Jingtian Zhang
3493913
Math114
Final Project

Index

Note: all outputs in the problem portion are not original C++ outputs. To browse the original output, you may check the reference.

Poisson Equation

We are given that

$\mathrm{u(x,y)}=\sin(\mathrm{x}*(1-2*\mathrm{y}))*\sin(\mathrm{y}*(1-\mathrm{x}))$, and $\Delta\mathrm{u}=u_{xx}+u_{yy}$

So we can compute that

$u_{xx} = 4*\cos(\mathrm{xy}-\mathrm{y})*\cos(2*\mathrm{xy}-\mathrm{x})*\mathrm{y}*\left(\mathrm{y}-\frac{1}{2}\right)+\sin(\mathrm{xy}-\mathrm{y})*\sin(2*\mathrm{xy}-\mathrm{x})\,(-5*y^2+4*\mathrm{y}-1)$

$u_{yy} = 4*\cos(\mathrm{xy}-\mathrm{y})*\cos(2*\mathrm{xy}-\mathrm{x})*\mathrm{x}*(\mathrm{x}-1)+\sin(\mathrm{xy}-\mathrm{y})*\sin(2*\mathrm{xy}-\mathrm{x})*(-5*x^2-+2*\mathrm{x}-1)$

→$\Delta\mathrm{u}=(4*x^2-4*\mathrm{x}+4*y^2-2*\mathrm{y})*\cos(\mathrm{xy}-\mathrm{y})*\cos(2*\mathrm{xy}-\mathrm{x})+(-5*x^2+2*\mathrm{x}-5*\mathrm{y}^2+4*\mathrm{y}-2)*\sin(\mathrm{xy}-\mathrm{y})*\sin(2*\mathrm{xy}-\mathrm{x})$

Since f = -Δu, we can conclude that
f= $-(4*x^2-4*\mathrm{x}+4*y^2-2*\mathrm{y})*\cos(\mathrm{xy}-\mathrm{y})*\cos(2*\mathrm{xy}-\mathrm{x})+(5*x^2-2*\mathrm{x}+5*\mathrm{y}^2-4*\mathrm{y}+2)*\sin(\mathrm{xy}-\mathrm{y})*\sin(2*\mathrm{xy}-\mathrm{x})$

Using following C++ code to implement Gaussian Elimination method, Jacobi Iteration method, Gauss-Seidel Iteration method and Conjugate Gradient method to solve this Poisson Equation problem, and we also need to find the most efficient way for computing such problem.

1. Gaussian Elimination(direct solver)

```
#include "stdafx.h"
#include <iostream>
#include <fstream>
#include <ctime>
#include <vector>
using namespace std;

double u(double x,double y)
{//equition of u as given
   return sin(x*(1-2*y))*sin(y*(1-x));
}

void printU(int nx,int ny)
{//find the real value
   double h=1.0/(nx+1);
   for(int i=0;i<ny+2;i++)//the most external rows
   {
          for(int j=0;j<nx+2;j++)//the most external columns
          {
                 cout<<u(j*h,i*h)<<" ";
          }
          cout<<endl;
   }
}

double f(double x,double y)
{//function of f(x,y)
   return        (4*x*(1-x)+2*y*(1-2*y))*cos(x*(1-2*y))*cos(y*(1-x))+(pow(1-x,2)+4*pow(x,2)+pow(1-2*y,2)+pow(y,2))*sin(x*(1-
2*y))*sin(y*(1-x));
}

double *CreateArr(int n)
{//create array
   return new double[n];
}

double **CreateMatrix(int n,int m)
{//create matrix
   double**mat = nullptr;
   mat = new double*[n];
```

```
    for (int i = 0; i < n; i++)
    {
            mat[i] = new double[m];
    }
    return mat;
}

double max(double *x, int n)
{//find infinity norm for a pointer
    double max = fabs(x[0]);
    for (int i = 0; i < n; i++)
    {
            if (fabs(x[i]) > max)
            {
                    max =fabs( x[i]);
            }
    }
    return max;
}

double max1(vector<double> x, int n)
{//find infinity norm for a vector
    double max = fabs(x[0]);
    for (int i = 0; i < n; i++)
    {
            x[i]=fabs(x[i]);
            if (x[i] > max)
            {
                    max = x[i];
            }
    }
    return max;
}

void GaussianElimination(double **a,double *b,int n)
{//Gaussian Elimination
    double **m=nullptr;
    m=new double *[n];
    m[0]=new double[n*n];
    for(int i=0;i<n*n;i++)
    {
            m[0][i]=0;//initial value
    }
    for(int i=1;i<n;i++)
```

```
  {
        m[i]=m[i-1]+n;
  }
  for(int i=0;i<n-1;i++)
  {
        for(int j=i+1;j<n;j++)
        {
              if(a[i][i]==0)
              {//when A[i][i]=0, switch to another row
                    double temp=0;
                    for(int k=0;k<n;k++)
                    {
                          temp=a[i][k];
                          a[i][k]=a[n-1][k];
                          a[n-1][k]=temp;
                    }
              }
              m[j][i]=a[j][i]/a[i][i];
              for(int k=0;k<n;k++)
              {
                    a[j][k]=a[j][k]-m[j][i]*a[i][k];
              }
              b[j]=b[j]-m[j][i]*b[i];
        }
  }
  delete[] m[0];
  delete[] m;
}

void BackwardSubstitution(double **a,double *b,double *x,int n)
{//backward substitution
  x[n-1]=b[n-1]/a[n-1][n-1];
  for(int i=n-2;i>=0;i--)
  {
        double sum=0;
        for(int j=i+1;j<n;j++)
        {
              sum+=a[i][j]*x[j];
        }
        x[i]=(b[i]-sum)/a[i][i];
  }
}

void Gaussian()
```

```
{//gaussian elimination method
  int nx[6] = {9,19,29,39,49,59};
  int ny[6] = {4,9,14,19,24,29};
  ofstream outfile("C:\\Users\\Jingtian Zhang\\Desktop\\gauss.timing");
  cout<<"h\ttime\texponent"<<endl;
  for(int k=0;k<6;k++)
  {
          int n=nx[k]*ny[k];//size of matrix
          double h=1.0/(nx[k]+1);//h=0.1,0.05.....
          //create matrix A and vector b st b(Av=b)
          double **a = CreateMatrix(n,n);
          for (int i = 0; i <n; i++)
          {
                  for (int j = 0; j < n; j++)
                  {
                          if(i==j)
                          {
                                  a[i][j] =4;//diagonal
                          }
                          else if(fabs(double(i-j))==1 && (i+j)%nx[k]!=(nx[k]-1))
                          {
                                  a[i][j] = -1;
                          }
                          else if(fabs(double(i-j))==nx[k])
                          {
                                  a[i][j]=-1;
                          }
                          else
                          {
                                  a[i][j]=0;
                          }
                  }
          }
          double *v=CreateArr(n);
          double *b=CreateArr(n);
          for(int i=0;i<ny[k];i++)//row
          {
                  for(int j=0;j<nx[k];j++)//column
                  {
                          v[i*nx[k]+j]=0;
                          b[i*nx[k]+j]=(h*h*f((j+1)*h,(i+1)*h));//h*h*f(x,y)
                  }
          }
          clock_t t1,t2;//set clock
```

```
            t1=clock();
            GaussianElimination(a, b, n);
            BackwardSubstitution(a, b, v, n);
            t2=clock();
            clock_t time=t2-t1;//count time cost
            double r=log(time)/log(n);
            cout<<h<<"\t"<<time<<"\t"<<r<<endl;
            outfile<<h<<" "<<time<<" "<<r<<endl;
            char fileName[80];
            sprintf_s(fileName, "%s%d%s", "C:\\Users\\Jingtian Zhang\\Desktop\\gauss_h", k+1,".txt");
            ofstream gauss_h(fileName);// generate file gauss_hx.txt for plotting
            sprintf_s(fileName, "%s%d%s", "C:\\Users\\Jingtian Zhang\\Desktop\\true_h", k+1,".txt");
            ofstream true_h(fileName);//generate file gauss_hx.txt for plotting
            for(int i=1;i<=ny[k];i++)
            {
                  for(int j=1;j<=nx[k];j++)
                  {
                        gauss_h<<j*h<<" "<<i*h<<" "<<v[(i-1)*nx[k]+j-1]<<endl;
                        true_h<<j*h<<" "<<i*h<<" "<<u(j*h,i*h)<<endl;
                  }
            }
             gauss_h.close();
            true_h.close();
            for (int i = 0; i < n; i++)
            {
                  delete[] a[i];
            }
            delete[] a;//recycle
            delete[] v;
            delete[] b;
      }
      outfile.close();
}
```

After running the main function, we get following data for the function above (note: the table is not original C++ output)

n_x (number of columns)	n_y (number of rows)	h (step size)	Time	Exponent
9	4	0.1	1	$-\infty$
19	9	0.05	42	0.726938
29	14	0.03333	573	1.05736
39	19	0.025	3327	1.22727
49	24	0.02	13072	1.34065
59	29	0.01667	39745	1.4225

2. Jacobi Iteration

General form of Jacobi Iteration is

$$u_{i,j}^{(k+1)} = \left(f_{i,j} + \frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}}{(\Delta y)^2} \right) \Big/ \left(\frac{2}{(\Delta x)^2} + \frac{2}{(\Delta y)^2} \right)$$

Here, Δx = Δy = h, so it can be transferred as

$$(f_{i,j} + \frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)}}{h^2} + \frac{u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)}}{h^2})/(\frac{4}{h^2})$$

Then we multiply both numerator and denominator by h^2 and get

$$(h^2 * f_{i,j} + u_{i+1,j}^{(k)} + u_{i-1,j}^{(k)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k)})/4$$

```
void Jacobi()
{//Jacobi method
  int nx[6] = {9,19,29,39,49,59};
  int ny[6] = {4,9,14,19,24,29};
  ofstream outfile("C:\\Users\\Jingtian Zhang\\Desktop\\Jacobi.timing");
  cout<<"h\ttime\texponent"<<endl;
  for(int k=0;k<6;k++)
  {
        int n=(nx[k]+2)*(ny[k]+2);//number of elements
        double h=1.0/(nx[k]+1);
        double tol=1e-10;
        double *initialU=CreateArr(n);
        double *generateU=CreateArr(n);
        double *arrayF=CreateArr(n);
        double *error=CreateArr(n);
        for(int i=0;i<ny[k]+2;i++)//the most external rows
        {
              for(int j=0;j<nx[k]+2;j++)//the most external columns
              {
                    initialU[i*(nx[k]+2)+j]=0;
                    generateU[i*(nx[k]+2)+j]=0;
                    arrayF[i*(nx[k]+2)+j]=f(j*h,i*h);
                    error[i*(nx[k]+2)+j]=0;//error
              }
        }
        int count=0;
        clock_t t1,t2;//set clock
        t1=clock();
        while(true)
        {
                for(int i=1;i<ny[k]+1;i++)//inner rows
                {
                    for(int j=1;j<nx[k]+1;j++)//inner columns
                    {
                        initialU[i*(nx[k]+2)+j]=generateU[i*(nx[k]+2)+j];

        generateU[i*(nx[k]+2)+j]=(pow(h,2)*arrayF[i*(nx[k]+2)+j]+initialU[i*(nx[k]+2)+j+1]+initialU[i*(nx[k]+2)+j-
    1]+initialU[(i+1)*(nx[k]+2)+j]+initialU[(i-1)*(nx[k]+2)+j])/4;
                    }
                }
                count++;//count iteration times
                for(int size=0;size<n;size++)
                {
                    error[size]=generateU[size]-initialU[size];
                }
                if(max(error,n)<tol)
                {//when error is too small, we stop the whole process
                    break;
                }
            }
            t2=clock();
            clock_t time=t2-t1;//count time cost
            double r=log(time)/log(n);
            cout<<h<<"\t"<<time<<"\t"<<r<<endl;
            outfile<<h<<" "<<time<<" "<<r<<endl;
            char fileName[80];
            sprintf_s(fileName, "%s%d%s", "C:\\Users\\Jingtian Zhang\\Desktop\\Jacobi_h", k+1,".txt");
            ofstream Jacobi_h(fileName);//generate file gauss_hx.txt for plotting
            for(int i=1;i<=ny[k];i++)
            {
                for(int j=1;j<=nx[k];j++)
                {
                    Jacobi_h<<j*h<<" "<<i*h<<" "<<generateU[i*(nx[k]+2)+j]<<endl;
                }
            }
             Jacobi_h.close();
            delete[] initialU;
            delete[] generateU;
            delete[] arrayF;
            delete[] error;
        }
        outfile.close();
      }
```

After running main function, we get following data for the function above (note: the table is not original C++ output)

n_x (number of columns)	n_y (number of rows)	h (step size)	Time	Exponent
9	4	0.1	2	0.165443
19	9	0.05	38	0.668377
29	14	0.03333	150	0.807311
39	19	0.025	466	0.909159
49	24	0.02	1124	0.977013
59	29	0.01667	2205	1.02036

---

3. Gauss-Seidel Iteration

The general form of Gauss-Seidel Iteration is

$$u_{i,j}^{(k+1)} = \left(f_{i,j} + \frac{u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)}}{(\Delta x)^2} + \frac{u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)}}{(\Delta y)^2}\right) / \left(\frac{2}{(\Delta x)^2} + \frac{2}{(\Delta y)^2}\right)$$

Like what we did in Jacobi Iteration, we substitute Δx and Δy with h and get

$$(h^2 * f_{i,j} + u_{i+1,j}^{(k)} + u_{i-1,j}^{(k+1)} + u_{i,j+1}^{(k)} + u_{i,j-1}^{(k+1)})/4$$

The only difference is that we use we use the (k+1)th iteration of $u_{i-1,j}$ and $u_{i,j-1}$ here.

```
void Gauss_Seidel()
{//Gauss-Seidel method
  int nx1[6] = {9,19,29,39,49,59};
  int ny1[6] = {4,9,14,19,24,29};
  ofstream outfile("C:\\Users\\Jingtian Zhang\\Desktop\\Seidel.timing");
  cout<<"h\ttime\texponent"<<endl;
  for(int k=0;k<6;k++)
  {
        int nx=nx1[k];
        int ny=ny1[k];
        int n=(nx+2)*(ny+2);//number of elements
        double h=1.0/(nx+1);
        double tol=1e-10;
        double *initialU=CreateArr(n);
        double *generateU=CreateArr(n);
        double *arrayF=CreateArr(n);
        double *error=CreateArr(n);
        for(int i=0;i<ny+2;i++)//the most external rows
        {
              for(int j=0;j<nx+2;j++)//the most external columns
              {
                    initialU[i*(nx+2)+j]=0;
                    generateU[i*(nx+2)+j]=0;
```

```
                arrayF[i*(nx+2)+j]=f(j*h,i*h);
                error[i*(nx+2)+j]=0;//error
            }
        }
        int count=0;
        clock_t t1,t2;
        t1=clock();
        while(true)
        {//temporarily store iteration result of last time
            for(int i=1;i<ny+1;i++)//inner rows
            {
                for(int j=1;j<nx+1;j++)//inner columns
                {
                    initialU[i*(nx+2)+j]=generateU[i*(nx+2)+j];
                }
            }
            for(int i=1;i<ny+1;i++)//inner rows
            {
                for(int j=1;j<nx+1;j++)//inner columns
                {
                    generateU[i*(nx+2)+j]=(pow(h,2)*arrayF[i*(nx+2)+j]+initialU[i*(nx+2)+j+1]+generateU[i*(nx+2)+j-
1]+initialU[(i+1)*(nx+2)+j]+generateU[(i-1)*(nx+2)+j])/4;
                }
            }
            count++;//count iteration times
            for(int size=0;size<n;size++)
            {
                error[size]=generateU[size]-initialU[size];
            }
            if(max(error,n)<tol)
            {
                break;
            }
        }
        t2=clock();
        clock_t time=t2-t1;
        double r=log(time)/log(n);
        cout<<h<<"\t"<<time<<"\t"<<r<<endl;
        outfile<<h<<" "<<time<<" "<<r<<endl;
        char fileName[80];
        sprintf_s(fileName, "%s%d%s", "C:\\Users\\Jingtian Zhang\\Desktop\\Seidel_h", k+1,".txt");
        ofstream Seidel_h(fileName);//generate file gauss_hx.txt for plotting
        for(int i=1;i<=ny;i++)
        {
                for(int j=1;j<=nx;j++)
                {
                        Seidel_h<<j*h<<" "<<i*h<<" "<<generateU[i*(nx+2)+j]<<endl;
                }
        }
         Seidel_h.close();

        delete[] initialU;
        delete[] generateU;
        delete[] arrayF;
        delete[] error;
    }
    outfile.close();
}
```

After running main function, we get following data for the function above (note: the table is not original C++ output)

n_x (number of columns)	n_y (number of rows)	h (step size)	Time	Exponent
9	4	0.1	1	$-\infty$
19	9	0.05	11	0.440594
29	14	0.03333	55	0.645659
39	19	0.025	163	0.753726
49	24	0.02	391	0.830149
59	29	0.01667	795	0.885151

4. Conjugate Gradient Method

This method is used to solve linear equations Ax=b, where A is a matrix and a, b are two vectors. Here is the algorithm

Algorithm 1 The Conjugate Gradient Method

```
 1: Given $\mathbf{x}_0$;
 2: $\mathbf{r}_0 = \mathbf{A}\mathbf{x}_0 - \mathbf{b}$, $\mathbf{p}_0 = -\mathbf{r}_0$, $k = 0$.
 3: while $\mathbf{r}_k >$ tol do
 4:     $\alpha_k = \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{A}\mathbf{p}_k}$;
 5:     $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{p}_k$;
 6:     $\mathbf{r}_{k+1} = \mathbf{r}_k + \alpha_k \mathbf{A}\mathbf{p}_k$;
 7:     $\beta_{k+1} = \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$;
 8:     $\mathbf{p}_{k+1} = -\mathbf{r}_{k+1} + \beta_{k+1}\mathbf{p}_k$;
 9:     $k = k + 1$;
10: end while
```

To avoid saving matrix A in the Conjugate Gradient part, we use existing template vector in C++ library to define the multiplication of matrix and vector.

```
vector<double> MatrixMultipleVector(vector<double> &v,int nx)
{
    int n=v.size();
```

```
   vector<double> tempVector(n);
   vector<double> result(n);
   for(int i=0;i<n;i++)
   {
            for(int j=0;j<n;j++)
            {
                     if(i==j)
                     {
                              tempVector[j] =4;//diagonal
                     }
                     else if(fabs(double(i-j))==1 && (i+j)%nx!=(nx-1))
                     {
                              tempVector[j] = -1;
                     }
                     else if(fabs(double(i-j))==nx)
                     {
                              tempVector[j]=-1;
                     }
                     else
                     {
                              tempVector[j]=0;
                     }
                     result[i]+=tempVector[j]*v[j];
            }//calculate product of row vector and column vector
   }
   return result;
}

void CG1()
{//Conjugate Gradient method
   int nx1[6] = {9,19,29,39,49,59};
   int ny1[6] = {4,9,14,19,24,29};
   ofstream outfile("C:\\Users\\Jingtian Zhang\\Desktop\\CG_MV.timing");
   cout<<"h\ttime\texponent"<<endl;
   for(int k=0;k<6;k++)
   {
            int nx=nx1[k];
            int ny=ny1[k];
            int n=nx*ny;
            double h=1.0/(nx+1);
            double tol=1e-6;
            vector<double> initialX(n);//define some vectors for later computation
            vector<double> initialError(n);
            vector<double> initialP(n);
```

```
vector<double> generateX(n);

vector<double> generateError(n);

vector<double> generateP(n);

vector<double> arrayF(n);

double alpha=0;

double belta=0;

for(int i=0;i<ny;i++)

{//initialization

        for(int j=0;j<nx;j++)

        {

                initialX[i*nx+j]=0;//initial value

                arrayF[i*nx+j]=h*h*f((j+1)*h,(i+1)*h);

        }

}

vector<double> Ax(n);

Ax=MatrixMultipleVector(initialX,nx);

for(int i=0;i<n;i++)

{

        initialError[i]=Ax[i]-arrayF[i];//initial error r0=Ax0-b

        initialP[i]=-initialError[i];//p0=-r0

}

int count=0;

double norm=max1(initialError,n);

clock_t t1,t2;

t1=clock();

while(norm>tol)

{

        double numerator=0,denominator=0;

        vector<double> currentResult(n);

        currentResult=MatrixMultipleVector(initialP,nx);

        for(int i=0;i<n;i++)

        {

                numerator+=initialError[i]*initialError[i];

                denominator+=initialP[i]*currentResult[i];

        }

        alpha=numerator/denominator;//alpha

        for(int i=0;i<n;i++)

        {

                generateX[i]=initialX[i]+alpha*initialP[i];

                initialX[i]=generateX[i];

                generateError[i]=initialError[i]+alpha*currentResult[i];


                initialError[i]=generateError[i];

        }
```

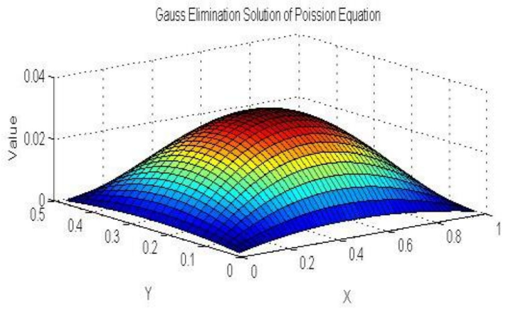
```
                        denominator=numerator;//r[k]*r[k]
                        numerator=0;
                        for(int i=0;i<n;i++)
                        {
                                numerator +=generateError[i]*generateError[i];//r[k+1]*r[k+1]
                        }
                        belta=numerator /denominator ; //beta
                        for(int i=0;i<n;i++)
                        {
                                generateP[i]=-generateError[i]+belta*initialP[i];
                                initialP[i]=generateP[i];
                        }
                        count++;
                        norm=max1(generateError,n);
                }
                t2=clock();
                clock_t time=t2-t1;
                double r=log(time)/log(n);
                cout<<h<<"\t"<<time<<"\t"<<r<<endl;
                outfile<<h<<" "<<time<<" "<<r<<endl;
                char fileName[80];
                sprintf_s(fileName, "%s%d%s", "C:\\Users\\Jingtian Zhang\\Desktop\\CG_MV", k+1,".txt");
                ofstream CG(fileName);//generate file gauss_hx.txt for plotting
                for(int i=1;i<=ny;i++)
                {
                        for(int j=1;j<=nx;j++)
                        {
                                CG<<j*h<<" "<<i*h<<" "<<generateX[(i-1)*nx+j-1]<<endl;
                        }
                }
                 CG.close();
   }
}

int main()
{
   cout<<"Result of Gaussian Elimination： "<<endl;
   Gaussian();
   cout<<"Result of Jacobi： "<<endl;
   Jacobi();
   cout<<"Result of Gauss-Seidel： "<<endl;
   Gauss_Seidel();
   cout<<"Result of Conjugate Gradient Method (Matrix A was not stored!)： "<<endl;
   CG1();
```

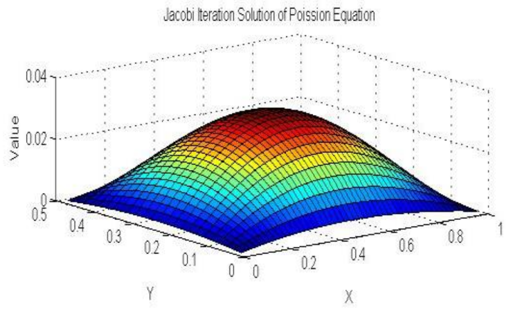
```
        return 0;
    }
```

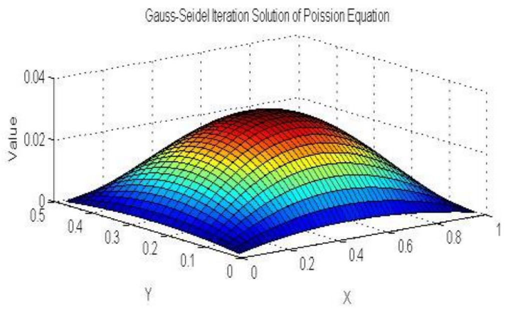
After running main function, we get following data for the function above (note: the table is not original C++ output)

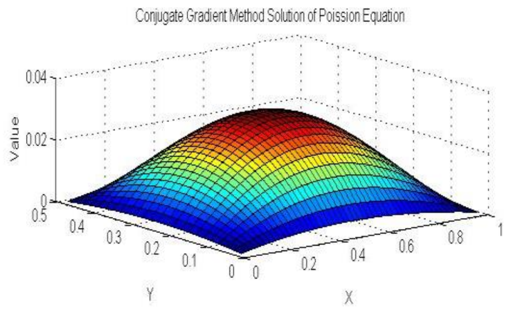
n_x (number of columns)	n_y (number of rows)	h (step size)	Time	Exponent
9	4	0.1	87	1.24624
19	9	0.05	3187	1.56892
29	14	0.03333	25367	1.68841
3	19	0.025	106985	1.75249
49	24	0.02	304687	1.78603
59	29	0.01667	759074	1.81869

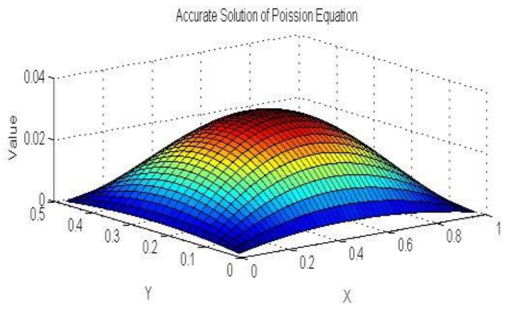
After all, we plot 3D graphs of relationships among u(x, y), x and y through files *true_h5.txt*, *gauss_h5.txt*, *Jacobi_h5.txt*, *Seidel_h5.txt* and *CG_MV5.txt* (because all h5 files contain 49*24 points, which is enough for a clear graph) and Matlab functions surf and plot3d, then we have follows graphs

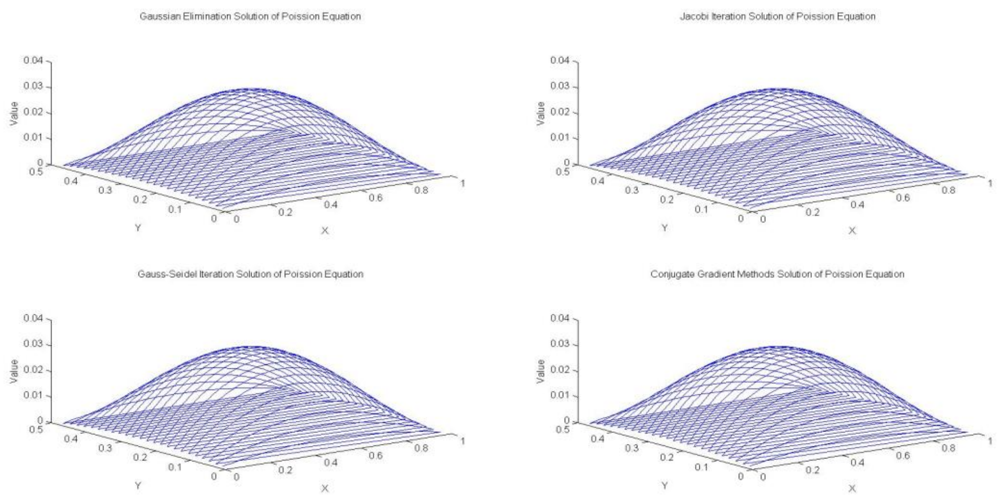






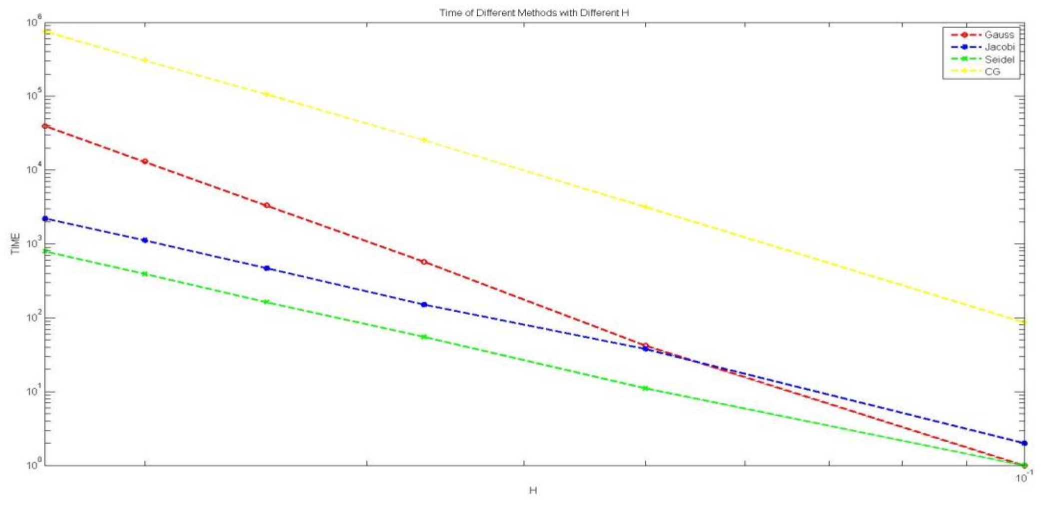






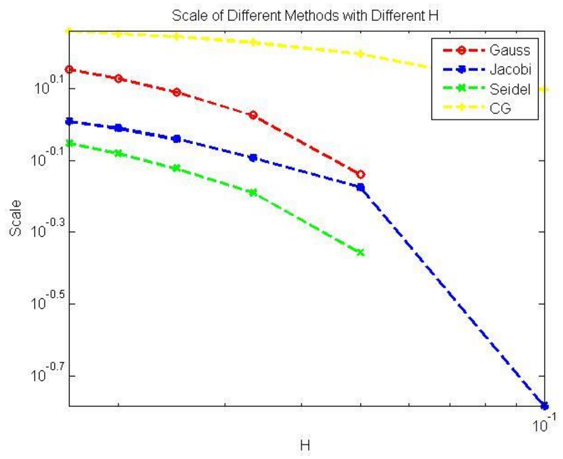
Based on these graphs, we can say that all these methods can accurately approximate the Poisson Equation, since all these four 3D graphs look pretty similar to that of the accurate Poisson Equation.

We then plot the relationship between time and h (step size) for all these four methods based on *gauss.timing*, *Jacobi.timig*, *Seidel.timing* and *CG_MV.timing*.



Through this graph, we can conclude that Gauss-Seidel is the most efficient way for approximating Poisson Equation, because its time cost is always pretty low. Whereas, Conjugate Gradient Method is the least efficient way in all these four methods, since its time cost line is always above that of other methods. From tables generated before we can see that when n_x=59 and n_y=29, the time cost of Gauss-Seidel Iteration is 795 which is much faster than the time cost of Conjugate Gradient Method (3187) when n_x=19 and n_y=9. In addition, at the beginning, Jacobi Iteration is faster than Gaussian Elimination, but the time cost growth rate of Jacobi Iteration is also bigger than that of Gaussian Elimination. So as h increasing, the efficiency of Jacobi Iteration will finally be surpassed by that of Gaussian Elimination.

Moreover, through these four timing files, we can also plot the relationship between h and exponent.



We can see that as h → 0, all exponents of these four methods get closer to 2, and as h increases, the exponents become smaller. From tables above, we can see that when h=0.01667 the exponent of Conjugate Gradient Method is 1.81869 which is the closest to 2 among these four methods. However, the exponent of Gauss-Seidel Iteration is only 0.885151 which is the furthest from 2 among these four methods. Hence, from the graph, we can conclude that the more time costing a method is, the closer its exponent is away from 2 as h→ 0.

Overall, all these four methods are viable for approximating Poisson Equation, but to increase our efficiency, we had better use Gauss-Seidel Iteration to do this job.

Adaptive Quadrature

Algorithm of adaptive quadrature is

$$Q(f;a,b) = \frac{b-a}{2}\sum_{i=1}^{3} w_i f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right)$$

$$E = \left|Q(f;a_0,b_0) - \left(Q\left(f;a_0,\frac{a_0+b_0}{2}\right) + Q\left(f;\frac{a_0+b_0}{2},b_0\right)\right)\right|.$$

If $E <$ tol, we accept

$$Q\left(f;a_0,\frac{a_0+b_0}{2}\right) + Q\left(f;\frac{a_0+b_0}{2},b_0\right)$$

as an accurate approximation to the integral in the interval $[a_0,b_0]$. If $E >$ tol, we repeat the procedure with the intervals $[a_0,(a_0+b_0)/2]$ and $[(a_0+b_0)/2,b_0]$.

We use the following code to implement adaptive quadrature

```
#include "stdafx.h"

#include <iostream>

#include <stack>

using namespace std;
```

```
double tol=1e-10;
stack<double> st;
double f1(double x)
{
	return 1.0/pow(x,0.5);
}
double f2(double x)
{
	return log(fabs(x-1));
}

double Q(double (*f)(double),double a,double b)
{//calculate integral
	double w[3]={5.0/9,8.0/9,5.0/9};
	double x[3]={-1.0*sqrt(3.0/5),0,sqrt(3.0/5)};
	if(a>b)
	{
		exit(0);
	}
	double result=0;
	double mid=(b-a)/2;
	double mean=(a+b)/2;
	for(int i=0;i<3;i++)
	{
		result+=w[i]*f(mid*x[i]+mean);
	}
	result*=mid;
	return result;
}

double iteration(double (*f)(double),double a,double b)
{//iteration
	double mean=(a+b)/2;
	double head=Q(f,a,mean);//Q(f;a,(a+b)/2)
	double rear=Q(f,mean,b);//Q(f;(a+b)/2,b)
	double error=fabs(Q(f,a,b)-(head+rear));//error
	double result=0;
	if(error>tol)
	{
		st.push(mean);//maximum of [a,mean(a,b)]
		st.push(a);//minimum of [a,mean(a,b)]
		st.push(b);//maximum of [mean(a,b),b]
		st.push(mean);//minimum of [mean(a,b),b]
		while(st.empty()==false)
```

```
            {
                a=st.top();
                st.pop();//remove from the stack
                b=st.top();
                st.pop();//remove from the stack
                mean=(a+b)/2;
                head=Q(f,a,mean);
              rear=Q(f,mean,b);
              error=fabs(Q(f,a,b)-(head+rear));
                if(error>tol)
                {//when error is too big, we repeat procedure above while loop
                    st.push(mean);
                    st.push(a);
                    st.push(b);
                    st.push(mean);
                }
                else
                {
                    result+=(head+rear);
                }
            }
        }
        else
        {
            result+=(head+rear);
        }
        return result;
}
int main()
{
        double q1=iteration(f1,0,1);
        cout<<"Integral of 1/sqrt(x) from 0 to 1 is "<<q1<<endl;
        double q2=iteration(f2,0,1)+iteration(f2,1,2); //when x in (0,1), we have log(1-x). when x in
(1,2), we have log(x-1)
        cout<<"Integral of log|x-1| from 0 to 2 is "<<q2<<endl;
        return 0;
}
```

After running the whole program, we get $\int_0^1 1/\sqrt{x}\,\mathrm{dx} = 2$

and $\int_0^2 \log|x-1|\,dx = -2$.

To check the accuracy of adaptive quadrature approximation, we need to compute these two integrations manually

1. $\int_0^1 1/\sqrt{x}\,\mathrm{dx} = \int_0^1 x^{-1/2}\,dx = 2 * x^{\frac{1}{2}}|_0^1 = 2 - 0 = 2$

2. $\int_0^2 \log|x-1|\,dx = \int_1^2 \log(x-1)\,dx + \int_0^1 \log(1-x)\,dx = [(x-1) * \log(x-1) - x]|_1^2 + [(x-1) * \log(1-x) - x]|_0^1 = 0 - 2 - 0 + 1 - 0 - 1 - 0 + 0 = -1 - 1 = -2$

Thus, both integrals were computed correctly via adaptive quadrature. Therefore, we can conclude that adaptive quadrature is a precise way to approximate integrals.

Reference

Q1 output:

```
Result of Gaussian Elimination：
h        time     exponent
0.1      1        -inf
0.05     42       0.726938
0.0333333         573      1.05736
0.025    3327     1.22727
0.02     13072    1.34065
0.0166667         39745    1.4225
Result of Jacobi：
h        time     exponent
0.1      2        0.165443
0.05     38       0.668377
0.0333333         150      0.807311
0.025    466      0.909159
0.02     1124     0.977013
0.0166667         2205     1.02036
Result of Gauss-Seidel：
h        time     exponent
0.1      1        -inf
0.05     11       0.440594
0.0333333         55       0.645659
0.025    163      0.753726
0.02     391      0.830149
0.0166667         795      0.885151
Result of Conjugate Gradient Method (Matrix A was not stored!)：
h        time     exponent
0.1      87       1.24624
0.05     3187     1.56892
0.0333333         25367    1.68841
0.025    106985   1.75249
0.02     304687   1.78603
0.0166667         759074   1.81869
Press any key to continue . . .
```

(Files gauss.timing, Jacobi.timing, Seidel.timing and CG_MV.timing contains the same data as the output data above)

Q2 output:

```
Integral of 1/sqrt(x) from 0 to 1 is 2
Integral of log|x-1| from 0 to 2 is -2
Press any key to continue . . .
```