# WEAK GALERKIN FINITE ELEMENT SOLVER WITH POLYGONAL MESHES

Ansh Bhatt

Naman Goyal

Supervisor: Dr Bhupen Deka

Indian Institute of Technology, Guwahati

November 11, 2021

# Outline

# Outline

# Objective

- To understand WGFEM
- To implement 3D mesh generation and FEM solver
- To solve a model problem and look at the results

# Outline

# Abstract


- In mathematics and various associated fields Finite Element Method is used to numerically approximate the solutions to partial differential equations which are not possible to solve analytically. In this project we try to develop an efficient solver for Finite Element approximations using polygonal meshes. We have already looked at FEM and a few of its computational techniques. We shall now look at WGFEM and proceed to develop the FEM solver.

# Outline

# Introduction

- Classical FEM has limitation, such that it is restricted to piece wise polynomials with prescribed continuity.
- Use of discontinuous approximation functions in our models, WGFEM is required in constructing finite element space.
- Recall following weak formulation to the BVP: Find $u \in H_0^1(\Omega)$ such that $\int_\Omega \nabla u \cdot \nabla v dx = \int_\Omega f v dx \ \ \forall v \in H_0^1(\Omega)$
- Weak Galerkin formulation can be obtained from the above by replacing the classical differential operators by their weak counterparts.

# Weak Functions

- Let $K$ be any polygonal or polyhedral domain with interior $K^0$ and boundary $\partial K$.
- A weak function on the region $K$ refers to a pair of scalar-valued functions $v = \{v_0, v_b\}$
- For any weak function $v = \{v_0, v_b\}$, its weak gradient $\nabla_w v$ is defined as a linear functional on $H$ whose action on each $q \in H$ is given by $(\nabla_w v, q)_K = -\int_K v_0 \nabla \cdot q dK + \int_{\partial K} v_b q \cdot \mathbf{n} ds$, where $\mathbf{n}$ is the outward normal to $\partial K$.

# Weak Galerkin Space $\left(\mathcal{P}_k(K),\ \mathcal{P}_j(\partial K),\ \left[\mathcal{P}_l(K)\right]^2\right)$

- $k \geq 1$ is the degree of polynomials in the interior of the element $K$
- $j \geq 0$ is the degree of polynomials on the boundary of $K$ and
- $l \geq 0$ is the degree of polynomials employed in the computation of weak gradients or weak first order partial derivatives.
- $k,\ j,\ l$ are selected in such a way that minimizes the number of unknowns in the numerical scheme without compromising the accuracy of the numerical approximation.
- A lowest order WG-FEM space is $\left(\mathcal{P}_1(K),\ \mathcal{P}_0(\partial K),\ \left[\mathcal{P}_0(K)\right]^2\right)$.

# Weak Galerkin Algorithm

- We define a bilinear map $a(\cdot,\cdot) : V_h \times V_h \to \mathbb{R}$ by
  $a(u_h, v_h) = (\nabla_w u_h, \nabla_w v_h) = \sum_{K \in \mathcal{T}_h} (\nabla_w u_h, \nabla_w v_h)_K$
- with a stabilizer $s(\cdot,\cdot) : V_h \times V_h \to \mathbb{R}$ defined by
  $s(u_h, v_h) = \sum_{K \in \mathcal{T}_h} h_K^{-1} \langle u_0 - u_b, v_0 - v_b \rangle_{\partial K}$
- Find $u_h = \{u_0, u_b\} \in V_h^0$ such that: $a_s(u_h, v_h) = (f, v_h) \ \ \forall v_h \in V_h^0$
  with $a_s(u_h, v_h) = a(u_h, v_h) + s(u_h, v_h)$

# $L^2$ Projections

- For each element $K \in \mathcal{T}_h$, denote by $Q_0$ the usual $L^2$ projection operator from $L^2(K)$ onto $\mathcal{P}_k(K)$ and by $Q_b$ the $L^2$ projection from $L^2(e)$ onto $\mathcal{P}_{k-1}(e)$ for any $e \in \mathcal{E}_h$.
- We shall combine $Q_0$ with $Q_b$ by writing $Q_h = \{Q_0, Q_b\}$. More precisely, for $\phi \in H^1(K)$, we have $Q_h\phi = \{Q_0\phi, Q_b\phi\}$.
- In addition to $Q_h$, let $\mathbb{Q}_h$ be an another local $L^2$ projection from $[L^2(K)]^2$ onto $[\mathcal{P}_{k-1}(K)]^2$.

# Error Equation I

- We split our error into two components using an intermediate operator. We write

$$u - u_h = (u - Q_h u) + (Q_h u - u_h).$$

- For simplicity, we introduce the following notation

$$e_h := \{e_0, e_b\} = u_h - Q_h u$$

- Testing original equation by using $v_0$ of $v = \{v_0, v_b\} \in V_h^0$, we arrive at

$$a_s(Q_h u, v) = (f, v_0) + s(Q_h u, v) - \langle v_0 - v_b, (\mathbb{Q}_h(\nabla u) - \nabla u) \cdot \mathbf{n} \rangle_{\partial K}$$

# Error Equation II

- Finally, subtracting WG approximation, we obtain

$$a_s(e_h, v) = l_1(u, v) + l_2(u, v) \quad \forall v \in V_h^0$$

where bilinear forms $l_1(\cdot, \cdot)$ and $l_2(\cdot, \cdot)$ are given by

$$l_1(u, v) = \sum_{K \in \mathcal{T}_h} \langle (\nabla u - \mathbb{Q}_h(\nabla u)) \cdot \mathbf{n}, v_0 - v_b \rangle_{\partial K}$$

$$l_2(u, v) = \sum_{K \in \mathcal{T}_h} h_K^{-1} \langle Q_0 u - Q_b(u|_{\partial K}), v_0 - v_b \rangle_{\partial K}.$$

# Convergence Results

- **Convergence Results for $H^1$-norm:** Let $u_h \in V_h$ be the weak Galerkin finite element solution of the problem. Assume that the exact solution is so regular that $u \in H^{k+1}(\Omega)$. Then, there exists a constant $C$ such that

$$\|e_h\| \leq Ch^k \|u\|_{k+1,\Omega}$$

- **Convergence Results for $L^2$-norm:** Let $u_h \in V_h$ be the weak Galerkin finite element solution of the problem. Assume that the exact solution is so regular that $u \in H^{k+1}(\Omega)$. Then, there exists a constant $C$ such that

$$\|e_0\| \leq Ch^{k+1} \|u\|_{k+1,\Omega}$$

# Outline

# Rectangular Domain I

When considering mesh generation on a two dimensional domain, the simplest form is triangulation of a rectangular domain. The following algorithm generates triangular elements from a given rectangular domain with a fixed step size in both directions. Figure 1 shows a rectangular domain defined by $[-1, 1] \times [-1, 1]$. It has been triangulated with a step size of 1 in both directions. Each node and element has been numbered.

# Rectangular Domain II

**Algorithm 1** Triangulation of Rectangular Domain

**Require:** $[x_0, x_1, y_0, y_1]$ denoting the domain and *step size* denoting the step size

**Ensure:** node contains array of all nodes. elem contains list of all elements.

$square \leftarrow [x_0, x_1, y_0, y_1]$

$h \leftarrow step\ size$

▷ Creating Array of Nodes

$x_0 \leftarrow square(1), x_1 \leftarrow square(2), y_0 \leftarrow square(3), y_1 \leftarrow square(4)$

$h_x \leftarrow h(1), h_y \leftarrow h(2)$

Generate meshgrid from $[x_0 : h_x : x_1] \times [y_0 : h_y : y_1]$

node $\leftarrow$ array of vertices from meshgrid

▷ Listing Elements

$ni \leftarrow$ number of rows in mesh

$k \leftarrow 1$

**while** $k \leq$ Number of nodes $- ni$ **do**

$elem \leftarrow [elem, (k, k + ni, k + ni + 1)]$

$elem \leftarrow [elem, (k, k + 1, k + ni + 1)]$

$k \leftarrow k + 1$

**end while**

# Rectangular Domain III

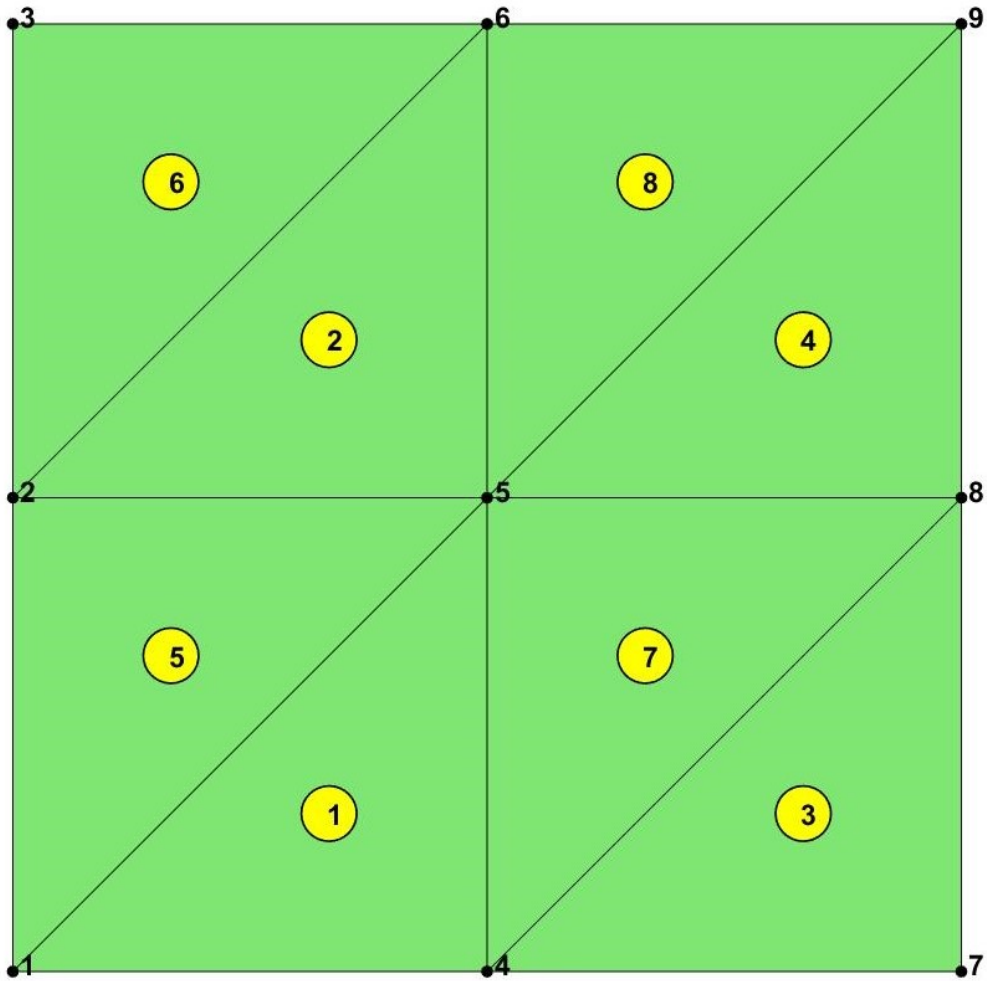


Figure 1: Triangulation of Rectangular Domain

# Rectangular Domain IV

- Once the mesh has been generated, we follow it up by extracting all the necessary information about the elements and their positioning.
- We will obviously need to recognize the edges and what element they are a part of.
- Since, for solving any PDE, we will have boundary conditions which will have to be satisfied, therefore we will need to mark all the edges and elements which lie on the boundary of our mesh.
- Further, we will also require information about neighboring elements.
- All of this information is stored in a structure T.

# Rectangular Domain V

```
function T = auxstructure(elem)
%  AUXSTRUCTURE auxiliary structure for a 2-D triangulation.
%  T = AUXSTRUCTURE(elem) constucts the indices map between elements, edges
%  and nodes, and the boundary information. T is a structure.

    totalEdge = uint32(sort([elem(:,[2,3]); elem(:,[3,1]); elem(:,[1,2])], 2));
    [edge,i_tot_last,i_edge] = myunique(totalEdge);
    NT = size(elem, 1);
    elem2edge = uint32(reshape(i_edge, NT, 3));
    i_tot_first(i_edge(3*NT:-1:1)) = 3*NT:-1:1;
    i_tot_first = i_tot_first';
    first_elem_edge = ceil(i_tot_first/NT);
    last_elem_edge = ceil(i_tot_last/NT);
    first_elem = i_tot_first - NT*(first_elem_edge-1);
    last_elem = i_tot_last - NT*(last_elem_edge-1);
    diff_ind = (i_tot_first ~= i_tot_last);
    neighbor = uint32(accumarray([[first_elem(diff_ind) first_elem_edge(diff_ind)];...
        [last_elem last_elem_edge]], [last_elem(diff_ind); first_elem], [NT 3]));
    edge2elem = uint32([first_elem last_elem first_elem_edge last_elem_edge]);
    bdElem = first_elem(first_elem == last_elem);
    first_edge_bd = first_elem_edge(first_elem == last_elem);
    bdEdge = uint32([elem(bdElem(first_edge_bd==1),[2 3]);...
        elem(bdElem(first_edge_bd==2),[3 1]); elem(bdElem(first_edge_bd==3),[1 2])]);
    bdEdge2elem = uint32([bdElem(first_edge_bd==1);bdElem(first_edge_bd==2);...
        bdElem(first_edge_bd==3)]);
    T = struct('neighbor', neighbor, 'edge2elem', edge2elem, 'edge', edge,...
        'elem2edge', elem2edge, 'bdElem', bdElem, 'bdEdge', bdEdge,...
        'bdEdge2elem', bdEdge2elem);
end
```

# Cubical Domain I

- Similar to the two dimensional domain, we have taken into consideration a simple form of mesh generation on a cubical domain.
- In place of triangles, we consider each element to be a tetrahedron.
- The following code generates tetrahedrons of fixed step size in all three directions from a given cuboid in the three dimensional space.
- Figure 2 shows a cubical domain defined by $[0, 1] \times [0, 1] \times [0, 1]$, which has been divided into six tetrahedrons with step size 1 in each direction.

# Cubical Domain II

```
function [node, elem, HB] = cubemesh(box,h)
    x0 = box(1); x1 = box(2);
    y0 = box(3); y1 = box(4);
    z0 = box(5); z1 = box(6);
    if length(h) == 1
        [x, y, z] = ndgrid(x0:h:x1, y0:h:y1, z0:h:z1);
    elseif length(h) == 3
        [x, y, z] = ndgrid(x0:h(1):x1, y0:h(2):y1, z0:h(3):z1);
    end
    node = [x(:), y(:), z(:)];
    [nx, ny, nz] = size(x);
    elem = zeros(6*(nx-1)*(ny-1)*(nz-1), 4);
    indexMap = reshape(1:nx*ny*nz,nx,ny,nz);
    localIndex = zeros(8,1);
    idx = 1;
    for k = 1:nz-1
        for j = 1:ny-1
            for i = 1:nx-1
                localIndex(1) = indexMap(i,j,k);
                localIndex(2) = indexMap(i+1,j,k);
                localIndex(3) = indexMap(i+1,j+1,k);
                localIndex(4) = indexMap(i,j+1,k);
                localIndex(5) = indexMap(i,j,k+1);
                localIndex(6) = indexMap(i+1,j,k+1);
                localIndex(7) = indexMap(i+1,j+1,k+1);
                localIndex(8) = indexMap(i,j+1,k+1);
                elem(idx:idx+5,:) = localIndex([1 2 3 7; 1 4 3 7; 1 5 6 7;...
                                                1 5 8 7; 1 2 6 7; 1 4 8 7]);
                idx = idx + 6;
            end
        end
    end
    N0 = size(node,1);
    HB = zeros(N0,4);
    HB(1:N0,1:3) = repmat((1:N0)',1,3);
end
```

# Cubical Domain III

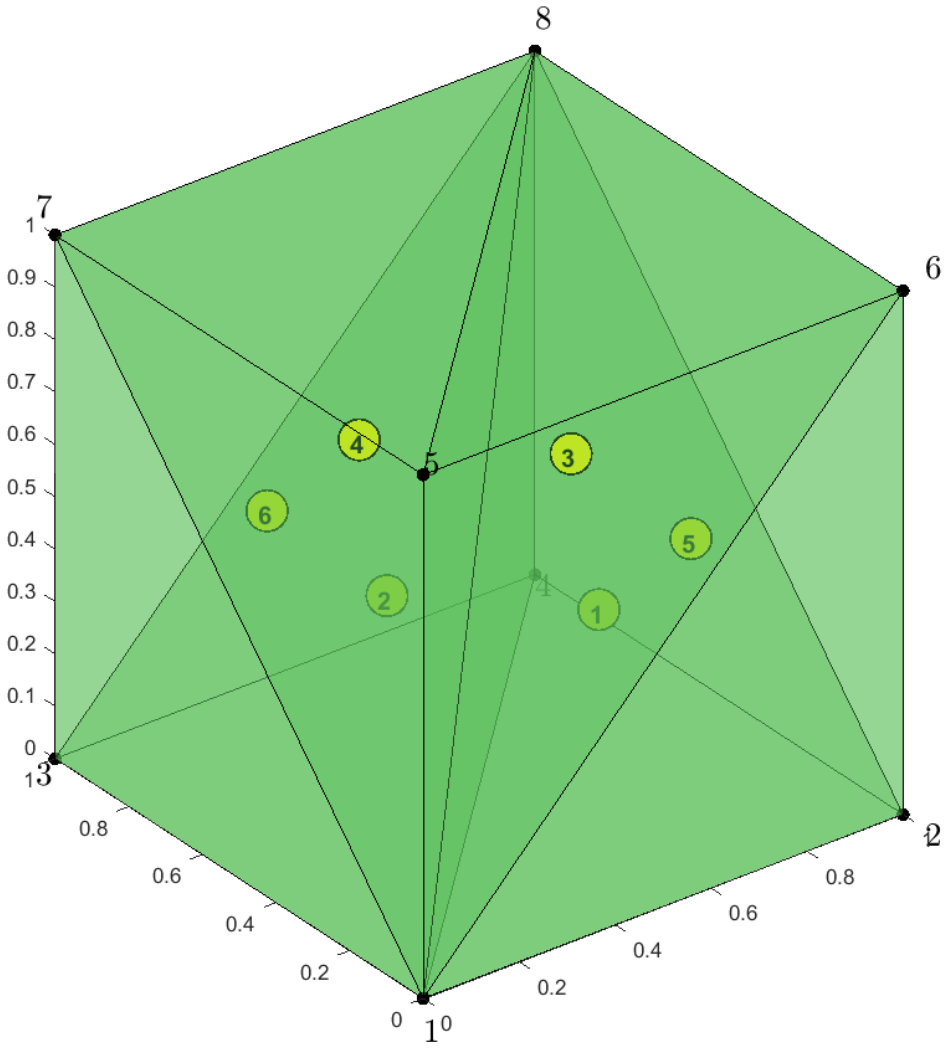


Figure 2: Discretization of Cubical Domain

# Cubical Domain IV

- We will extract the same set of information as in the two dimensional case. However, we are now interested in faces instead of edges. Our set of observations for writing the code are also all similar to the rectangular case. The following code has been used for extracting information about the tetrahedral elements.

# Cubical Domain V

```
function T = auxstructure3(elem)
%%  AUXSTRUCTURE3 auxstructure for 3 dimensional elements
%   T = suxstructure3(elem) makes an indices map between the elements, faces,
%   and nodes and the boundary information. T is a structure.

    totalFace = uint32(sort([elem(:,[2 3 4]); elem(:,[1 4 3]);...
        elem(:,[1 2 4]); elem(:,[1 3 2])], 2));
    [face, i_tot_last, i_face] = myunique(totalFace);
    NT = size(elem, 1);
    elem2face = uint32(reshape(i_face, NT, 4));
    i_tot_first(i_face(4*NT:-1:1)) = 4*NT:-1:1;
    i_tot_first = i_tot_first';
    first_elem_face = ceil(i_tot_first/NT);
    last_elem_face = ceil(i_tot_last/NT);
    first_elem = i_tot_first - NT*(first_elem_face-1);
    last_elem = i_tot_last - NT*(last_elem_face-1);
    diff_ind = (i_tot_first ~= i_tot_last);
    neighbor = uint32(accumarray([[first_elem(diff_ind) first_elem_face(diff_ind)];...
        [last_elem last_elem_face]], [last_elem(diff_ind); first_elem], [NT 4]));
    face2elem = uint32([first_elem last_elem first_elem_face last_elem_face]);
    bdElem = first_elem(first_elem == last_elem);
    first_face_bd = first_elem_face(first_elem == last_elem);
    bdFace = uint32([elem(bdElem(first_face_bd==1),[2 3 4]);...
                     elem(bdElem(first_face_bd==2),[1 3 4]);...
                     elem(bdElem(first_face_bd==3),[1 2 4]);...
                     elem(bdElem(first_face_bd==4),[1 3 2])]);
    bdFace2elem = uint32([bdElem(first_face_bd==1);bdElem(first_face_bd==2);...
        bdElem(first_face_bd==3);bdElem(first_face_bd==4)]);
    T = struct('neighbor',neighbor,'elem2face',elem2face,'face',uint32(face),...
        'face2elem',face2elem,'bdElem',bdElem,'bdFace',bdFace,'bdFace2elem', bdFace2elem);
end
```

# Outline

# Introduction

- Now that we have the mesh structure ready for the cubical domain, we can proceed towards making an FEM solver. We have considered the Poisson Equation as the example.
- We have established the variational formulation for the Poisson Equation, nodal basis functions, finite element discretization and reference element technique.
- We will utilize all of this and put it together with the mesh generation code to obtain a functional FEM solver for the Poisson Equation.

# Model Problem

- Our model problem is defined as,

$$-\Delta u \ = \ f \ in \ \Omega$$

- The variational formulation for the same is,

$$\int_{\Omega} \nabla u . \nabla v dx \ = \int_{\Omega} f v dx \quad \forall v \in H_0^1(\Omega)$$

- The value of our function of interest, $u$, is zero on the entire boundary $\delta\Omega$.
- We take all the basis functions defined on the boundary nodes and faces to be identically zero.

# Removing the Boundary

Since only the internal nodes are of interest to us, we need a way to distinguish between the nodes on the boundary and those in the interior of our domain. Recall that we had stored the list of the faces on the boundary in the structure T during mesh generation. We can make use of that to mark the interior nodes.

```
function [intNodeIndex, intFaceIndex] = remove_bdNodeFace(node, T)
    bdFace = double(T.bdFace);
    [bdNode, ~, ~] = myunique(bdFace(:));
    N = size(node, 1);
    allNode = (1:N)';
    intNodeIndex = allNode(~ismember(allNode, bdNode));
    bdFace = sort(bdFace, 2);
    N = size(T.face, 1);
    allFace = (1:N)';
    intFaceIndex = allFace(~ismember(T.face, bdFace, 'rows'));
end
```

# Basis Functions

- We will use the Reference Element Technique as described earlier to define the basis functions on each node. Therefore, to begin with, let us consider the unit tetrahedron defined by the points $P_1 = (0, 0, 0),\ P_2 = (1, 0, 0),\ P_3 = (0, 1, 0),\ P_4 = (0, 0, 1)$.
- We define the basis functions for these four nodes of the reference tetrahedron. We take $\phi_{P_1}(x, y, z) = 1 - x - y - z$, $\phi_{P_2}(x, y, z) = x$, $\phi_{P_3}(x, y, z) = y$ and $\phi_{P_4}(x, y, z) = z$. All these functions take value zero at all points outside the element.
- We will extend the definition of the map $F_K$, as defined earlier, from two dimensions to three dimensions.

$$F_K(\hat{X}) = P_1^K \phi_1(\hat{X}) + P_2^K \phi_2(\hat{X}) + P_3^K \phi_3(\hat{X}) + P_4^K \phi_4(\hat{X})$$

.

# Solving Varitaional Formulation

$$\int_\Omega \nabla u.\nabla v dx = \int_\Omega f v dx \quad \forall v \in H_0^1(\Omega)$$

- Here, we have already taken $u$ as the linear combination of the nodal basis functions.
- Since the above equation is valid for all $v \in H_0^1(\Omega)$, we can take $v$ to be the basis functions.
- We shall have as many equations as the unknown coefficients. We can break down the above computation into two parts:
  - Calculating the RHS
  - Calculating the LHS

# Calculating the RHS I

- What we are essentially calculating is $\int_{\Omega} f\phi_i dx$.
- Each internal node is a part of $24$ surrounding tetrahedral elements. We can now proceed to approximate the integral over some element $T$ with nodes $P_1, P_2, P_3, P_4$.

$$\int_T f\phi_i dx \approx vol(T) * \frac{f(P_1)\phi_i(P_1) + f(P_2)\phi_i(P_2) + f(P_3)\phi_i(P_3) + f(P_4)}{4}$$

- For any node $a_j$, $\phi_i(a_j) = \delta_{ij}$.

$$\int_{\Omega} f\phi_i dx \approx h_x h_y h_z f(P)$$

- P is the node on which $\phi_i$ has been defined and $h_x, h_y, h_z$ denote the step sizes in the respective directions.

# Calculating the RHS II

```
function L = RHS(node, intNodeIndex, h)
    intNode = node(intNodeIndex, :);
    if size(h) == 1
        h = [h, h, h];
    end
    L = f(intNode(:, 1), intNode(:, 2), intNode(:, 3))*h(1)*h(2)*h(3);
end
```

# Calculating the LHS I

- Calculation of the LHS will involve calculating $\int_T \nabla\phi_i.\nabla\phi_j$.
- $\phi(X) = \hat{\phi}(F^{-1}(X))$.

$$\begin{bmatrix} \frac{\delta\phi}{\delta x} & \frac{\delta\phi}{\delta y} & \frac{\delta\phi}{\delta z} \end{bmatrix} = \begin{bmatrix} \frac{\delta\hat{\phi}}{\delta x} & \frac{\delta\hat{\phi}}{\delta y} & \frac{\delta\hat{\phi}}{\delta z} \end{bmatrix} \begin{bmatrix} \frac{\delta F_1^{-1}}{\delta x} & \frac{\delta F_1^{-1}}{\delta y} & \frac{\delta F_1^{-1}}{\delta z} \\ \frac{\delta F_2^{-1}}{\delta x} & \frac{\delta F_2^{-1}}{\delta y} & \frac{\delta F_2^{-1}}{\delta z} \\ \frac{\delta F_3^{-1}}{\delta x} & \frac{\delta F_3^{-1}}{\delta y} & \frac{\delta F_3^{-1}}{\delta z} \end{bmatrix}$$

- Jacobian $J$ of $F^{-1}$ used above is equal to the matrix $B_K^{-1}$ from reference technique.
- We know the value of $\nabla\hat{\phi}$ and can in turn find the value of $\nabla\phi_i$.
- $\nabla\phi_i.\nabla\phi_j$, in the LHS will hold a non zero value only when both $\phi_i$ and $\phi_j$ belong to the same element.

# Calculating the LHS II

```
function A = LHS(node,elem,intNodeIndex,h)
    N = size(node, 1);
    intFlag = ismember(1:N, intNodeIndex);
    itgrl = sparse(N, N);
    dphi = [1,0,0;0,1,0;0,0,1;-1,-1,-1];
    for i = 1:size(elem, 1)
        loc = elem(i, :);
        F = [node(loc(1),1)-node(loc(4),1),node(loc(2),1)-node(loc(4),1),...
            node(loc(3),1)-node(loc(4),1); node(loc(1),2)-node(loc(4),2),...
            node(loc(2),2)-node(loc(4),2),node(loc(3),2)-node(loc(4),2);...
            node(loc(1),3)-node(loc(4),3),node(loc(2),3)-node(loc(4),3),...
            node(loc(3),3)-node(loc(4),3)];
        for j = 1:4
            if ~intFlag(loc(j))
                continue;
            end
            for k = 1:4
                if ~intFlag(loc(k))
                    continue;
                end
                dphij = dphi(j,:)/F;
                dphik = dphi(k,:)/F;
                calc = dot(dphij, dphik)*h(1)*h(2)*h(3)/6;
                itgrl(loc(k), loc(j)) = itgrl(loc(k), loc(j)) + calc;
            end
        end
    end
    A = itgrl(intNodeIndex,intNodeIndex);
end
```

# Obtaining the Final Solution

- Now that we have solved both the sides of the variational formulation, we have reduced the problem to a system of linear equations $AD = L$.
- From here, we can simply obtain $D$, which represents the coefficients of the linear combination of the nodal basis functions.
- Hence, we obtain our approximation.

$$\tilde{u} = \Sigma D_i \phi_i$$

# Outline

# Example

- Let us consider the following problem

$$-\Delta u = f \ in \ \Omega$$
$$u = 0 \ on \ \delta\Omega$$

- Let $\Omega$ denote the cubical domain $[-1,1] \times [-1,1] \times [-1,1]$.
- Let $f(x,y,z) = 3\pi^2 sin(\pi x) sin(\pi y) sin(\pi z)$.
- It can be verified that $u(x,y,z) = sin(\pi x) sin(\pi y) sin(\pi z)$.
- Our goal is to approximate the function $u$.

# Graphs and Error I

- Since $u : \mathbb{R}^3 \to \mathbb{R}$, to make a graph of the approximation and the actual function we take a slice of domain for a fixed value of z.
- The graphs of the approximation for varying step sizes have been attached.
- The actual function has also been graphed for comparison.

# Graphs and Error II

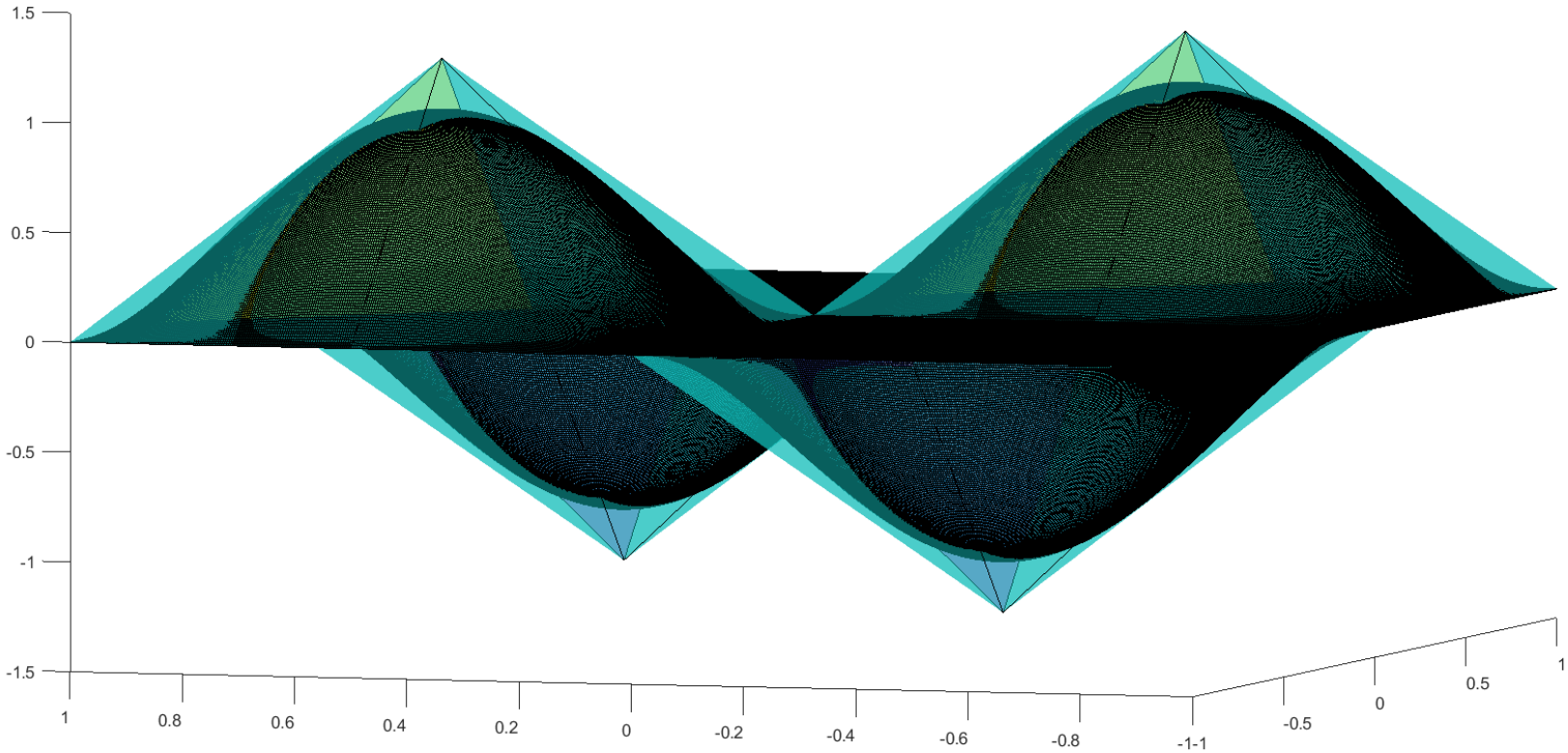


Figure 3: Approximation Obtained For Step Size = 1/2

# Graphs and Error III

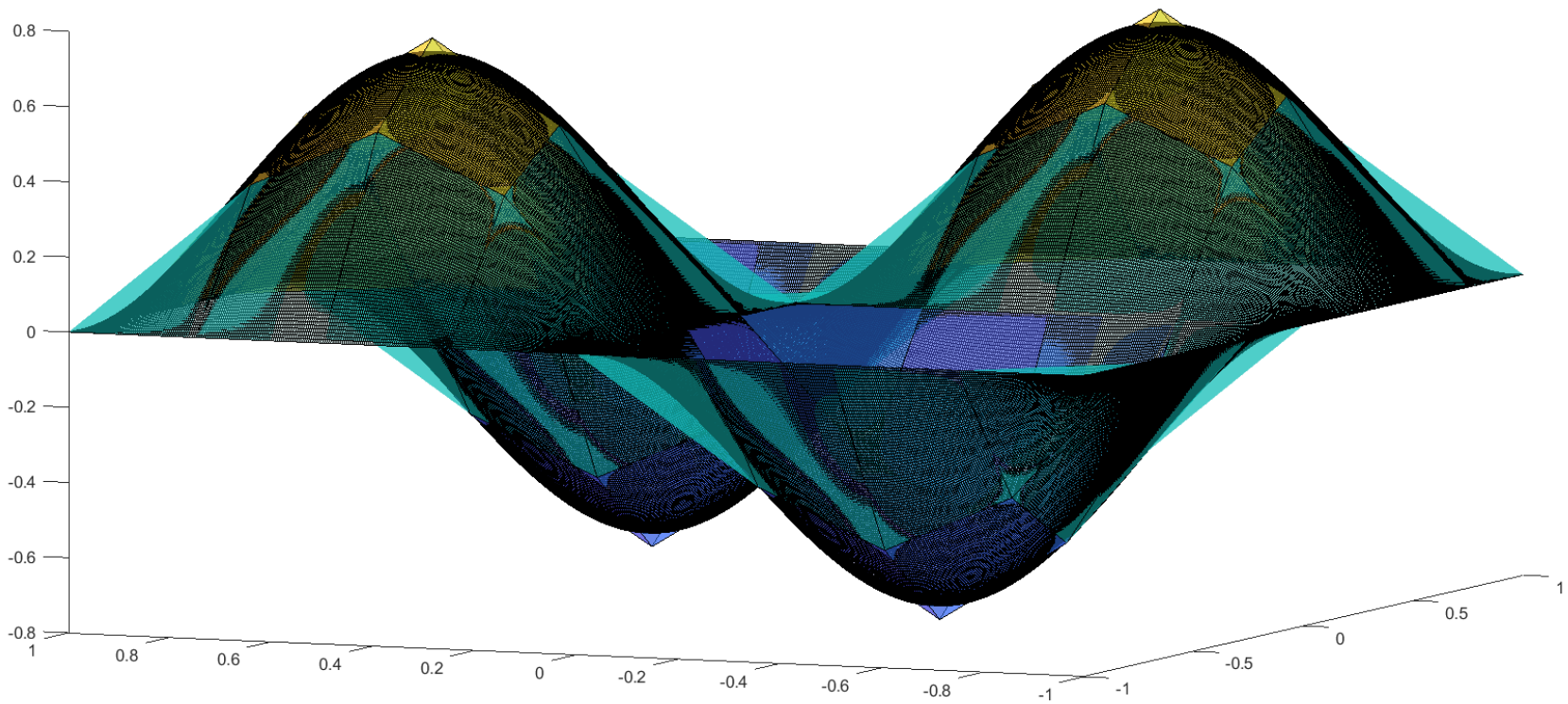


Figure 4: Approximation Obtained For Step Size = 1/4

# Graphs and Error IV

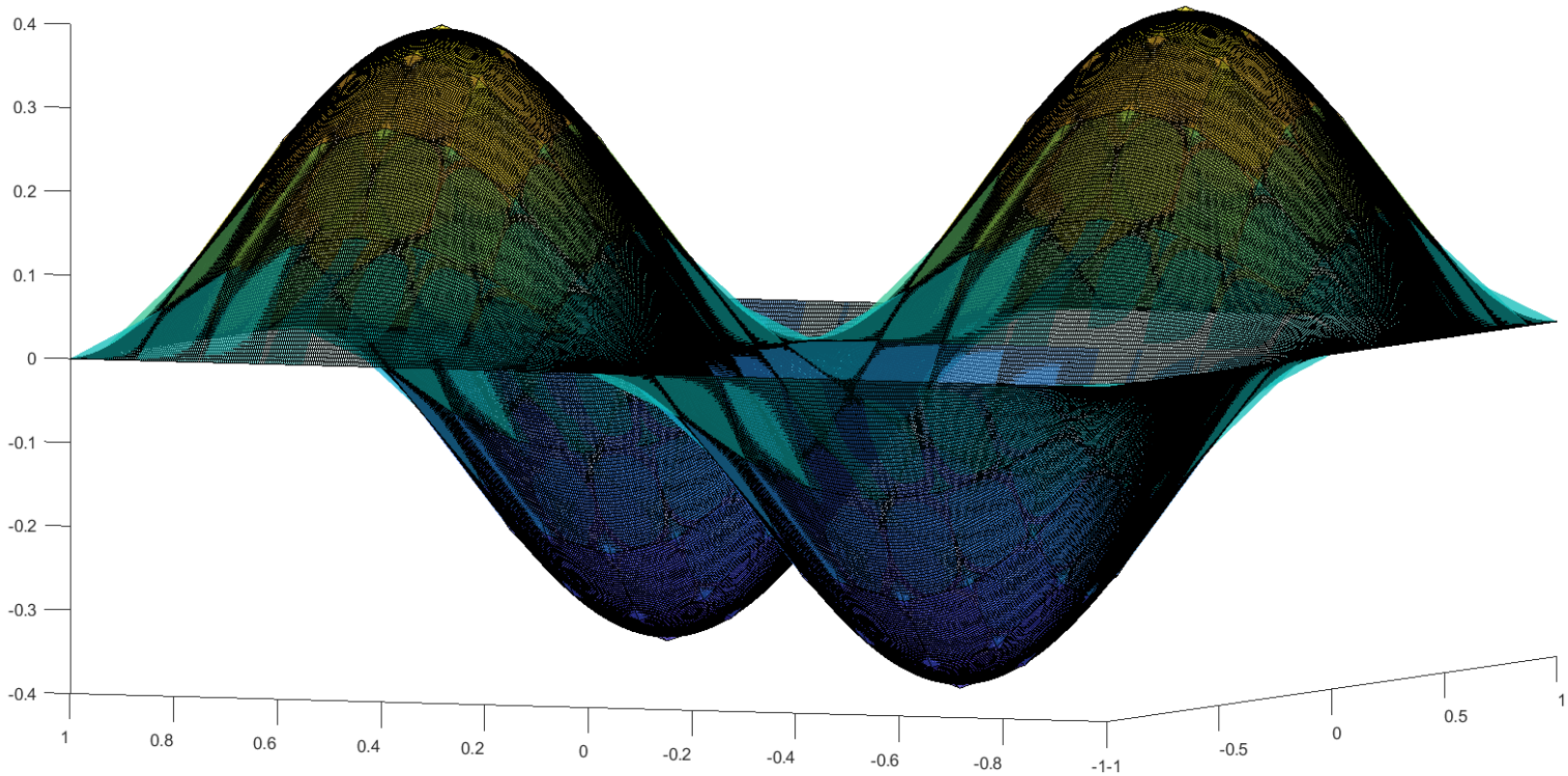


Figure 5: Approximation Obtained For Step Size = 1/8

# Graphs and Error V

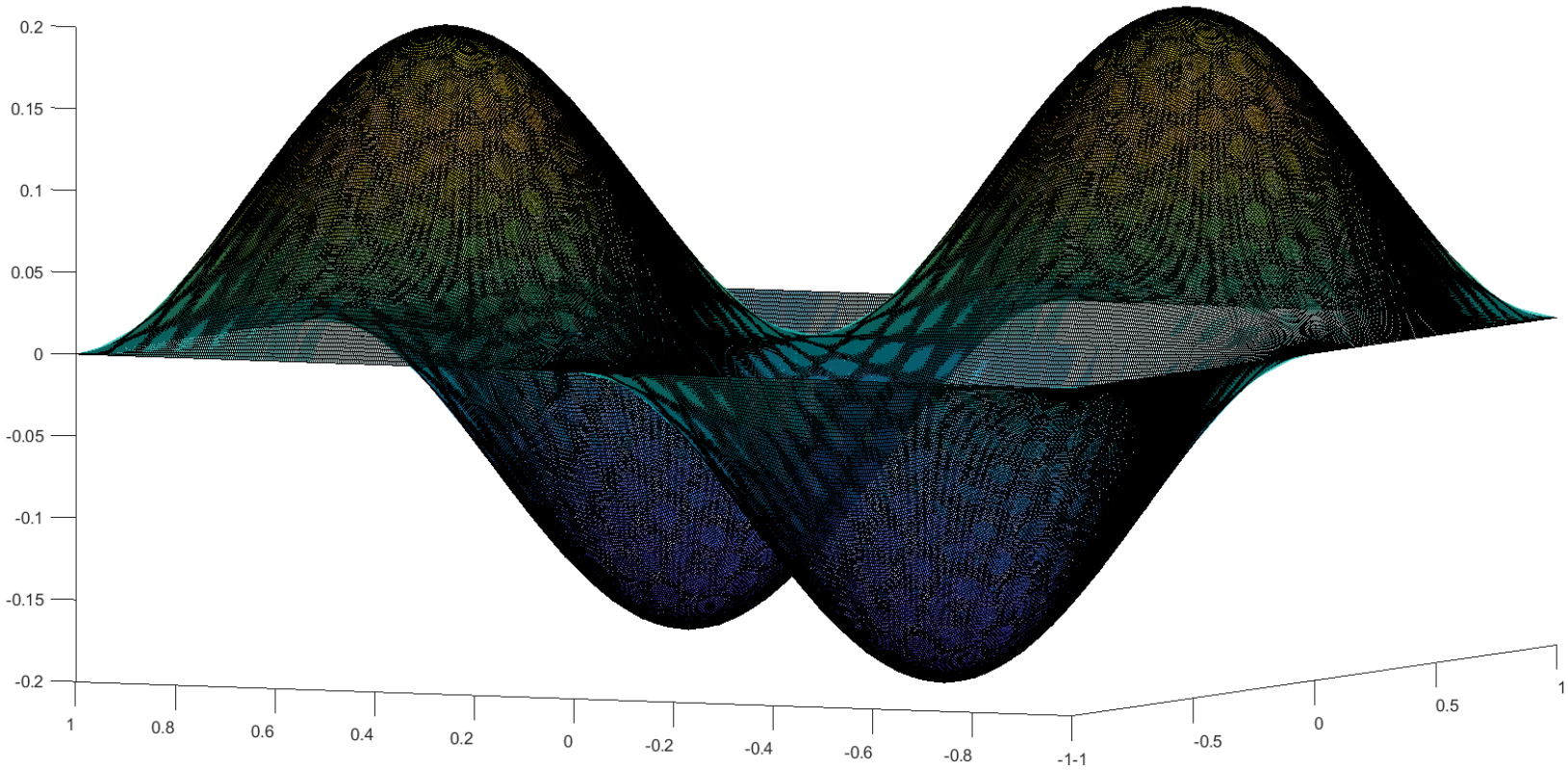


Figure 6: Approximation Obtained For Step Size = 1/16

# Graphs and Error VI

- Let $D$ represent the value of $\tilde{u}$ at the nodes and $val$ represent the value of $u$ at the nodes. The error of approximation has been taken as:

$$E = \sqrt{(D - val)^T A (D - val)}$$

- For varying step sizes we get the error as:

| **Step Size** | **Error** | $E_i / E_{i-1}$ |
|---|---|---|
| 0.5 | 1.1449 | - |
| 0.25 | 0.2812 | 0.2456 |
| 0.125 | 0.0700 | 0.2489 |
| 0.625 | 0.0175 | 0.25 |

- The data matches our expectation that the error should be of order $O(h^2)$. Hence, with decreasing step size our error reduces and our approximation converges to the actual function.

# Thank You!