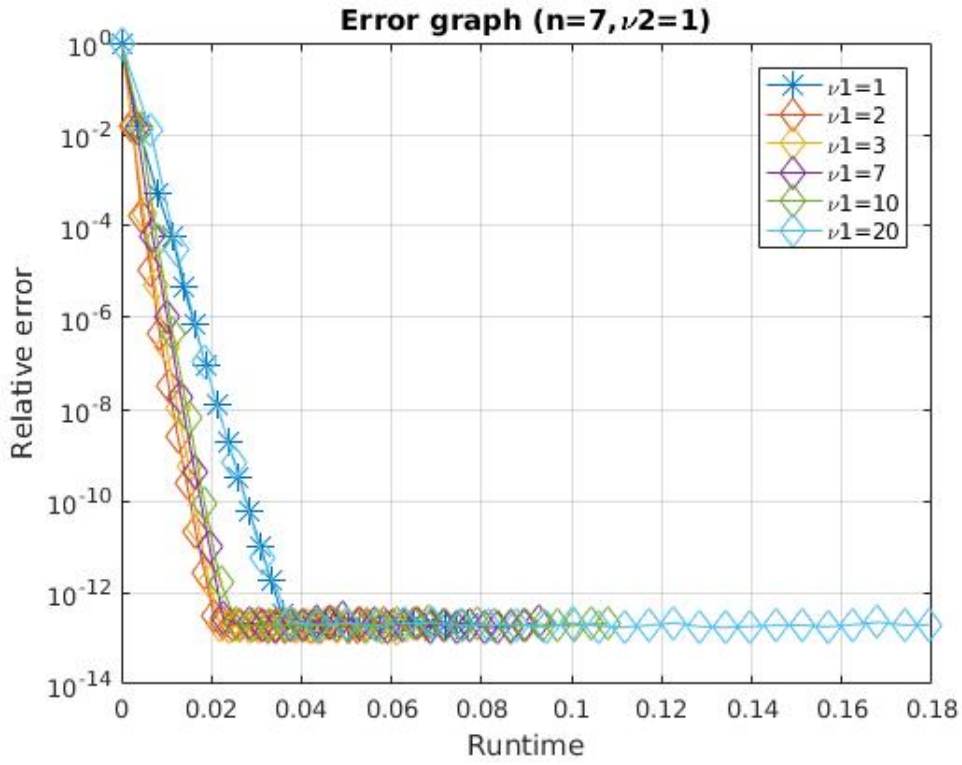

Figure 3: Runtime comparision

The convergence criteria is the point that relative error goes below 10^{-12}.

The fastest convergence in error is made when $\nu 1 = 2$ and it takes $20.63ms$. The slowest case is when $\nu 1 = 20$, and the runtime is $37.62ms$. The graph in Figure 3 proves that there is optimal number of smoothing. If there is too much smoothing, it will not only decrease the performance of computation but also takes more time than one smoothing. In conclusion, two or three times for smoothing is optimal.

3. Implementations of Gauss-Seidel Smoother, restriction operator, prolongation, and main function.

Important functions are copied on following page. The entire code is also submitted.

C code:

```
void GS(double* u0, double* f, int nu, int N)
{
    int iteration=0, i, j, k, l, m;
    double h = (1/(double)N);
    double tol = 1e-10;
    double norm = 1;
```

```
    double (*u_t) = (double(*)) calloc((N+1)*(N+1), sizeof(double)
       );

    for (m=0; m<nu; m++)
    {iteration++;
        // Load values onto u_t
        copy(u_t, u0, (N+1)*(N+1));

        for (j=1; j<=N-1;j++)
        {
            for (i=1; i<=N-1;i++)
            {
                u0[(N+1)*i + j] = (h*h*f[(N+1)*i + j] + u0[(N+1)*(
                   i-1) + j] + u0[(N+1)*i + j-1] + u0[(N+1)*(i+1)
                   + j] + u0[(N+1)*i + j+1])/4;
            }
        }
    }
    printf("GS smoothing done iteration = %d norm = %e \n",
       iteration, norm);

    free(u_t);
}



void Restriction(double* u_c, double* u, int N)
{
    int ii, jj, Nc = N/2;

    for (int i=1; i<=Nc-1;i++)
    {
        ii = 2*i;
        for (int j=1; j<=Nc-1;j++)
        {
            jj = 2*j;
            u_c[(Nc+1)*i + j] = (u[(N+1)*(ii-1) + (jj-1)] +2*u[(N
               +1)*(ii) + (jj-1)] + u[(N+1)*(ii+1) + (jj-1)] + 2*u
               [(N+1)*(ii-1) + (jj)] +4*u[(N+1)*(ii) + (jj)] + 2*u
               [(N+1)*(ii+1) + (jj)]
                    + u[(N+1)*(ii-1) + (jj+1)] + 2*u[(N+1)*(ii) +
                       (jj+1)] + u[(N+1)*(ii+1) + (jj+1)])/16;
        }
    }
}
```

```
void Prolongation(double* u, double* u_c, int N)
{
    int Nc = N/2;
    // initialize u
    for (int i=1; i<=N-1; i++)
    {
        for (int j=1; j<=N-1; j++)
        {
            u[(N+1)*i + j] = 0;
        }
    }

    for (int i=1; i<=Nc-1; i++)
    {
        int ii = 2*i;
        for (int j=1; j<=Nc-1; j++)
        {
            int jj = 2*j;
            double temp = (u_c[(Nc+1)*(i)+ (j)]);
            u[(N+1)*(ii-1)+ (jj-1)] = u[(N+1)*(ii-1)+ (jj-1)] + (
               double)1/4*temp;
            u[(N+1)*(ii-1)+ (jj+1)] = u[(N+1)*(ii-1)+ (jj+1)] + (
               double)1/4*temp;
            u[(N+1)*(ii+1)+ (jj+1)] = u[(N+1)*(ii+1)+ (jj+1)] + (
               double)1/4*temp;
            u[(N+1)*(ii+1)+ (jj-1)] = u[(N+1)*(ii+1)+ (jj-1)] + (
               double)1/4*temp;

            u[(N+1)*(ii)+ (jj-1)] = u[(N+1)*(ii)+ (jj-1)] + (
               double)1/2*temp;
            u[(N+1)*(ii)+ (jj+1)] = u[(N+1)*(ii)+ (jj+1)] + (
               double)1/2*temp;
            u[(N+1)*(ii-1)+ (jj)] = u[(N+1)*(ii-1)+ (jj)] + (
               double)1/2*temp;
            u[(N+1)*(ii+1)+ (jj)] = u[(N+1)*(ii+1)+ (jj)] + (
               double)1/2*temp;

            u[(N+1)*(ii)+ (jj)] += temp;

        }
    }
}
```

```
void MG(int l, double *u, double *f, int N, int gamma, int nu1,
   int nu2)
{// l: level, u: grid, f, N: N at l-th level, gamma: #iteration at
    each level
    int Nc = N/2;
    double *u_c = (double(*)) calloc((Nc+1)*(Nc+1), sizeof(double)
       );
    double (*r_l) = (double(*)) calloc((N+1)*(N+1), sizeof(double)
       );
    double (*r_l2) = (double(*)) calloc((Nc+1)*(Nc+1), sizeof(
       double));
    double (*e_l) = (double(*)) calloc((N+1)*(N+1), sizeof(double)
       );
    double (*e_l2) = (double(*)) calloc((Nc+1)*(Nc+1), sizeof(
       double));
    double (*u_t) = (double(*)) calloc((N+1)*(N+1), sizeof(double)
       );
    double h = (1/(double)N);
    double h2 = (double)h/2;

    printf("\nstart %d-th level, h = %e N = %d \n",l,h,N);

    GS(u, f, nu1, N);

    // Implement r_l = f- A_l*u_l (A_l: is Lagrangian operator)
    // u_t = A_l*u_t
    Laplacian(u_t, u, h, N);

    subtract(r_l, f, u_t, (N+1)*(N+1));
    // r_l2 = Restriction r_l
    Restriction(r_l2, r_l, N);

    if (l==1)
    {
        printf("reached at just before the lowest level Nc = %d\n"
           , Nc);
        GS(e_l2, r_l2, 1, Nc);
        //Inv_Laplacian( e_l2, r_l2, h2, Nc);
        // Jinxuan told me that Gauss relaxation is inverse
           opeation of laplacian
        // e_l2 = -e_l2
        Inv_sign( e_l2, (Nc+1)*(Nc+1));
```

```
        printf("The lowest error equation is solved!\n");
    }
    else
    {
        Inv_sign( r_l2 , (Nc+1)*(Nc+1));
        Init(e_l2 , (Nc+1)*(Nc+1), 0);
        for (int j=0; j< gamma; j++)
        {
            MG(l-1, e_l2 , r_l2 , Nc, gamma, nu1, nu2);
        }
    }
    // Continue writing from here
    Prolongation(e_l , e_l2 , N);

    subtract(u, u, e_l , (N+1)*(N+1));

    GS(u, f , nu2, N);

    free (u_c);
    free (r_l);
    free (r_l2);
    free (e_l);
    free (e_l2);
    free (u_t);
}



int main(void)
{
    int l=7;
    int N= pow(2,l);
    int Nc = (N/2);
    double h = (1/(double)N);
    double (*u) = (double(*)) calloc((N+1)*(N+1), sizeof(double));
    // coarse mesh
    double (*u_c) = (double(*)) calloc((Nc+1)*(Nc+1), sizeof(
       double));
    // real solution
    double (*u_s) = (double(*)) calloc((N+1)*(N+1), sizeof(double)
       );
    double (*f) = (double(*)) calloc((N+1)*(N+1), sizeof(double));

    printf("N = %d\n",N);
    printf("Nc = %d\n",Nc);
```

```
printf("h = %e\n",h);

// f and u_s initialization
Initialization(u_s, f, N);

int gamma = 2;
int nu1 = 7, nu2 = 1, iteration = 30;

double r0 = abs_max(f, (N+1)*(N+1));
double r[1 + iteration];
double r_time[1 + iteration];
double sta_t=clock();

r[0] = 1;

// Perform multigrid method for a certain number of times.
for (int m=0; m < iteration; m++)
{
    MG(l, u, f, N, gamma, nu1, nu2);
    r[m+1] = r_inf_norm(u, f, N)/r0;
    r_time[m+1] = (clock()- sta_t)/CLOCKS_PER_SEC;
}

char aa[20] = "error";
char bb[20] = "r_time";
char cc[20] = ".txt";
char nu1_c[10];
sprintf(nu1_c,"%d",nu1);

strcat(aa,nu1_c);
strcat(aa,cc);

strcat(bb,nu1_c);
strcat(bb,cc);

// save error and runtime as text file.
M_fprint(aa, r, iteration+1, 1);
M_fprint(bb, r_time, iteration+1, 1);

if ((r[iteration+1]) < 1e-8)
{
    printf("\nconverged! Relative error = %e \n\n", (r[
        iteration+1]/r0));
}
else
```

```
    {
        printf("\nFailed to converge, %e\n\n", (r[iteration+1]/r0)
            );
    }


    free (u);
    free (u_c);
    free (u_s);
    free (f);

    return 0;
}
```