FE formulation for 2D-Poisson equation

C++ code for CG with Linear basis

June 14, 2024

H.kishnani

1 Problem Definition

The 2D Diffusion problem to be solved as given in assignment is:

$$-\alpha\left(\frac{\partial^2\phi}{\partial x^2}+\frac{\partial^2\phi}{\partial y^2}\right)=f \qquad in \quad \Omega \tag{1}$$

with the Boundary Condition

$$\phi = g \qquad on \quad \partial\Omega \tag{2}$$

here f and g are given functions as:

$$g(x,y)=\begin{cases}0 & \text{if x=0}\\ y & \text{if x=1}\\ (x-1)sin(x) & \text{if y=0}\\ x(2-x) & \text{if y=1}\end{cases} \tag{3}$$

Taking the initial guess as $\phi^{(0)}=0$ and $\alpha=1$. With mesh size $h=1/10$, $h=1/20$ and $h=1/40$.

2 Mesh Setup

Mesh is generated using Gmsh C++ API. This can be obtained by installing it from source.
Using Gmsh is easier with OCC (OpenCasCade) but it has an extra dependency thus has been avoided here. Without going into the specific commands, we can create a square using geo entities like points, lines and curve loop. Then Plane surface can be generated and each of these entities can be given some unique tags. Two entities of same dimensions, cannot have same tags. Physical group can also be created using entities and they can be further utilized in imposing boundary condition.

```cpp
// mesh_domain.cpp
#include <set>
#include <stdlib.h>
#include <gmsh.h>
#include <iostream>
#include <vector>
#include <string>

using namespace std;
using namespace gmsh;

int main(int argc, char* argv[])
{

    // initializing cpp API for GMSH
    initialize();
    double lc = (double)atof(argv[1]);

    model::add("square");

    // Square domain points
    int P1 = model::geo::addPoint(0, 0, 0, lc, 1),
        P2 = model::geo::addPoint(1, 0, 0, lc, 2),
        P3 = model::geo::addPoint(1, 1, 0, lc, 3),
        P4 = model::geo::addPoint(0, 1, 0, lc, 4);

```

```
    int L1 = model::geo::addLine(P1, P2, 1),
        L2 = model::geo::addLine(P2, P3, 2),
        L3 = model::geo::addLine(P3, P4, 3),
        L4 = model::geo::addLine(P4, P1, 4);

    int Csq = model::geo::addCurveLoop({L1, L2, L3, L4}, 1);

    model::geo::addPlaneSurface({Csq}, 1);

    // Do less number of times
    model::geo::synchronize();

    // For Boundary Conditions
    model::addPhysicalGroup(1, {L1}, 1, "bottom");
    model::addPhysicalGroup(1, {L2}, 2, "right");
    model::addPhysicalGroup(1, {L3}, 3, "top");
    model::addPhysicalGroup(1, {L4}, 4, "left");

    model::addPhysicalGroup(2, {Csq}, -1, "surface");

    model::mesh::generate(2);

    option::setNumber("Mesh.SaveAll", 1);
    write("square.msh");

    gmsh::finalize();

    return 0;
}
/*
P4(0,1)     (L3)    P3(1,1)
    *-----------------*
    |                 |
    |                 |
    |                 |
(L4)|                 |(L2)
    |                 |
    |                 |
    |                 |
    *-----------------*
P1(0,0)     (L1)    P2(1,0)


g++ -o sq_mesh -Iinclude mesh_domain.cpp -Llib -lgmsh
*/

```

3 Problem Setup

For setting up the problem, we declare boundary conditions in **BC.cpp** implementation file and import it into the main file. Below is the implementation file for Boundary Conditions attached.

```
// BC.cpp

#include "BC.hpp"

//-------------PROBLEM SPECIFIC--------------------
// Boundary Terms
float g_x0(float y) {return 0;}
float g_xL(float y) {return y;}
float g_y0(float x) {return (x-1)*sin(x);}
float g_yL(float x) {return x*(2-x);}

// RHS Term
float f(float x, float y) {return 0;}
//-------------PROBLEM SPECIFIC--------------------
```

4 Basis Function Subroutines

For linear basis function we use a generalized definition of basis function as follows:

$$\phi_i = a_i + b_i x_i + c_i y_i \tag{4}$$

$$a_i = \frac{x_j y_k - x_k y_j}{2|K|} \tag{5}$$

$$b_i = \frac{y_j - y_k}{2|K|} \tag{6}$$

$$c_i = \frac{x_k - x_j}{2|K|} \tag{7}$$

where, i = 1,2,3 with cyclic permutation of indices. Here, notice that $\nabla\phi_i$ is constant.

$$\nabla\phi_i = \begin{bmatrix} b_i \\ c_i \end{bmatrix} \tag{8}$$

Using this, we can evaluate the Local Stiffness Matrix as A_{ij}^K:

$$A_{ij}^K = \int_K a\nabla\phi_i \cdot \nabla\phi_j dx \tag{9}$$

$$= (b_i b_j + c_i c_j)\int_K adx \tag{10}$$

$$\approx \bar{a}(b_i b_j + c_i c_j)|K|, \quad i,j = 1,2,3 \tag{11}$$

where $\bar{a} = a(\frac{N_1+N_2+N_3}{3})$ is due to center of gravity value of A on K. Thus, Local Element stiffness matrix thus becomes:

$$A^K = \bar{a}\begin{bmatrix} b_1^2 + c_1^2 & b_1 b_2 + c_1 c_2 & b_1 b_3 + c_1 c_3 \\ b_2 b_1 + c_2 c_1 & b_2^2 + c_2^2 & b_2 b_3 + c_2 c_3 \\ b_3 b_1 + c_3 c_1 & b_3 b_2 + c_3 c_2 & b_3^2 + c_3^2 \end{bmatrix} \tag{12}$$

Now, this local element matrix can be mapped into the global stiffness matrix using node tags stored in a vector in main function.

```cpp
#include "basis.hpp"

// Function pointer alias for callback
typedef double (*alpha)(Array3d, Array3d);

//------------Local Element Stiffness Matrix Calculation----------------
// phi = a + bx + cy;

//--------------------------------------------------------------------
// calculating coefficient 'a(i)'
Array3d a(Array3d x, Array3d y)
{
    Array3d a_ = Array3d::Zero();

    // Round robin
    int j[3] = {1,2,0};
    int k[3] = {2,0,1};

    for (int i = 0; i < 3; i++)
        a_(i) = ( x(j[i])*y(k[i]) - x(k[i])*y(j[i]) ) / 2.0;

    return a_;
}
//--------------------------------------------------------------------


//--------------------------------------------------------------------
// calculating coefficient 'b(i)'
Array3d b(Array3d x, Array3d y)
{
    Array3d b_ = Array3d::Zero();

    int j[3] = {1,2,0};
    int k[3] = {2,0,1};

    for (int i = 0; i < 3; i++)
        b_(i) = ( y(j[i]) - y(k[i]) ) / 2.0;

    return b_;
}
//--------------------------------------------------------------------


//--------------------------------------------------------------------
// calculating coefficient 'c(i)'
Array3d c(Array3d x, Array3d y)
{
    Array3d c_ = Array3d::Zero();
    int j[3] = {1,2,0};
```

```
    int k[3] = {2,0,1};

    for (int i = 0; i < 3; i++)
        c_(i) = ( x(k[i]) - x(j[i]) ) / 2.0;

    return c_;
}
//----------------------------------------------------------------------------


//----------------------------------------------------------------------------
// LOCAL_MATRIX_ASSEMBLER
Array33d LOCAL_MATRIX_ASSEMBLER(Array3d x, Array3d y, alpha al)
{
    double area = poly_area(x,y);

    Array3d b_ = b(x, y)/area;
    Array3d c_ = c(x, y)/area;

    // Set to constant curretly
    double alpha_ = al(x, y);

    Array33d A_K = Array33d::Ones();

    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++)
            A_K(i,j) = b_(i)*b_(j) + c_(i)*c_(j);

    return A_K * alpha_ * area;
}
//----------------------------------------------------------------------------
```

5 Utility Functions in main Solver

To highlight that Gmsh always returns a linear array of nodes and coordinates in concatenated fashion. Thus to check them in terminal we need to separate x, y and z coordinates or three nodes from a list of nodes which identify a specific element. To perform this we perform Operator overloading of ostream operator to handle a vector.

```
template <typename S>
ostream& operator<<(ostream& os, const vector<S>& v)
{
    for (int e = 0; e < v.size(); e++)
        os  <<  "(" << v[e]   << ","
                    << v[++e] << ","
                    << v[++e] << ")"
                    << endl;

    return os;
}
```

Please note that α is passed as a callback function in the Local Matrix generator along with 3 node coordinates for a specific element.

```cpp
double al(Array3d x, Array3d y)
{
    return 1.0;
}
```

6 Miscellaneous functions

This file consists of misc. functions that are required while assembling Global Mass matrix using mesh data. The following two functions are poly_area and isNull. There utility is summarised below:

1. poly_area : Calculates the area of polygon with given nodes.
2. isNull : Checks whether the entry for a corresponding node has already been inserted into the global sparse matrix or not.

```cpp
// mesh_utils.cpp
#include "mesh_utils.hpp"

//------------------------------------------------------------------
double poly_area(ArrayXd x, ArrayXd y)
{
    double area = 0.0;
    area = 0.5*(    x(0)*(y(1)-y(2))
                +   x(1)*(y(2)-y(0))
                +   x(2)*(y(0)-y(1))
                );

    return area;
}
//------------------------------------------------------------------

//------------------------------------------------------------------
bool isNull(const SparseMatrix<double>& mat, int row, int col)
{
    for (SparseMatrix<double>::InnerIterator it(mat, col); it; ++it)
        if (it.row() == row)
            return false;

    return true;
}
//------------------------------------------------------------------
```

7 Main function

The main function is executed in following steps after Elementary Mesh Analysis (EMA):

1. Fetching all node tags from mesh and their co-ordinates (x,y,z) for 2D surface only.
2. Storing node co-ordinates in non-parametric form in Arrays as provided in Eigen template library.

3. Fetching node tags of elements with 3 columns for triangular elements.
4. Assembling Global Stiffness matrix.
5. Imposing Boundary Conditions on physical groups.
6. Solve system of equation.
7. Write solution file.

```
int main( int argc, char *argv[])
{
    cout << setprecision(4);
    string msh_file = "square.msh";

    //----------------------------EMA start--------------------------------
    initialize();
    open(msh_file);
    model::getCurrent(msh_file);
    cout << "Model " << msh_file <<
    " (" << model::getDimension() << "D)" << endl;
    //----------------------------EMA end----------------------------------

    // Consistently return non-parametric form with boundaries included
    const bool return_param_coord = false;
    const bool include_boundary = true;


    //----------------------step 1.) start--------------------------------
    // Fetching node tags and node co-ordinates for 2D surface
    vector<size_t> nodeTags;
    vector<double> nodeCoords, nodeParams;
    int dim = 2, tag = -1;
    model::mesh::getNodes(  nodeTags, nodeCoords, nodeParams,
                            dim, tag,
                            include_boundary, return_param_coord
                            );
    //----------------------step 1.) end----------------------------------


    //----------------------step 2.) start--------------------------------
    // -------Storing node coordinates in 1D Array<double> (x,y)--------
    ArrayXd x = ArrayXd::Zero(nodeTags.size());
    ArrayXd y = ArrayXd::Zero(nodeTags.size());

    int p = -1;
    for (auto &&tag : nodeTags)
    {
        x(tag-1) = nodeCoords[++p];
        y(tag-1) = nodeCoords[++p];
        p++;
    }
```

```
//---------------------step 2.) end-------------------------------


//---------------------step 3.) start-----------------------------
// Fetching node tags of elements. 3 Columns for triangular elements
vector<int> elemTypes;
vector< vector<size_t> > elemTags, elemNodeTags;
model::mesh::getElements(   elemTypes, elemTags,
                            elemNodeTags, 2, -1);

Array<int, Dynamic, Dynamic> El_nodes;
El_nodes = ArrayXXi::Zero(elemNodeTags[0].size()/3, 3);

int row = 0;
for (int i = 0; i < elemNodeTags[0].size(); i++)
{
    El_nodes(row,0) = elemNodeTags[0][i  ];
    El_nodes(row,1) = elemNodeTags[0][++i];
    El_nodes(row,2) = elemNodeTags[0][++i];
    row++;
}

//---------------------step 3.) end-------------------------------

//---------------------step 4.) start-----------------------------
// Assembling global stiffness Matrix
Array3d x_l, y_l;
Array33d A_K;
SparseMatrix<double> A_G(x.size() , x.size());

A_G.reserve(9*elemNodeTags[0].size()/3);

for (int el = 0; el < elemNodeTags[0].size()/3; el++)
{
    // Evaluate Local Matrix
    x_l = Array3d::Zero();
    y_l = Array3d::Zero();
    for (int i = 0; i < 3; i++)
    {   x_l(i) = x(El_nodes(el,i)-1);
        y_l(i) = y(El_nodes(el,i)-1);
    }

    A_K = LOCAL_MATRIX_ASSEMBLER(x_l, y_l, al);
    // Evaluate Local Matrix

    // Inserting Local stiffness into Global Stiffness matrix
    int i = 0;
    for (auto &&row : El_nodes(el,all))
    {
        int j = 0;
```

```
            for (auto &&col : El_nodes(el,all))
            {
                if (isNull(A_G, row-1, col-1))
                    A_G.insert(row-1,col-1) = A_G.coeff(row-1,col-1) \
                                            + A_K(i,j);
                else
                    A_G.coeffRef(row-1,col-1) = A_G.coeff(row-1,col-1) \
                                              + A_K(i,j);
                j++;
            }
            i++;
        }
    }
    //----------------------step 4.) end--------------------------------


    // 0 --> bottom | 1 --> right | 2 --> top | 3 --> left
    //----------------------step 5.) start------------------------------
    // Now Imposing Boundary Conditions
    VectorXd b_RHS = VectorXd::Zero(nodeTags.size());
    VectorXd zeta = VectorXd::Zero(nodeTags.size());
    vectorpair dimTags;
    model::getPhysicalGroups(dimTags, 1);
    // line entities defined as boundaries

    string name;

    for (auto &&dT : dimTags)
    {
        int dim = dT.first, tag = dT.second;
        cout << "dim=" << dim << " tag=" << tag << endl;

        model::mesh::getNodes(nodeTags, nodeCoords,
        nodeParams, dim, tag, true, false);

        cout << "------------------" << endl;
        for (auto &&i : nodeTags)
        {
            cout << i << endl;
            A_G.row(i-1) *= 0.0;
            A_G.coeffRef(i-1, i-1) = 1.0;
        }

        model::getPhysicalName(dim, tag, name);
        for (auto &&i : nodeTags)
        {
            if(name == "bottom")
                b_RHS(i-1) = g_y0(x(i-1));

            else if(name == "top")
```

```
                b_RHS(i-1) = g_yL(x(i-1));

            else if(name == "left")
                b_RHS(i-1) = g_x0(y(i-1));

            else if(name == "right")
                b_RHS(i-1) = g_xL(y(i-1));
        }

        cout << "-----------------------" << endl;
    }

    // cout << "b_RHS = \n" << b_RHS << endl;
    //----------------------step 5.) end---------------------------------

    A_G.makeCompressed();


    //---------------------step 6.) begin-------------------------------
    BiCGSTAB< SparseMatrix<double> > solver;
    solver.compute(A_G);

    zeta = solver.solve(b_RHS);

    cout << "#iterations:     " << solver.iterations() << endl;
    cout << "estimated error: " << solver.error()      << endl;
    //---------------------step 6.) begin-------------------------------

    //---------------------step 7.) begin-------------------------------
    ofstream sol_file("zeta.txt");
    sol_file << "x\t y\t zeta" << endl;
    for (int node = 0; node < x.size(); node++)
    {
        sol_file << x(node)     << "\t"
                 << y(node)     << "\t"
                 << zeta(node)  << endl;
    }
    sol_file.close();
    //---------------------step 7.) end---------------------------------

    finalize();
    return 0;
}
```

8 Compilation and Run command

```
IFLAG="-I /usr/local/include/api"
CC=g++
FLAGS="-Llib -lgmsh -lm"
```

```
MESH=mesh_domain
lc=0.1

SOL_FILE=solver.cpp
BASIS_FILE=basis.cpp
BC_FILE=BC.cpp
MESH_UTIL=mesh_utils.cpp

OUT_FILE=run.out

# -----------Create mesh and get out-----------
g++ -o $MESH.out $IFLAG $MESH.cpp -Llib -lgmsh
./$MESH.out $lc
# -----------Create mesh and get out-----------

# --------Solve Diffusion Equation-------------
$CC -c $SOL_FILE -I .
$CC -c $BASIS_FILE -I .
$CC -c $BC_FILE -I .
$CC -c $MESH_UTIL -I .

$CC -o $OUT_FILE \
$SOL_FILE $BASIS_FILE $BC_FILE $MESH_UTIL $IFLAG $FLAGS

./$OUT_FILE

# --------------Remove .out files--------------
rm *.o
rm *.out
```

Header files

8.1 Headers for main function

```
// headers.hpp
#ifndef __HEADERS_HPP
// Include guard

#define __HEADERS_HPP

#include <iostream>
#include <eigen3/Eigen/Dense>
#include <eigen3/Eigen/Sparse>
#include <gmsh.h>
#include <string>
#include <cmath>
#include <vector>
#include <iomanip>
```

```
#include <fstream>

#include "BC.hpp"
#include "mesh_utils.hpp"
#include "basis.hpp"

#endif
```