HiperLife Tutorial: Poisson Equation

Arash Imani*

LaCàN

November 3, 2024

1 Problem Definition

The Poisson equation extends from the simpler Laplace equation, which assumes no sources. When sources are present, the Poisson equation becomes useful in describing how fields, such as electric or gravitational potential fields, are influenced by these sources. The Poisson equation appears frequently in classical physics and underlies much of the theoretical framework for modeling potential fields such as Electrostatics, Gravitational Potential, Heat Conduction and Fluid Mechanics.The Poisson equation's versatility comes from its ability to connect a spatially varying source (charge, mass, heat, etc.) with a field (potential, temperature, pressure, etc.) that responds to that source. Solving the Poisson equation in any given physical context allows scientists and engineers to predict how a system will respond to internal sources, like an electric charge distribution or a temperature source, and adjust designs accordingly.

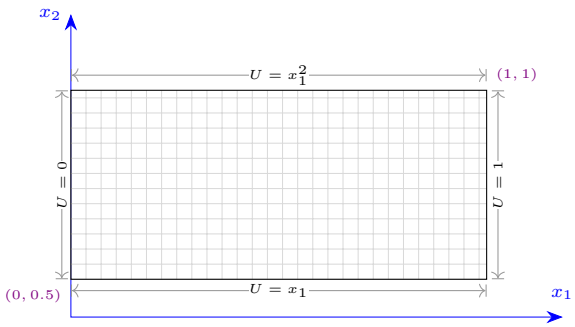

Figure 1: Geometry, BC and computational domain used for the analysis of Poisson problem.

In practical applications, boundary conditions are often applied to solve the Poisson equation in specific regions, which is essential for real-world systems that are typically bounded by physical surfaces. The equation is solved using methods such as separation of variables, Green's functions, or numerical techniques like finite difference or finite element methods, especially in complex geometries.

2 Governing Equations

Poisson Equation is a simple elliptic model, given by

$$-\Delta U = -\nabla^2 U = -\frac{\partial^2 U}{\partial x_1^2} - \frac{\partial^2 U}{\partial x_2^2} = f \tag{1}$$

*Disclaimer: The implemented code (as can be seen in its respective section) is a part of HiperLife primary tutorials, so it is programmed by others.

We will use this equation in this example for introducing the implemention of finite element method in the HiperLife. Notice that here we have used $f = 0$. As shown in Figure 1, We have used homogeneous Dirichlet $(U = 0, \quad U = 1)$ along the lines $(x_1 = 0, \quad x_1 = 1)$, and along the lines $(x_2 = 0.5, \quad x_2 = 1)$ we applied Inhomogenous Dirichelet $(U = x_1, \quad U = x_1^2)$, respectively.

3 Weak Form

The starting point for the development of the finite element model of Eq. (1) is its weak form. The variational formulation of our model problem reads: Find $u \in V$ such that

$$\mathcal{F}(u; v) = 0 \quad \forall v \in \hat{V} \tag{2}$$

where

$$\mathcal{F}(u; v) = -\int_\Omega v\nabla \cdot \nabla u - vf \mathrm{d}\Omega \,. \tag{3}$$

and

$$\begin{aligned} &\hat{V} = \{v \in H^1(\Omega) : v = 0 \text{ on } \partial\Omega\}, \\ &V = \{v \in H^1(\Omega) : v = 0 \text{ on } x_1 = 0,\ v = 1 \text{ on } x_1 = 1,\ v = x_1 \text{ on } x_2 = 0.5,\ v = x_1^2 \text{ on } x_2 = 1\} \,. \end{aligned} \tag{4}$$

where v is a test function, which will be equated, in the our FE model to the interpolation function used for u.The discrete problem arises as usual by restricting V and $\hat{V}$ to a pair of discrete spaces. using integrating by part this expression over Ω, we have

$$\mathcal{F}(u; v) = -\int_\Omega \nabla \cdot [v\nabla u] \ \mathrm{d}\Omega + \int_\Omega \nabla v \nabla u \ \mathrm{d}\Omega - \int_\Omega vf \ \mathrm{d}\Omega \,. \tag{5}$$

Using Gauss's theorem we get

$$\mathcal{F}(u; v) = -\int_\Gamma [v\nabla u] \cdot \mathbf{n} \ \mathrm{d}\Gamma + \int_\Omega \nabla v \nabla u \ \mathrm{d}\Omega - \int_\Omega vf \ \mathrm{d}\Omega \,. \tag{6}$$

By applying boundary condition since we do not have any flux, the final relation of weak form would be as

$$\mathcal{F}(u; v) = \int_\Omega \nabla v \nabla u \ \mathrm{d}\Omega - \int_\Omega vf \ \mathrm{d}\Omega \,. \tag{7}$$

4 Finite Element Model

For Ritz-Galerkin FE model, the choice of the test functions is restricted to the spaces of approximation functions used for the solution field. Suppose that the variable u approximated by expansions of the form

$$u(\mathbf{x}) = \sum_{j=1}^{M} \phi_j(\mathrm{x}) u_j = \mathbf{\Phi}^T \mathbf{u} \,. \tag{8}$$

By substituting Eq. (8) in Eq. (7) we get

$$\mathcal{F}(u; \phi) = \int_\Omega \nabla \phi_i \nabla \phi_j u_j \ \mathrm{d}\Omega - \int_\Omega \phi_i f_i \ \mathrm{d}\Omega \,. \tag{9}$$

The above equations can be written symbolically in matrix form as

$$\mathbf{K}\mathbf{u} = \mathbf{F} \,. \tag{10}$$

The coefficient matrices $\mathbf{K}$ and $\mathbf{F}$ shown in Eq. (15) are defined as

$$\begin{aligned} K(i,j) &= \int_{\Omega_T} \left(\frac{\partial \phi_i}{\partial x_1}\frac{\partial \phi_j}{\partial x_1} + \frac{\partial \phi_i}{\partial x_2}\frac{\partial \phi_j}{\partial x_2}\right) dA \\ F(i) &= \int_{\Omega_T} \phi_i f_i \mathrm{d}A \end{aligned} \tag{11}$$

keep in mind that in our case $f = 0$, too. Note that $\mathrm{d}A = \mathrm{d}x_1 \times \mathrm{d}x_2 = jac\mathrm{d}\xi\mathrm{d}\eta$, which $jac = \det(Jacobian)$[1]. The elemental representation of the vector and matrix required for implementation in the Hiperlife would have the format of

$$Ak(i,j) = jac \times \left[\frac{\partial \phi_i}{\partial x_1}\frac{\partial \phi_j}{\partial x_1} + \frac{\partial \phi_i}{\partial x_2}\frac{\partial \phi_j}{\partial x_2}\right],$$
$$Bk(i) = jac \times [\phi_i f]\,. \tag{12}$$

5 Choice of Elements

Thus, for this simple problem every Lagrange and serendipity family of interpolation functions are admissible for the interpolation of the temperature field, our choice is would be linear quadrilateral element.Linear Quadrilateral Elements is the simplest quadrilateral element consists of four nodes. The associated interpolation functions for geometry and field variables are bilinear.

$$\mathbf{\Phi}_I(\xi,\eta) = \frac{1}{4}(1+\xi_I\xi)(1+\eta_I\eta) \quad (I \text{ from 1 to 4}) \tag{13}$$

where ξ_I and η_I are the corner coordinates at element T in domain of $\Omega_T \in (-1,1)^2$. As it shown in Figure 2 we are using 2×2 Gauss–Legendre quadrature integration.

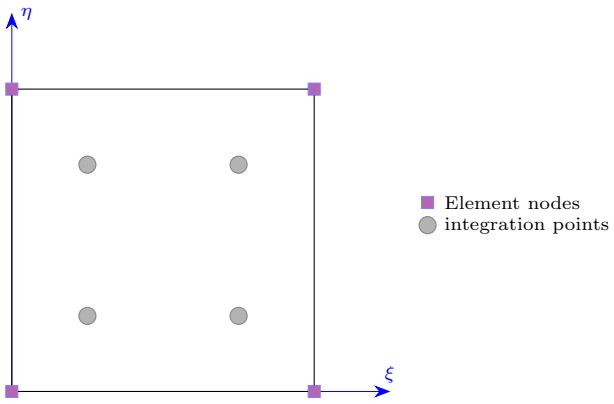

Figure 2: Linear quadrilateral element used for finite element model.

In this example each node only have one degree of freedom and for the purpose of discretization we use 500×500 uniform mesh.

6 Implementation

In this section, we present the implementation of our solution in the Hiperlife. The program is divided into three separate files, main part which we create our problem by the Hiperlife headers, auxiliary header where we introduce parameters and declare defined functions, and at last auxiliary file, where we define some functions which provide required matrices like the tangent matrix. As it is already mentioned this code is from the Hiperlife primary tutorials and for any use the Copyright policy and License should be check from the project homepage[2]. The Flowchart of a typical HiperLife program is shown in Figure 3.

6.1 Poisson.cpp

[1] for more details see the Appendix
[2] https://hiperlife.gitlab.io/hiperlife

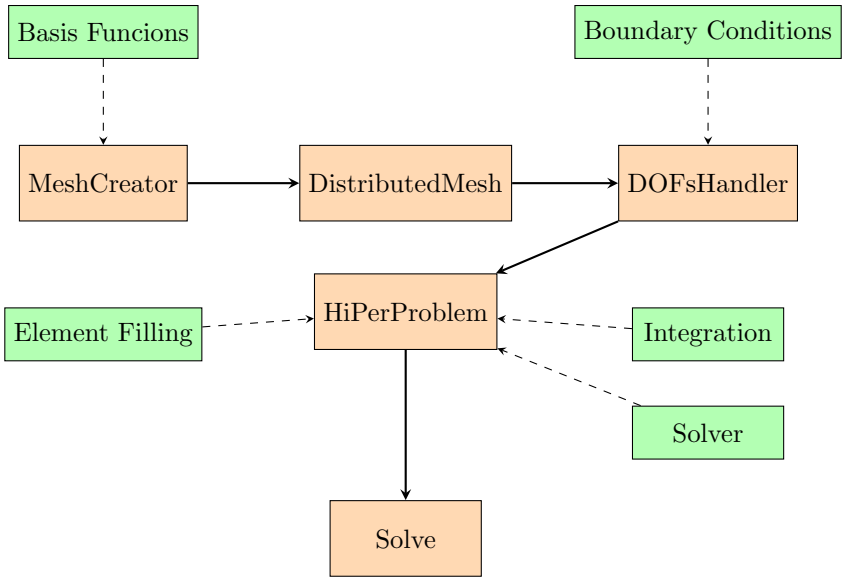

Figure 3: Flow chart.

```
/*
*******************************************************************************
Copyright (c) 2017-2023 Team hiperlife
Authors: Daniel Santos-OlivAn, Alejandro Torres-SAnchez and Guillermo Vilanova
Contributors:
*******************************************************************************
This file is part of hiperlife
Project homepage: https://hiperlife.gitlab.io/hiperlife
Distributed under the GNU General Public License, see the accompanying
file LICENSE or https://opensource.org/license/gpl-3-0.
*******************************************************************************
*/



// hiperlife headers
#include "hl_Core.h"
#include "hl_ParamStructure.h"
#include "hl_Parser.h"
#include "hl_TypeDefs.h"
#include "hl_GlobalBasisFunctions.h"
#include "hl_StructMeshGenerator.h"
#include "hl_MeshLoader.h"
#include "hl_DistributedMesh.h"
#include "hl_FillStructure.h"
#include "hl_DOFsHandler.h"
#include "hl_HiPerProblem.h"
#include "hl_LinearSolver_Iterative_AztecOO.h"
#include "hl_LinearSolver_Direct_MUMPS.h"

//using another header file for filling the elemental linear
#include "AuxPoisson.h"

int main(int argc, char** argv)
{
        using namespace std;
        using namespace hiperlife;

        // ************************************************************//
        // *****                    INITIALIZATION                 *****//
```

```
41          // ********************************************************//
42          hiperlife::Init(argc, argv);
43          SmartPtr<ParamStructure> paramStr = CreateParamStructure<PoissonParams>();
44          // ********************************************************//
45          // *****                  MESH CREATION                 *****//
46          // ********************************************************//
47          //====================== Mesh generation ====================//
48
49          SmartPtr<StructMeshGenerator> structMesh = Create<StructMeshGenerator>();
50          structMesh->setNDim(3);
51          structMesh->setBasisFuncType(BasisFuncType::Lagrangian);
52          structMesh->setBasisFuncOrder(1);
53          structMesh->setElemType(ElemType::Square);
54          structMesh->genRectangle(500, 500, 1.0, 0.5);
55          structMesh->translateY(0.5);
56          //==========             Distribute mesh           ==========//
57          SmartPtr<DistributedMesh> disMesh = Create<DistributedMesh>();
58          disMesh->setMesh(structMesh);
59          disMesh->setBalanceMesh(true);
60          disMesh->Update();
61
62          // ********************************************************//
63          // *****              DOFsHANDLER CREATION              *****//
64          // ********************************************************//
65
66          //==========            Create DOFHandler            ==========//
67          SmartPtr<DOFsHandler> dofHand = Create<DOFsHandler>(disMesh);
68          dofHand->setNameTag("dofHand");
69          dofHand->setNumDOFs(1);
70          dofHand->Update();
71          //==========           Boundary conditions           ==========//
72          dofHand->setBoundaryCondition(0, 0.0);
73          dofHand->setBoundaryCondition(0, MAxis::Xmax, 1.0);
74          dofHand->setBoundaryCondition(0, MAxis::Ymin, [](double x){return x;});
75          dofHand->setBoundaryCondition(0, MAxis::Ymax, [](double x){return x*x;});
76          dofHand->UpdateGhosts();
77
78
79          // ********************************************************//
80          // *****              HIPERPROBLEM CREATION             *****//
81          // ********************************************************//
82          SmartPtr<HiPerProblem> hiperProbl = Create<HiPerProblem>();
83          hiperProbl->setParameterStructure(paramStr);
84          hiperProbl->setDOFsHandlers({dofHand});
85
86          //==========            Set integration              ==========//
87          hiperProbl->setIntegration("Integ", {"dofHand"});
88          hiperProbl->setCubatureGauss("Integ", 3);
89          hiperProbl->setElementFillings("Integ", ElementFilling);
90
91          //==========                Update                   ==========//
92          hiperProbl->Update();
93
94          //==========              Set solver                 ==========//
95
96          SmartPtr<MUMPSDirectLinearSolver> solver = Create<MUMPSDirectLinearSolver>();
97          solver->setHiPerProblem(hiperProbl);
98          solver->setVerbosity(MUMPSDirectLinearSolver::Verbosity::Extreme);
99          solver->setDefaultParameters();
100         solver->Update();
101
102         solver->solve();
103         hiperProbl->UpdateSolution();
104
105         // The results
106         dofHand->printFileLegacyVtk("Poisson");
107         // for visualization purposes.
108         ofstream rhsFile;
```

```
109         rhsFile.open(paramStr->getStringParameter(PoissonParams::filemesh)
110         +to_string(dofHand->myRank())+".txt");
111         hiperProbl->sol->Print(rhsFile);
112         rhsFile.close();
113
114
115         // ***************************************************************//
116         // *****                      FINALIZE                       *****//
117         // ***************************************************************//
118
119         hiperlife::Finalize();
120         return 0;
121 }
```

6.2 AuxPoisson.h

```
1  /*
2  ********************************************************************************
3  Copyright (c) 2017-2023 Team hiperlife
4  Authors: Daniel Santos-Olivan, Alejandro Torres-Sanchez and Guillermo Vilanova
5  Contributors:
6  ********************************************************************************
7  This file is part of hiperlife
8  Project homepage: https://hiperlife.gitlab.io/hiperlife
9  Distributed under the GNU General Public License, see the accompanying
10 file LICENSE or https://opensource.org/license/gpl-3-0.
11 ********************************************************************************
12 */
13
14
15 #ifndef AUXPoisson_H
16 #define AUXPoisson_H
17
18 // C headers
19 #include <iostream>
20
21 // hiperlife headers
22 #include "hl_ParamStructure.h"
23 #include "hl_FillStructure.h"
24 #include "hl_DOFsHandler.h"
25 #include "hl_HiPerProblem.h"
26
27 struct PoissonParams
28 {
29         enum RealParameters
30         {
31                 force
32         };
33         enum StringParameters
34         {
35                 filemesh
36         };
37         HL_PARAMETER_LIST DefaultValues
38         {
39                 {"force", 0.0},
40                 {"filemesh", ""},
41         };
42 };
43
44
45 void ElementFilling(hiperlife::FillStructure& fillStr);
46
47 #endif
```

6.3 AuxPoisson.cpp

```
// hiperlife headers
#include "hl_Core.h"
#include "hl_ParamStructure.h"
#include "hl_Parser.h"
#include "hl_TypeDefs.h"
#include "hl_GlobalBasisFunctions.h"
#include "hl_StructMeshGenerator.h"
#include "hl_DistributedMesh.h"
#include "hl_FillStructure.h"
#include "hl_DOFsHandler.h"
#include "hl_HiPerProblem.h"
#include "hl_LinearSolver_Iterative_AztecOO.h"
#include "hl_LinearSolver_Direct_MUMPS.h"
#include "AuxPoisson.h"

void ElementFilling(hiperlife::FillStructure& fillStr)
{
        using namespace std;
        using namespace hiperlife;
        using namespace hiperlife::Tensor;

        //==========          Initialize variables          ==========//
        SubFillStructure& subFill = fillStr["dofHand"];
        //   which we use to declare the variables:
        int eNN  = subFill.eNN;       // Number of neighbor nodes
        int pDim = subFill.pDim;      // Number of parametric dimension
        wrapper<double,1>  bf(subFill.nborBFs(), eNN);;
        tensor<double,2> Dbf_g(eNN,pDim);     //[dN_i/dx_1 dN_i/dx_2]
        double jac; //Jacobian (differential of line/area/volume)
        GlobalBasisFunctions::gradients(Dbf_g, jac, subFill);


        // We are going to fill the matrix and the right hand side. Here, we define
        //   the matrix Ak (eNN*eNN) and the vector Bk (eNN).
        wrapper<double,2> Ak(fillStr.Ak(0, 0).data(),eNN,eNN);
        wrapper<double,1> Bk(fillStr.Bk(0).data(),eNN);

        //====================  Fill linear system  ====================//
        //
        // We loop through the neighboring nodes to fill both the matrix
        //    and the right-hand side at the same time

        // Fill matrix
        //Ak = jac * Dbf_g * Dbf_g.T();
        // Fill RH
        //Bk = jac * bf * force;

        //using ttl::index::i, ttl::index::j, ttl::index::a;
        // Fill matrix
        //Ak = (jac * Dbf_g(i,a) * Dbf_g(j,a))(i,j);
        // Fill RH
        //Bk = (jac * bf(i) * force)(i);

        for (int i = 0; i < eNN; i++)
        {
                for (int j = 0; j < eNN; j++)
                {
                        // Compute the product of the gradient of the basis functions
                        double dotij{};
                        for (int d = 0; d < pDim; d++)
                        dotij += Dbf_g(i,d)*Dbf_g(j,d);

                        // Fill matrix
                        Ak(i,j) += jac * dotij;
                }

```

```
67                      // Fill RHS
68                      Bk(i) += jac * bf(i) * force;
69              }
70
71      }
```

7 Results

In this section, we present the results of our solution. The contour demonstration of result u is also shown in Figure 4.

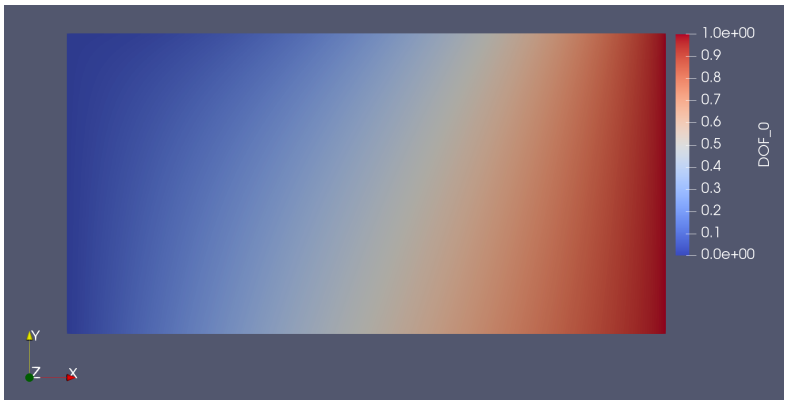

Figure 4: Solution.

Appendix

In this section we present the derivation behind hiperlife function that calculates the gradients of basis functions and Jacobian. by the linear mapping between (ξ, η) and (x_1, x_2), we can define $\frac{\partial \phi}{\partial x_1}$ and $\frac{\partial \phi}{\partial x_2}$

$$\begin{aligned} \frac{\partial \phi}{\partial \xi} &= \frac{\partial \phi}{\partial x_1}\frac{\partial x_1}{\partial \xi} + \frac{\partial \phi}{\partial x_2}\frac{\partial x_2}{\partial \eta} \\ \frac{\partial \phi}{\partial \eta} &= \frac{\partial \phi}{\partial x_1}\frac{\partial x_1}{\partial \xi} + \frac{\partial \phi}{\partial x_2}\frac{\partial x_2}{\partial \eta} \end{aligned} \tag{14}$$

Since $x = x(\xi, \eta)$, we get

$$\begin{bmatrix} dx_1 \\ \\ dx_2 \end{bmatrix} = \begin{bmatrix} \frac{\partial x_1}{\partial \xi} & \frac{\partial x_2}{\partial \xi} \\ \\ \frac{\partial x_1}{\partial \eta} & \frac{\partial x_2}{\partial \eta} \end{bmatrix} \begin{bmatrix} d\xi \\ \\ d\eta \end{bmatrix} \tag{15}$$

by defining Jacobian as

$$J = \begin{bmatrix} \frac{\partial x_1}{\partial \xi} & \frac{\partial x_2}{\partial \xi} \\ \\ \frac{\partial x_1}{\partial \eta} & \frac{\partial x_2}{\partial \eta} \end{bmatrix} \tag{16}$$

so we can rewrite the Eq. (14) as

$$\begin{bmatrix} \frac{\partial \phi}{\partial x_1} \\ \\ \frac{\partial \phi}{\partial x_2} \end{bmatrix} = J^{-1} \begin{bmatrix} \frac{\partial \phi}{\partial \xi} \\ \\ \frac{\partial \phi}{\partial \eta} \end{bmatrix} \tag{17}$$

where J^{-1} is

$$J^{-1} = \frac{1}{\det J} \begin{bmatrix} \frac{\partial x_2}{\partial \eta} & -\frac{\partial x_2}{\partial \xi} \\ \\ -\frac{\partial x_1}{\partial \eta} & \frac{\partial x_1}{\partial \xi} \end{bmatrix} \tag{18}$$

so by the known relation for basis functions like Eq. (13), one can calculate the jacobian and the gradient of basis function.