Tutorial on FEniCS: solving 2D Poisson equation

Hanfeng Zhai
hzhai@stanford.edu

January 4, 2025

Introduction

This tutorial demonstrates how to solve the Poisson equation using the finite element method (FEM) with the FEniCS library[1]. The Poisson equation is a widely used partial differential equation (PDE) that models physical phenomena such as heat conduction, electrostatics, and diffusion.

Mathematical Formulation

The Poisson equation in two dimensions is written as:

$$-\nabla \cdot (k \nabla u) = f \quad \text{in } \Omega, \tag{1}$$

where:

- u is the unknown scalar field (e.g., temperature).
- k is the thermal conductivity (assumed constant in this example).
- f is the source term (e.g., heat generation). $f = 0$ in this example.
- Ω is the computational domain ($\mathbb{R}^2$).

The boundary conditions are defined as:

$$u = g_D \quad \text{on } \Gamma_D, \tag{2}$$

$$-k\frac{\partial u}{\partial n} = g_N \quad \text{on } \Gamma_N, \tag{3}$$

where Γ_D and Γ_N are Dirichlet and Neumann boundaries, respectively, and n is the outward normal vector.

In this example, we solve Equation (1) with Dirichlet boundary conditions on all boundaries.

Implementation in FEniCS

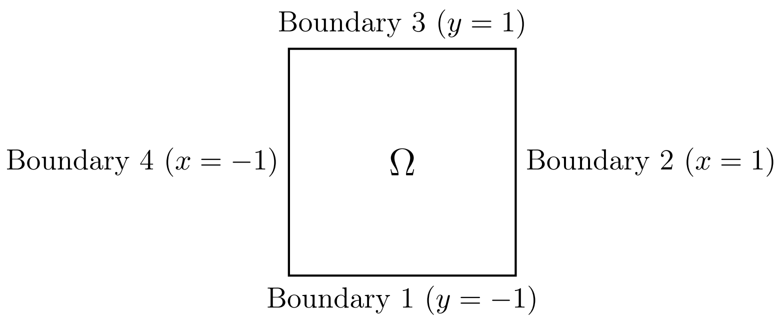
The following Python code implements the solution of the Poisson equation using FEniCS. The computational domain is a square, and the Dirichlet boundary conditions set $u = 1000$ on all edges of the domain. The source term is constant, $f = 0.2$, and $k = 2 \times 10^{-4}$. The solution is visualized as a heatmap and as a 3D surface plot.

[1]We use FEniCS 2019 version

Schematic of the Domain

The computational domain is a square defined as $[-1,1] \times [-1,1]$. The boundaries are labeled as follows:

- Boundary 1: $y = -1$,
- Boundary 2: $x = 1$,
- Boundary 3: $y = 1$,
- Boundary 4: $x = -1$.



Key Steps in the Code

1. **Mesh Generation:** The domain is discretized using a triangular mesh generated by Gmsh and converted for use in FEniCS.
2. **Boundary Conditions:** Dirichlet boundary conditions are applied on all edges of the square.
3. **Variational Formulation:** The weak form of the Poisson equation is derived as:
$$\int_\Omega k \nabla u \cdot \nabla v \, \mathrm{d}x = \int_\Omega f v \, \mathrm{d}x, \tag{4}$$
where v is the test function.
4. **Solution Computation:** The linear system resulting from the discretization is solved, yielding the scalar field u.
5. **Visualization:** The solution is plotted as a 2D heatmap and a 3D surface.

```
[ ]: # installing required packages
try:
    import dolfin
except ImportError:
    !wget "https://fem-on-colab.github.io/releases/fenics-install-real.sh" -O "/
 ↪tmp/fenics-install.sh" && bash "/tmp/fenics-install.sh"
    import dolfin
!pip install meshio
!apt-get install gmsh
!gmsh --version
!pip install --upgrade gmsh
```

We begin with importing the necessary packages.

```
[ ]: import gmsh, meshio
     from fenics import *
     import matplotlib.pyplot as plt
     from matplotlib.tri import Triangulation
     from dolfin import *
     import numpy as np
     from mesh_converter import msh_to_xdmf
```

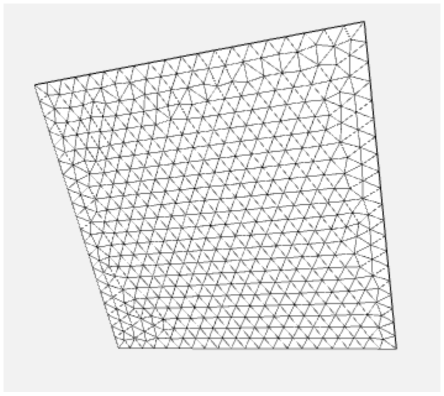
The mesh is defined accordingly given the geometry.

```
[ ]: def Tutorialmesh(Hmax, elementOrder, elementType):
         # Given Hmax, construct a mesh to be read by FEniCS
         gmsh.initialize()
         gmsh.model.add('Tutorialmesh')
         meshObject = gmsh.model

         # Points for the outer boundary
         point1 = meshObject.geo.addPoint(-1,-1,0,Hmax, 1)
         point2 = meshObject.geo.addPoint(1,-1,0,Hmax, 2)
         point3 = meshObject.geo.addPoint(1,1,0,Hmax, 3)
         point4 = meshObject.geo.addPoint(-1,1,0,Hmax, 4)

         # Construct lines from points
         line1 = meshObject.geo.addLine(1, 2, 101)
         line2 = meshObject.geo.addLine(2, 3, 102)
         line3 = meshObject.geo.addLine(3, 4, 103)
         line4 = meshObject.geo.addLine(4, 1, 104)

         # Construct closed curve loops
         outerBoundary = meshObject.geo.addCurveLoop([line1, line2, line3, line4],␣
      ↪201)

         # Define the domain as a 2D plane surface with holes
         domain2D = meshObject.geo.addPlaneSurface([outerBoundary], 301)

         # Synchronize gmsh
         meshObject.geo.synchronize()

         # Add physical groups for firedrake
         meshObject.addPhysicalGroup(2, [301], name='domain')

         meshObject.addPhysicalGroup(1, [line2], 1)
         meshObject.addPhysicalGroup(1, [line3], 2)
         meshObject.addPhysicalGroup(1, [line4], 3)
         meshObject.addPhysicalGroup(1, [line1], 4)
```

```
    # Set element order
    meshObject.mesh.setOrder(elementOrder)

    if elementType == 2:
      # Generate quad mesh from triangles by recombination
      meshObject.mesh.setRecombine(2, domain2D)

    # Generate the mesh
    gmsh.model.mesh.generate(2)
    gmsh.write('mesh.msh')

    gmsh.finalize()
    mesh_from_gmsh = meshio.read("mesh.msh")

    triangle_mesh = meshio.Mesh(
        points=mesh_from_gmsh.points,
        cells={"triangle": mesh_from_gmsh.get_cells_type("triangle")},
    )

    meshio.write("mesh.xml", triangle_mesh)
    msh_to_xdmf('mesh')
    mesh = Mesh("mesh.xml")

    return mesh
```

The mesh of the domain is generated via

[ ]:
```
# Generate the mesh
elementOrder = 1 # Polynomial order in each element (integer)
elementType = 1  # 1 - Triangle; 2 - Quad
HMax = 0.1

mesh = Tutorialmesh(HMax, elementOrder, elementType)
'''visualize the mesh (the object)'''
mesh
```



```
[ ]: <dolfin.cpp.mesh.Mesh at 0x7fd76fd93f10>
```

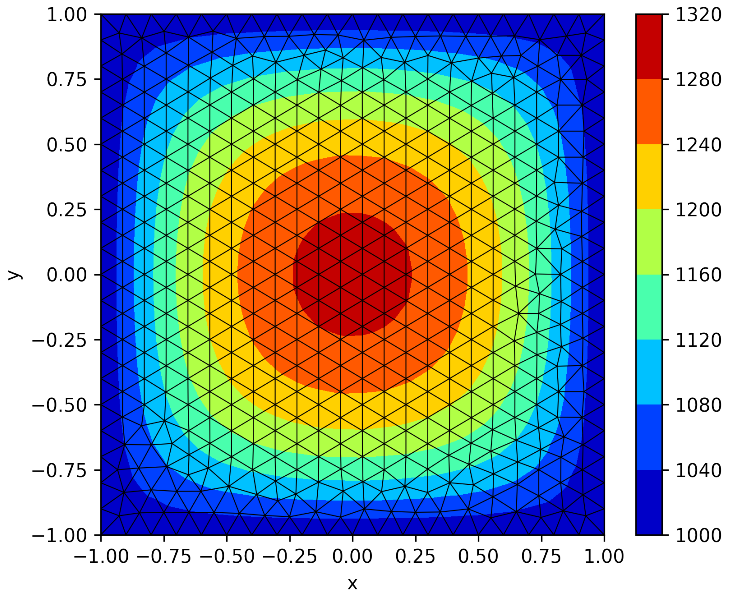
```
[ ]: mesh = Mesh()
     with XDMFFile('mesh' + ".xdmf") as infile:
         infile.read(mesh)
     # load boundary markers from facet file
     mvc_facet = MeshValueCollection("size_t", mesh, mesh.topology().dim() - 1)
     with XDMFFile('mesh' + "_facets.xdmf") as infile:
         infile.read(mvc_facet, "facet_marker")
     boundaries = cpp.mesh.MeshFunctionSizet(mesh, mvc_facet)
     ds = Measure("ds", subdomain_data=boundaries)
```

```
[ ]: V = FunctionSpace(mesh, "CG", 1)
     boundary_markers = MeshFunction("size_t", mesh, mesh.topology().dim()-1)

     DirBC = [DirichletBC(V, Constant(1000.0), boundaries, marker) for marker in␣
      ↪[1,2,3,4]]
     bcs = DirBC
```

```
[ ]: ke = Constant(2e-4)

     f = Constant(0.2)
     u = TrialFunction(V)
     v = TestFunction(V)

     a = ke * inner(grad(u), grad(v)) * dx
     L = dot(f, v) * dx

     u_sol = Function(V)

     solve(a == L, u_sol, bcs)
```

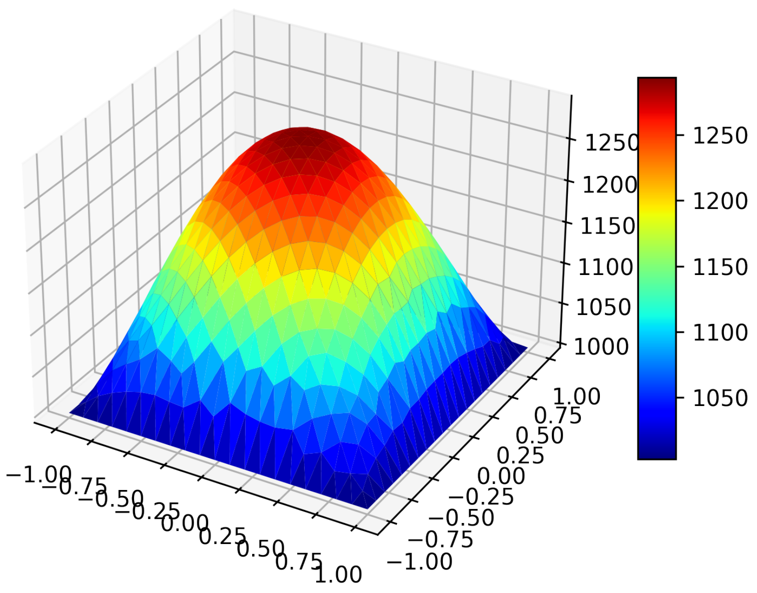
```
[ ]: coordinates = mesh.coordinates()
     values = u_sol.compute_vertex_values(mesh)
     x, y = coordinates[:, 1], coordinates[:, 0]

     triang = Triangulation(x, y, mesh.cells())
     plt.figure(figsize=(6,5))

     plt.tricontourf(triang, values, cmap='jet')
     plt.colorbar()
     plt.triplot(triang, 'k-', lw=0.5)
     plt.xlabel('x')
     plt.ylabel('y')
     plt.savefig(f'heat_2d', dpi=300, transparent=True); plt.show()
```

```
point = Point(0.25, 0.25)
u_value = u_sol(point)
print(f"Solution at point {point}: {u_value}")
```



```
Solution at point <dolfin.cpp.geometry.Point object at 0x7fd76fd9c430>:
1263.3179567734976
```

[ ]:
```
fig = plt.figure(figsize=(5, 5))
ax = fig.add_subplot(111, projection='3d')

x, y, z = triang.x, triang.y, values

ax.plot_trisurf(x, y, z, triangles=triang.triangles, cmap='jet',␣
 ↪edgecolor='none')

mappable = ax.collections[0]   # Extract the mappable object

plt.tight_layout()
fig.colorbar(mappable, ax=ax, shrink=0.5, aspect=10)
plt.savefig('heat_3d', dpi=300, transparent=True)
plt.show()
```



The code can be accessed via `Google Colab`.

Summary

This coding procedure outlined a systematic approach to simulating and visualizing heat conduction in a square domain using Python. The key steps included:

- Defining the computational domain with clear specifications for the grid and material properties.
- Applying boundary conditions to model the physical constraints accurately.
- Solving the governing equations using numerical methods for heat transfer.
- Visualizing the results to gain insights into the temperature distribution across the domain.

Students are encouraged to experiment with the provided framework by modifying the boundary conditions, such as changing the fixed temperatures or implementing insulated boundaries. Observing the resulting changes in temperature distribution provides a deeper understanding of how boundary conditions influence the system's behavior. This iterative process fosters critical thinking and reinforces concepts of heat transfer and numerical modeling.