Worked Example: 2D Poisson's Problem

DC Sweeney

December 9, 2017

Contents

1 Problem Definition

Given the problem

$$-\nabla^2 u(x,y) = 0$$

on a rectangular domain $\Omega := [a,0] \times [0,b]$ with the boundary conditions

$$u(x,0) = u(x,b) = 0, \qquad u(a,y) = \sin(y), \qquad u(0,y) = \sin(y),$$

solve for $u(x,y)$ everywhere in Ω.

2 Meshing

Before we are able to really solve anything, we must have a good grasp of what the domain looks like. We will generate a rectangular mesh composed of $N \times M$ linear elements over $\Omega := [a,0] \times [0,b]$. Therefore, there will be $(N+1) \times (M+1)$ nodal values.

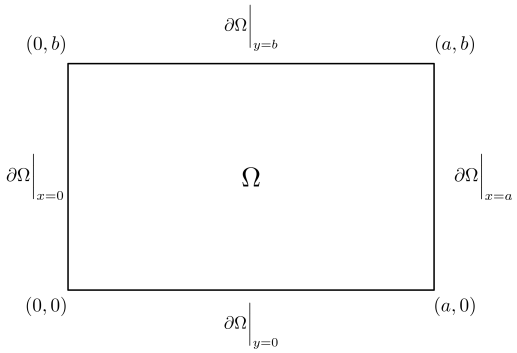

Figure 1: Rectangular domain $\Omega := [a,0] \times [0,b]$ and the boundary conditions prescribed to it.

2.1 Meshing Code

```
%% 2D Diffusion on a Rectangular Plane (Dirichlet Boundaries)
xmin = 0; xmax = 10;
ymin = 0; ymax = 10;
N = 10;
M = 10;
xvals = linspace(xmin, xmax, N+1);
yvals = linspace(ymin, ymax, M+1);

% Initialize vert_list because it's good practice
verts = zeros((N+1)*(M+1),2);

```

```
hold on
for i=1:N+1
    for j = 1:M+1
        % How do we create a list of vertex points?
        verts(i+(N+1)*(j-1),:) = [xvals(i), yvals(j)];
    end
end

%Initialize elements matrix for good practice
elements = zeros(N*M, 4);

for i = 1:N
    for j = 1:M

        % global vertex label for local vertex 1
        v1 = i+(N+1)*(j-1);

        % global vertex label for local vertex 2
        v2 = i+1+(N+1)*(j-1);

        % global vertex label for local vertex 3
        v3 = i+1+(N+1)*j;

        % global vertex label for local vertex 4
        v4 = i+(N+1)*j;

        % Add global element labels into a list of elements
        elements(i+(j-1)*N,:) = [v1, v2, v3, v4];
    end
end
```

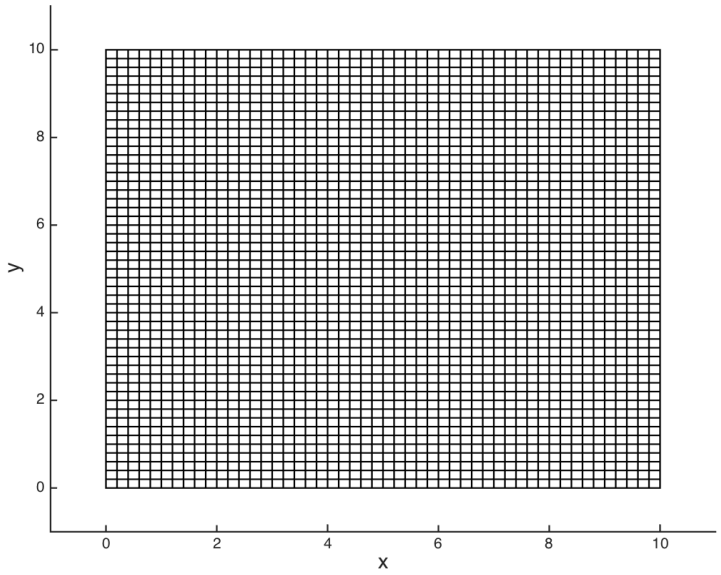

Figure 2: 2-D linear 50×50 element rectangular mesh generated in MATLAB.

3 Weak Form

NOTE: We drop the arguments from the function $u(x, y)$ and $v(x, y)$ such that in the above statement $u(x, y) = u$ and $v(x, y) = v$ for clarity in the notation.

We begin by integrating both sides over the whole domain Ω and then multiplying both sides by a test function $v(x, y)$ as

$$-\int_{\Omega}(v\nabla^2 u)dA = 0.$$

Notice that we are able to apply the product rule $\nabla\cdot(v\nabla u) = v\nabla^2 u+\nabla v\nabla u$ to the above expression to give

$$\begin{aligned} -\int_{\Omega}(v\nabla^2 u)dA &= \int_{\Omega}\Big(\nabla v\nabla u - \nabla\cdot(v\nabla u)\Big)dA \\ &= \int_{\Omega}(\nabla v\nabla u)dA - \int_{\Omega}\Big(\nabla\cdot(v\nabla u)\Big)dA. \end{aligned}$$

We recall the divergence theorem in two dimensions

$$\int_{\Omega}(\nabla\cdot\mathbf{X})dA = \oint_{\partial\Omega}(\mathbf{X}\cdot\mathbf{n})ds,$$

which states that the flux of vector $\mathbf{X}$ emanating from area $\Omega \in \mathbb{R}^2$ is equal to the flux of vector $\mathbf{X}$

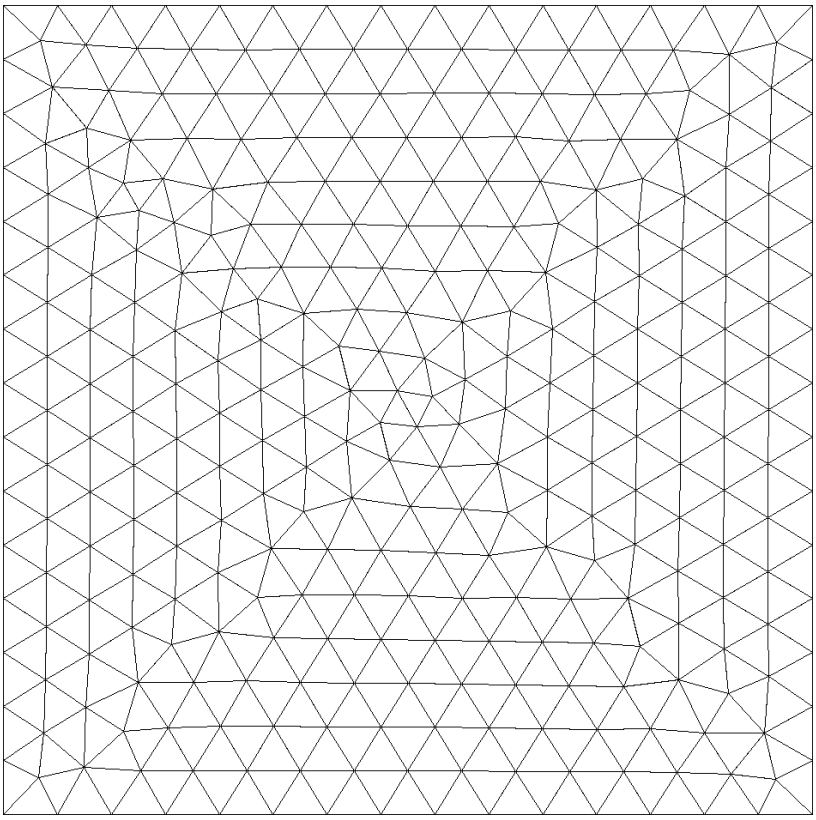

Figure 3: 2-D linear triangular mesh generated in COMSOL.

out of the bounding surface $\partial\Omega$. Using this principle, we modify our expression as

$$\int_\Omega (\nabla v \cdot \nabla u) dA - \int_\Omega \Big(\nabla \cdot (v \nabla u) \Big) dA = \int_\Omega (\nabla v \nabla u) dA - \int_{\partial\Omega} (v \nabla u \cdot \mathbf{n}) dS.$$

We now inspect the problem for boundary conditions dealing with the flux of our primary variable $u(x, y)$ through a domain boundary.

As for the present problem, we may now rewrite the weak form of our original equation in the bilinear form

$$\mathcal{A}(u, v) - \lfloor (u, v) = (v, f)$$

where

$$\begin{aligned}\mathcal{A}(u,v) &= \int_\Omega (\nabla v \cdot \nabla u) dA, \\ \lfloor(u,v) &= \int_{\partial\Omega} (v \nabla u \cdot \mathbf{n}) dS \\ (v,f) &= 0.\end{aligned}$$

4 Integration Over Reference Element

Let us consider a simple element to start. Let there be an element $\hat{\Omega} : [0,1] \times [0,1]$ where $x' \in [0,1]$ and $y' \in [0,1]$. Furthermore, let there be a function $u(x', y')$ that we would like to approximate on $\hat{\Omega}$. We do this by approximating

$$\begin{aligned}u(x',y') \approx U(x',y') &= \sum_i c_i \varphi_i(x',y'), \\ &= c_1\varphi_1(x',y') + c_2(x',y')\varphi_2(x',y') + c_3\varphi_3(x',y') + c_4\varphi_4(x',y') + \ldots.\end{aligned}$$

We may use an arbitrarily large number of functions $c_i\varphi_i(x', y')$ to approximate $u(x', y')$, but we will restrict our discussion to four linear functions. From the above approximation, we can see that the c_i functions will approximate the value of $u(x', y')$ at a given point (x', y') and the $\varphi(x', y')$ functions will serve as a way to interpolate these coefficients inside of $\hat{\Omega}$. These functions φ_i are called shape functions. We define the linear 2-dimensional shape functions $\varphi_i = \varphi_i(x', y')$ on $\hat{\Omega}$ are given as

$$\begin{aligned}\hat{\varphi}_1 &= \frac{1}{2}(1-x')(1-y'), \\ \hat{\varphi}_2 &= \frac{1}{2}(x')(1-y'), \\ \hat{\varphi}_3 &= \frac{1}{2}(x')(y'), \\ \hat{\varphi}_4 &= \frac{1}{2}(1-x')(y').\end{aligned}$$

It is important to note that we define c_i at each vertex (i.e. c_1 is located at $(0,0)$; c_2 at $(1,0)$; c_3 at $(1,1)$; c_4 at $(0,1)$) and therefore each $\varphi_i(x'_i, y'_i) = 1$ at node i and $\varphi_i(x' \neq x'_i, y' \neq y'_i) = 0$.

Applying this approximation to the integral we derived above on $\hat{\Omega}$, we can write $\hat{\lfloor}(u,v)$ and (f,v) in terms of our approximations

$$\begin{aligned}\hat{\mathcal{A}}(v,u) &= \int_{\hat{\Omega}} (\nabla v \cdot \nabla u) dA, \\ &\approx \int_{\hat{\Omega}} \nabla v \cdot \nabla \Big(\sum_i c_i \varphi_i \Big) dA.\end{aligned}$$

Quickly, the problem of what to do with the test function v arises—what is it? How does it help? One method of solving this problem is to define $v(x', y')$ in the same form as u, using the same

shape functions φ_i, namely

$$\begin{aligned} v(x',y') &= \sum_i d_i\hat{\varphi}_i(x',y'), \\ &= d_1\hat{\varphi}_1(x',y') + d_2(x',y')\hat{\varphi}_2(x',y') + d_3\hat{\varphi}_3(x',y') + d_4\hat{\varphi}_4(x',y') + \ldots. \end{aligned}$$

Applying this form of $v(x',y')$ to the $\hat{\mathcal{A}}(u,v)$ gives

$$\begin{aligned} \hat{\mathcal{A}}(v,u) &\approx \int_{\hat{\Omega}} \nabla\Bigg(\begin{bmatrix} d_1 & d_2 & d_3 & d_4 \end{bmatrix} \begin{bmatrix} \hat{\varphi}_1 \\ \hat{\varphi}_2 \\ \hat{\varphi}_3 \\ \hat{\varphi}_4 \end{bmatrix} \Bigg) \cdot \nabla\Bigg(\begin{bmatrix} c_1 & c_2 & c_3 & c_4 \end{bmatrix} \begin{bmatrix} \hat{\varphi}_1 \\ \hat{\varphi}_2 \\ \hat{\varphi}_3 \\ \hat{\varphi}_4 \end{bmatrix} \Bigg) dA, \\ &= \int_{\hat{\Omega}} \Bigg(\begin{bmatrix} d_1 & d_2 & d_3 & d_4 \end{bmatrix} \begin{bmatrix} \nabla\hat{\varphi}_1 \\ \nabla\hat{\varphi}_2 \\ \nabla\hat{\varphi}_3 \\ \nabla\hat{\varphi}_4 \end{bmatrix} \Bigg) \cdot \Bigg(\begin{bmatrix} c_1 & c_2 & c_3 & c_4 \end{bmatrix} \begin{bmatrix} \nabla\hat{\varphi}_1 \\ \nabla\hat{\varphi}_2 \\ \nabla\hat{\varphi}_3 \\ \nabla\hat{\varphi}_4 \end{bmatrix} \Bigg) dA, \\ &= \int_{\hat{\Omega}} \Bigg(\begin{bmatrix} d_1 & d_2 & d_3 & d_4 \end{bmatrix} \begin{bmatrix} \nabla\hat{\varphi}_1 \\ \nabla\hat{\varphi}_2 \\ \nabla\hat{\varphi}_3 \\ \nabla\hat{\varphi}_4 \end{bmatrix} \Bigg) \cdot \Bigg(\begin{bmatrix} \nabla\hat{\varphi}_1 & \nabla\hat{\varphi}_2 & d\nabla\hat{\varphi}_3 & \nabla\hat{\varphi}_4 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \Bigg) dA, \\ &= \int_{\hat{\Omega}} \begin{bmatrix} d_1 & d_2 & d_3 & d_4 \end{bmatrix} \begin{bmatrix} \nabla\hat{\varphi}_1\cdot\nabla\hat{\varphi}_1 & \nabla\hat{\varphi}_1\cdot\nabla\hat{\varphi}_2 & \nabla\hat{\varphi}_1\cdot\nabla\hat{\varphi}_3 & \nabla\hat{\varphi}_1\cdot\nabla\hat{\varphi}_4 \\ \nabla\hat{\varphi}_2\cdot\nabla\hat{\varphi}_1 & \nabla\hat{\varphi}_2\cdot\nabla\hat{\varphi}_2 & \nabla\hat{\varphi}_2\cdot\nabla\hat{\varphi}_3 & \nabla\hat{\varphi}_2\cdot\nabla\hat{\varphi}_4 \\ \nabla\hat{\varphi}_3\cdot\nabla\hat{\varphi}_1 & \nabla\hat{\varphi}_3\cdot\nabla\hat{\varphi}_2 & \nabla\hat{\varphi}_3\cdot\nabla\hat{\varphi}_3 & \nabla\hat{\varphi}_3\cdot\nabla\hat{\varphi}_4 \\ \nabla\hat{\varphi}_4\cdot\nabla\hat{\varphi}_1 & \nabla\hat{\varphi}_4\cdot\nabla\hat{\varphi}_2 & \nabla\hat{\varphi}_4\cdot\nabla\hat{\varphi}_3 & \nabla\hat{\varphi}_4\cdot\nabla\hat{\varphi}_4 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} dA \\ &= \begin{bmatrix} d_1 & d_2 & d_3 & d_4 \end{bmatrix} \Bigg(\int_{\hat{\Omega}} \begin{bmatrix} \nabla\varphi_1\cdot\nabla\hat{\varphi}_1 & \nabla\hat{\varphi}_1\cdot\nabla\hat{\varphi}_2 & \nabla\hat{\varphi}_1\cdot\nabla\hat{\varphi}_3 & \nabla\hat{\varphi}_1\cdot\nabla\hat{\varphi}_4 \\ \nabla\hat{\varphi}_2\cdot\nabla\hat{\varphi}_1 & \nabla\hat{\varphi}_2\cdot\nabla\hat{\varphi}_2 & \nabla\hat{\varphi}_2\cdot\nabla\hat{\varphi}_3 & \nabla\hat{\varphi}_2\cdot\nabla\hat{\varphi}_4 \\ \nabla\hat{\varphi}_3\cdot\nabla\hat{\varphi}_1 & \nabla\hat{\varphi}_3\cdot\nabla\hat{\varphi}_2 & \nabla\hat{\varphi}_3\cdot\nabla\hat{\varphi}_3 & \nabla\hat{\varphi}_3\cdot\nabla\hat{\varphi}_4 \\ \nabla\hat{\varphi}_4\cdot\nabla\hat{\varphi}_1 & \nabla\hat{\varphi}_4\cdot\nabla\hat{\varphi}_2 & \nabla\hat{\varphi}_4\cdot\nabla\hat{\varphi}_3 & \nabla\hat{\varphi}_4\cdot\nabla\hat{\varphi}_4 \end{bmatrix} dA \Bigg) \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix} \end{aligned}$$

This expression looks large and in-charge, but it actually simplifies down quite nicely for the linear case using rectangular elements because each of the expressions in the gradient matrix are integrable by the way we have selected the shape functions. Given that

$$\begin{aligned} \nabla\hat{\varphi}_1 &= \frac{1}{2}\Big(-(1-y')\mathbf{x}' - (1-x')\mathbf{y}' \Big), \\ \nabla\hat{\varphi}_2 &= \frac{1}{2}\Big((1-y')\mathbf{x}' + -x'\mathbf{y}' \Big), \\ \nabla\hat{\varphi}_3 &= \frac{1}{2}\Big(y'\mathbf{x}' + x'\mathbf{y}' \Big), \\ \nabla\hat{\varphi}_4 &= \frac{1}{2}\Big(-y'\mathbf{x}' + (1-x)\mathbf{y}' \Big), \end{aligned}$$

and $dA = dxdy$, we can evaluate $\mathcal{A}(u,v)$ to

$$\begin{aligned}
\mathcal{A}(u,v) &= \begin{bmatrix} d_1 & d_2 & d_3 & d_4 \end{bmatrix} \left(\int_{\hat{\Omega}} \begin{bmatrix} \nabla\varphi_1 \cdot \nabla\varphi_1 & \nabla\varphi_1 \cdot \nabla\varphi_2 & \nabla\varphi_1 \cdot \nabla\varphi_3 & \nabla\varphi_1 \cdot \nabla\varphi_4 \\ \nabla\varphi_2 \cdot \nabla\varphi_1 & \nabla\varphi_2 \cdot \nabla\varphi_2 & \nabla\varphi_2 \cdot \nabla\varphi_3 & \nabla\varphi_2 \cdot \nabla\varphi_4 \\ \nabla\varphi_3 \cdot \nabla\varphi_1 & \nabla\varphi_3 \cdot \nabla\varphi_2 & \nabla\varphi_3 \cdot \nabla\varphi_3 & \nabla\varphi_3 \cdot \nabla\varphi_4 \\ \nabla\varphi_4 \cdot \nabla\varphi_1 & \nabla\varphi_4 \cdot \nabla\varphi_2 & \nabla\varphi_4 \cdot \nabla\varphi_3 & \nabla\varphi_4 \cdot \nabla\varphi_4 \end{bmatrix} dxdy \right) \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}, \\
&= \begin{bmatrix} d_1 & d_2 & d_3 & d_4 \end{bmatrix} \left(\frac{1}{2} \begin{bmatrix} 1/3 & -1/12 & -1/6 & -1/12 \\ -1/12 & 1/3 & -1/12 & -1/6 \\ -1/6 & -1/12 & 1/3 & -1/12 \\ -1/12 & -1/6 & -1/12 & 1/3 \end{bmatrix} \right) \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}.
\end{aligned}$$

As shown above, the integral simplifies nicely down to a matrix expression populated with constants.

5 Map to Reference Element to Global Element

Knowing the process of integration on the reference element $\hat{\Omega} := [0,1] \times [0,1]$, we are now able to perform integrations on this element relatively easily using a computer. However, these integrals are only useful on this element and do not provide much value elsewhere. One method of solving this issue is through a process called affine mapping. Affine mapping involves transforming a set of coordinates $(x,y) \to (x',y')$ in a manner that preserves a 1-to-1 correspondence between the points in each coordinate system. If we can relate the coordinates in (x,y) to the coordinates in (x',y'), we could perform the integration over the reference element, rather than on the element in its position within the global mesh. In order to generate this mapping function for our rectangular global elements to the square reference element, we define a function $F(x,y)$ such that

$$\begin{aligned}
F\begin{pmatrix} x \\ y \end{pmatrix} &= \sum_i \begin{pmatrix} x_i \\ y_i \end{pmatrix} \varphi_i(x',y'), \\
&= \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \hat{\varphi}_1(x',y') + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \hat{\varphi}_2(x',y') + \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} \varphi_3(x',y') + \begin{pmatrix} x_4 \\ y_4 \end{pmatrix} \varphi_4(x',y'), \\
&= \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \frac{1}{2}(1-x')(1-y') + \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} \frac{1}{2}(x')(1-y') + \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} \frac{1}{2}(x')(y') + \begin{pmatrix} x_4 \\ y_4 \end{pmatrix} \frac{1}{2}(1-x')(y').
\end{aligned}$$

where $\varphi_i(x',y')$ are the linear shape functions on the reference element defined previously. We use these shape functions because they ensure that every point on $\hat{\Omega}$ and that the vertices on $\hat{\Omega}$ correspond to their correct locations in the global mesh. We can simplify this expression to

$$F^{-1}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} + B\begin{pmatrix} x' \\ y' \end{pmatrix},$$

where

$$B = \begin{bmatrix} x_2 - x_1 & x_4 - x_1 \\ y_2 - y_1 & y_4 - y_1 \end{bmatrix}.$$

Similarly, the inverse mapping from $(x',y') \to (x,y)$ can be written as

$$F\begin{pmatrix} x' \\ y' \end{pmatrix} = B^{-1}\left[\begin{pmatrix} x \\ y \end{pmatrix} - \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \right].$$

Now that we have the mapping defined between each global element and the reference element, we can reexamine the shape functions on the canonical element to see how they have changed due to the mapping. Let's begin with looking at the shape functions on a global element $\varphi_i(x,y)$. The goal would be to use the same manipulations we used on the canonical element on the global shape function. However, the mapping affects the gradient, so that must be considered during the integration step. The chain rule for differentiation allows us to write

$$\begin{aligned}
\nabla\varphi_i(x,y) &= \frac{\partial\varphi_i(x,y)}{\partial x} + \frac{\partial\varphi_i(x,y)}{\partial y}, \\
&= \frac{\partial\varphi_i(x',y')}{\partial x}\frac{\partial x}{\partial x'} + \frac{\partial\varphi_i(x',y')}{\partial x}\frac{\partial x}{\partial y'} + \frac{\partial\varphi_i(x',y')}{\partial y}\frac{\partial y}{\partial y'} + \frac{\partial\varphi_i(x',y')}{\partial y}\frac{\partial y}{\partial x'}, \\
&= \begin{bmatrix} \dfrac{\partial x'}{\partial x} & \dfrac{\partial x'}{\partial y} \\ \dfrac{\partial y'}{\partial x} & \dfrac{\partial y'}{\partial y} \end{bmatrix} \nabla\hat{\varphi}_i(x',y'), \\
&= B^{-1}\nabla\hat{\varphi}_i(x',y')
\end{aligned}$$

where B^{-1} is the Jacobean of the mapping function from $(x,y) \to (x',y')$ we defined previously. Similarly to how $\hat{\mathcal{A}}(u,v)$ was generated on the reference element, we now generate $a_k(u,v)$ on a global element k using linear elements and linear shape functions

$$\begin{aligned}
\mathcal{A}_k(v,u) &= \int_{\Omega_k} (\nabla v \cdot \nabla u) dA, \\
&\approx \int_{\Omega_k} \nabla\Big(\sum_i d_i\varphi_i\Big) \cdot \nabla\Big(\sum_i c_i\varphi_i\Big) dA, \\
&= \int_{\hat{\Omega}} B^{-1}\nabla\Big(\sum_i d_i\varphi_i\Big) \cdot B^{-1}\nabla\Big(\sum_i c_i\varphi_i\Big) |\det(B)| dxdy \\
&= \int_{\hat{\Omega}} \nabla\Big(\sum_i d_i\varphi_i\Big)^T B^{-T}B^{-1}\nabla\Big(\sum_i c_i\varphi_i\Big) |\det(B)| dxdy \\
&= \begin{bmatrix} d_1 & d_2 & d_3 & d_4 \end{bmatrix} \left(\frac{1}{2(x_2-x_1)(y_4-y_1)} \begin{bmatrix} 1/3 & -1/12 & -1/6 & -1/12 \\ -1/12 & 1/3 & -1/12 & -1/6 \\ -1/6 & -1/12 & 1/3 & -1/12 \\ -1/12 & -1/6 & -1/12 & 1/3 \end{bmatrix} \right) \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix},
\end{aligned}$$

where $\det(B) = (x_2-x_1)(y_4-y_1) - (x_4-x_1)(y_4-y_1) = (x_2-x_1)(y_4-y_1)$ for rectangular elements and $dA = |\det(B)| dxdy$.

6 Neumann Boundary Conditions in the Galerkin Projection

Now we address the Neumann boundary conditions derived using the weak form. Specifically, we look at $b_k(u,v)$ on each element Ω_k, which we previously defined as

$$b_k(u,v) = \int_{\partial\Omega_k} (v\nabla u \cdot \mathbf{n}) dS.$$

The boundaries (edges) $\partial\Omega_k$ of a given element Ω_k may be classified as either external, meaning they are on the boundary of the global domain ($\partial\Omega_k \in \partial\Omega$) or they are internal (all others). Consider two adjacent elements Ω_k and Ω_{k+1} that share an edge $\partial\Omega_k$. Nothing is preventing a flux through that edge $\nabla u \cdot \mathbf{n}_k\big|_{\partial\Omega_k|\Omega_{k+1}}$ from Ω_k to Ω_{k+1}. In fact, this flux is *probably* non-zero provided, $u(x,y)$ is changing in the vicinity. This issue can be avoided by enforcing a flux continuity condition across the $\partial\Omega_k|\Omega_{k+1}$ boundary

$$\nabla u \cdot \mathbf{n}_k + \nabla u \cdot \mathbf{n}_{k+1}\Big|_{\partial\Omega_k|\Omega_{k+1}} = 0.$$

As we assemble the bilinear form $\mathcal{A}(u,v)+\lfloor(u,v)$ of the algebraic system, the flux out of one element into adjacent elements is cancelled by the flux of those adjacent element into the first. Therefore, by dropping this term entirely, the flux between elements cancels. The remaining edges—those elemental edges on the external boundary of the global mesh—must then be prescribed Neumann boundary conditions from the global boundary $\partial\Omega$. For this particular problem, there are no *Neumann* boundary conditions indicating $v = 0$ on these boundaries, and therefore the $\lfloor(u,v)$ term disappears. Furthermore, we now can now remove the d_i matrix from $\mathcal{A}(u,v)$, resulting in

$$\begin{aligned} a_k(v,u) &= \frac{1}{2(x_2-x_1)(y_4-y_1)} \begin{bmatrix} 1/3 & -1/12 & -1/6 & -1/12 \\ -1/12 & 1/3 & -1/12 & -1/6 \\ -1/6 & -1/12 & 1/3 & -1/12 \\ -1/12 & -1/6 & -1/12 & 1/3 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}, \\ &= K_k \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ c_4 \end{bmatrix}, \end{aligned}$$

where

$$K_k = \frac{1}{2(x_2-x_1)(y_4-y_1)} \begin{bmatrix} 1/3 & -1/12 & -1/6 & -1/12 \\ -1/12 & 1/3 & -1/12 & -1/6 \\ -1/6 & -1/12 & 1/3 & -1/12 \\ -1/12 & -1/6 & -1/12 & 1/3 \end{bmatrix}.$$

7 Assembly of Stiffness Matrix

Now, we loop over every element and begin to populate the larger matrix K, called the stiffness matrix, composed of all the local K_k matrices. Upon doing so, we have formed the algebraic system

$$Kc = f.$$

7.1 Code: Assembly of Stiffness Matrix

```
%% Assemble Stiffness Matrix
K = zeros((M+1)*(N+1),(M+1)*(N+1));
kk = [1/3, -1/12, -1/6, -1/12;
      -1/12, 1/3, -1/12, -1/6;
      -1/6, -1/12, 1/3, -1/12;
      -1/12, -1/6, -1/12, 1/3];
for i = 1:length(elements)
    nodes = elements(i,:);
    hx = verts(nodes(2),1) - verts(nodes(1),1);
    hy = verts(nodes(4),2) - verts(nodes(1),2);
    for j = 1:length(nodes)
        for k = 1:length(nodes)
            K(nodes(j), nodes(k)) = K(nodes(j), nodes(k)) + 1/(2*hx*
                hy)*kk(j,k);
        end
    end

end
```

8 Dirichlet Boundary Conditions

Next, we loop over the elements on the boundary of Ω that are prescribed a Dirichlet boundary condition. Because the value of vertex c_i is given, we must adjust the system (K and f) to account for these known values.

8.1 Code: Applying Dirichlet Boundary Conditions

```
%% Solve for Dirichlet Boundary Conditions
fxn1 = @(y) cos(y);
fxn2 = @(y) sin(y);
f = zeros((N+1)*(M+1), 1);
for i = 1:length(verts)
    if verts(i,1) == xmax
        K(i,:) = 0;
        K(i,i) = 1;
        f(i) = fxn1(verts(i,2));
    elseif verts(i,1) == xmin
        K(i,:) = 0;
        K(i,i) = 1;
        f(i) = fxn2(verts(i,2));
    end
end
```

9 Solving the System

Finally, we solve the system $Kc = F$. Though direct inversion is possible for many smaller systems, it is a slow or impossible approach for larger systems. Several methods are available to solve these equations, but at present, we will use a built-in function denoted by the "\" function (`mldivide()`). Finally, it is important to remember that the final solution is

$$u(x, y) \approx \sum_i c_i \varphi_i(x, y)$$

and is defined as such on each individual element. ***The shape functions must be plotted with the vertex values in order to obtain a valid solution!***.

9.1 Code: Solving the Linear System

```
%% Solve Linear System
c = K\f;

%% Plot Solution
% Reformat solution for plotting
sol = zeros(N+1, M+1);
for i = 1:M+1
    for j = 1:N+1
        sol(j,i) = c(j+(i-1)*(N+1));
    end
end
% Create contour plot
contourf(xvals, yvals, sol)
```

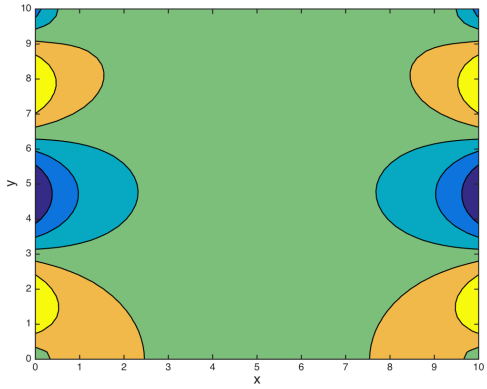

Figure 4: Solution using the code and algorithms herein written.A 50×50 rectangular mesh was generated with linear shape functions and plotted using the `conourf()` function.

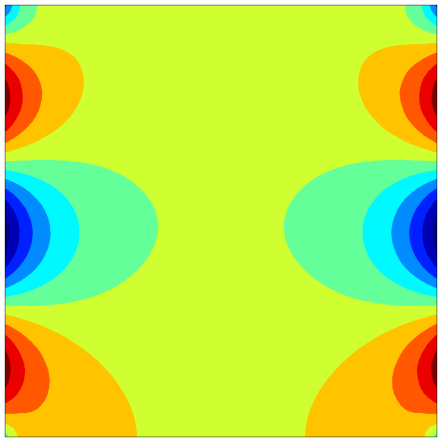

Figure 5: Solution generated and plotted using the commercial finite element software COMSOL (v5.0).

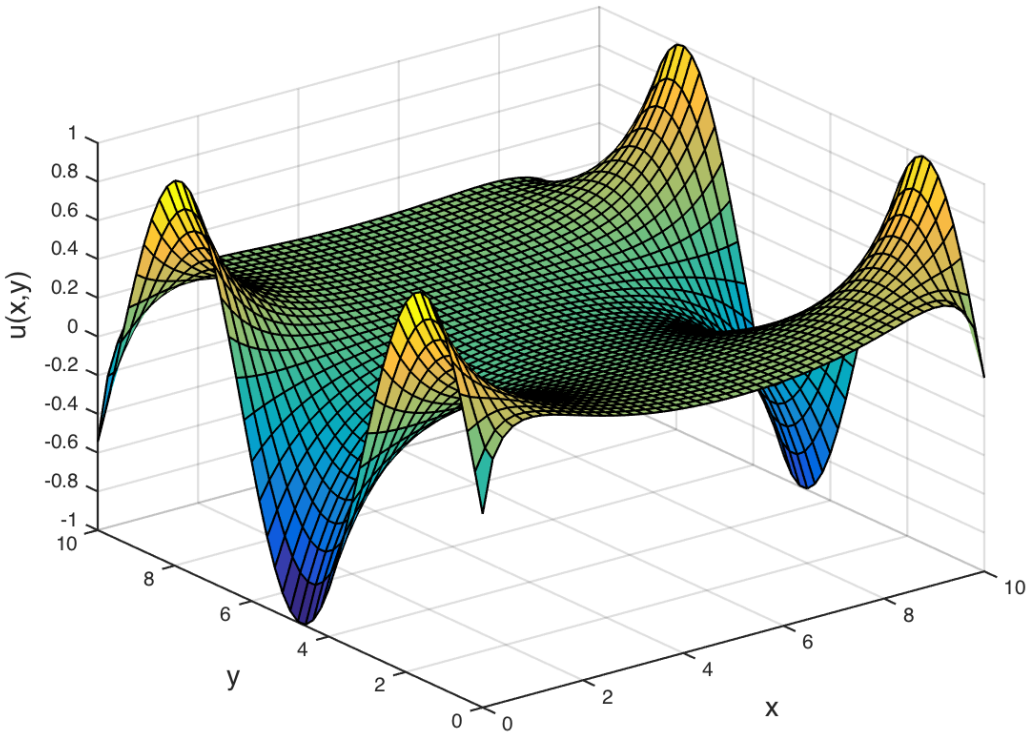

Figure 6: Solution using the code and algorithms herein written. A 50×50 rectangular mesh was generated with linear shape functions and plotted using the `surf()` function.

10 Full Code Appendix

```
%% 2D Diffusion on a Rectangular Plane (Dirichlet Boundaries)
clear; clc;
xmin = 0; xmax = 10;
ymin = 0; ymax = 10;
N = 50;
M = 50;
xvals = linspace(xmin, xmax, N+1);
yvals = linspace(ymin, ymax, M+1);

% Initialize vert_list because it's good practice
verts = zeros((N+1)*(M+1),2);

```

```
for i=1:M+1
    for j = 1:N+1
        % How do we create a list of vertex points?
        verts(j+(N+1)*(i-1),:) = [xvals(j), yvals(i)];
    end
end

%Initialize elements matrix for good practice
elements = zeros(N*M, 4);
fig = figure(1);
hold on
for i = 1:N
    for j = 1:M

        % global vertex label for local vertex 1
        v1 = i+(N+1)*(j-1);

        % global vertex label for local vertex 2
        v2 = i+1+(N+1)*(j-1);

        % global vertex label for local vertex 3
        v3 = i+1+(N+1)*j;

        % global vertex label for local vertex 4
        v4 = i+(N+1)*j;

        % Add global element labels into a list of elements
        elements(i+(j-1)*N,:) = [v1, v2, v3, v4];

                % Find the global labels of each of the nodes for a
                    given element
        nodes = elements(i+(j-1)*N,:);

        % Find the coordinates of the bounding rectangle of the
           element
        outline = [verts(nodes(1),:);
                   verts(nodes(2),:);
                   verts(nodes(3),:);
                   verts(nodes(4),:);
                   verts(nodes(1),:)];

        % How can we visualize what exactly we are plotting?
        plot(outline(:,1), outline(:,2), '-k')
    end
end
xlim([xmin-0.1*(xmax-xmin), 0.1*(xmax-xmin)+xmax]);
```

```
ylim([ymin-0.1*(ymax-ymin), 0.1*(ymax-ymin)+ymax]);
xlabel('x', 'FontSize', 14)
ylabel('y', 'FontSize', 14)
saveas(fig, 'workedProblem_meshHomemade.png')

%% Assemble Stiffness Matrix
K = zeros((M+1)*(N+1),(M+1)*(N+1));
kk = [1/3, -1/12, -1/6, -1/12;
      -1/12, 1/3, -1/12, -1/6;
      -1/6, -1/12, 1/3, -1/12;
      -1/12, -1/6, -1/12, 1/3];
for i = 1:length(elements)
    nodes = elements(i,:);
    hx = verts(nodes(2),1) - verts(nodes(1),1);
    hy = verts(nodes(4),2) - verts(nodes(1),2);
    for j = 1:length(nodes)
        for k = 1:length(nodes)
            K(nodes(j), nodes(k)) = K(nodes(j), nodes(k)) + 1/(2*hx*
                hy)*kk(j,k);
        end
    end

end

%% Solve for Dirichlet Boundary Conditions
fxn1 = @(y) sin(y);
fxn2 = @(y) sin(y);
f = zeros((N+1)*(M+1), 1);
for i = 1:length(verts)
    if verts(i,1) == xmax
        K(i,:) = 0;
        K(i,i) = 1;
        f(i) = fxn1(verts(i,2));
    elseif verts(i,1) == xmin
        K(i,:) = 0;
        K(i,i) = 1;
        f(i) = fxn2(verts(i,2));
    end
end

%% Solve Linear System
c = K\f;

%% Plot Solution
sol = zeros(N+1, M+1);
for i = 1:M+1
```

```
    for j = 1:N+1
        sol(i,j) = c(j+(i-1)*(N+1));
    end
end

fig = figure(2);
contourf(xvals, yvals, sol,5)
xlabel('x', 'FontSize', 14)
ylabel('y', 'FontSize', 14)
saveas(fig, 'workedProblem_figure2.png')

fig = figure(3);
surf(xvals, yvals, sol)
xlabel('x', 'FontSize', 14)
ylabel('y', 'FontSize', 14)
zlabel('u(x,y)', 'FontSize', 14)
saveas(fig, 'workedProblem_figure3.png')
```