Theory_FEM

May 30, 2021

1 FEM Theory

Here we describe the Theory behind the Finite Element Method (FEM) scheme we are using to solve the strongly damped wave equation.

1.1 Problem Reduction and FEM Formulation

The numerical scheme we are implementing is based on our basis functions for the discretization of space. Here we first describe that process more rigorously and then we proceed in describing the numerical scheme chosen to perform numerical analysis.

1.1.1 The IVP

We are working on a cyllindrical domain Ω which is a subset of $\mathbb{R}^2$. Specifically:

$$\Omega = S^2 \times (0,1) \subset \mathbb{R}^2$$

Now we are attempting to solve Problem P_1 in that domain.

$$P_1 = \begin{cases} \Delta u(x,t) + \Delta u_t(x,t) - u_{tt}(x,t) = f(x,t) & \forall (x,t) \in \Omega \times \mathbb{R}^+ \\ u(x,0) = 0 & \forall x \in \Omega \\ u(x,t) = 0 & \forall x \in \partial\Omega \times \mathbb{R}^+ \end{cases}$$

Where $u(x,t)$ is our nondimensionalized pressure function, and $f(x,t)$ is the source term. We impose one more restriction to the equation which is that the source term f is axisymmetric. As a result, it is now possible to reduce our problem into two like so. Using cyllindrical coordinates around the symmetry axis of $f(x,t)$ we have that: $f(x,t) = f(r,z,t)$ hence we can rewrite:

$$u(r,\phi,z,t) = u(r,\phi,z,t)v(\phi)$$

Where the difference between $u(r,\phi,z,t)$ and $u(r,\phi,z,t)$ is understood. We hence have:

$$v\Delta_{rz}(u+u_t) + (u+u_t)\Delta_\phi v - vu_{tt} = f$$

Here a solution immediately becomes apparent. $v(\phi) = 1$. We henceforth reduce our equation to cyllindrically symmetric solutions with that condition. This restriction will convert the 3D problem into 2D dramatically decreasing computation time.

Proposition: A solution $u(r, \phi, z, t)$ of the axisymmetric version of P_1 has a local extremum at $r = 0$.

Proof: Note that $u(r, \phi_1, z, t) = u(r, \phi_2, z, t)$. We now employ the smoothness of the solution. Specifically, for any smooth path $\gamma : [0, 1] \to \text{proj}_{\mathbb{R}^2}\Omega$ such that $\exists\ \alpha \in [0, 1]$ s.t. $\gamma(\alpha) = 0$ then the function $\tilde{u}_z$, defined for all z by:

$$\tilde{u}_z : [0, 1] \times \mathbb{R}^+ \to \mathbb{R}$$
$$\tilde{u}_z(s, t) \mapsto u(\gamma(s), z, t) \in \mathbb{R},$$

must be smooth. As a result of continuity for every open set A that contains $\gamma(\alpha)$ there exists an open set B in the domain where $\gamma(B) \subset A$. As a result for two points $p_1,\ p_2 \in B$ st $p_1 \neq p_2$ and $|p_1| = |p_2|$ we have that $\tilde{u}_z(p_1, t) = \tilde{u}_z(p_2, t)$. Hence from MVT there must be a point $p \in B$ such that $\partial_\rho \tilde{u}_z(\gamma(p), t) = 0$. Since this must be true for all open sets A containing α then $p = \alpha$.

Now we only need to show that points p_1 and p_2 exist fot all open sets A. Indeed, take the closest point in B to $\gamma(\alpha)$ be $p_1 = (r, \phi)$. Then we can pick $p_2 = (r, \phi + \pi)$. $\square$

Now that we have established these two conditions we can perform the 1 dinemensional reduction to P_1 from 3D to 2D, by elliminating ϕ-dependance.

1.1.2 Weak formulation

We can convert the wave equation to its weak formulation like so. Consider the space of all locally summable functions in Ω where their derivatives belong to $L^3(\Omega)$, $H^3(\Omega)$. Then, assuming a solution $u \in H^3(\Omega)$ of P_1 then $\forall v \in H^3(\Omega)$ the following must hold for all times $t \in \mathbb{R}^+$.

$$\int_\Omega \nabla v \cdot \nabla u\ dx \int_\Omega \nabla v \cdot \nabla u_t\ dx + \int_\Omega vu\ dx + \int_\Omega vf\ dx$$

Now we define an inner product on the sobolev space $H^3(\Omega)$ like so:

$$(\cdot, \cdot) : H^3(\Omega) \times H^3(\Omega) \to \mathbb{R}$$
$$(u, v) \mapsto \int_\Omega vu\ dx \in \mathbb{R}$$

by slightly abusing our notation we can rewrite our weak equation in terms of the inner product.

$$(\nabla v, \nabla u) + (\nabla v, \nabla u_t) + (v, u_{tt}) = -(v, f)$$

Hence, due to the linearity of the inner product we can define a bilinear functional α and restate our equation as follows:

$$\alpha(v, u) = (\nabla v, \nabla u) + (\nabla v, \nabla u_t) + (v, u_{tt})$$
$$F(v) = -(v, f)$$

$$\alpha(v,u) = F(v)$$

Hence by Lax-Milgram Theorem we have shown that there exists a unique solution for the weak formulation of P_1

1.1.3 Finite Element Method

We want to develop a finite elment method for the new problem P_2, which is basically P_1 but restricted on the half plane U enclosed in cyllinder Ω.

$$P_2 = \begin{cases} \alpha(v,u) = F(v) & \forall v \in H^3(U), t \in mathbbR^+ \\ u(x,0) = 0 & \forall x \in U \\ u(x,t) = 0 & \forall x \in \partial U \times \mathbb{R}^+ st\ z \neq 0 \\ \frac{\partial u}{\partial \nu}\big|_{(x,t)} = 0 & \forall x \in \partial U \times \mathbb{R}^+ st\ z = 0 \end{cases}$$

where ν is the outward normal to $\partial\Omega$

Now the idea is to select a finite dimensional subspace W of $H^3(U)$ such that our function $u \in H^3(U)$ can be approximated by $\tilde{u} \in W$.

To find a suitable subspace W we will discretize the domain Ω into triangular "elements" and associate a function ϕ_i to each element (called a basis function). We will take $W = span(\Phi)$ where $\Phi = \{\phi_i\}$ is the set of all the basis functions for that particular discetization. Then assuming $u(x,t) \approx \tilde{u}(x,t)$ in U we will attempt to solve the (now linear) problem P_3

$$P_3 = \begin{cases} \alpha(v,u) = F(v) & \forall v \in W \\ u(x,0) = 0 & \forall x \in U \\ u(x,t) = 0 & \forall x \in \partial U \times \mathbb{R}^+ st\ z \neq 0 \\ \frac{\partial u}{\partial \nu}\big|_{(x,t)} = 0 & \forall x \in \partial U \times \mathbb{R}^+ st\ z = 0 \end{cases}$$

Notice that now we need not solve for the entire cyllinder, solving for any half plane in the cyllinder would work just fine!

1.2 Discretization

How will we actually discretize our space? We will use triangles. The reason is that we can easily tesselate the space with them, and that for any arbitrary collection of points we can use Delaunay's Algorithm to extract a (not unique) set T of triangle elements T_i.

How do we extract the points? We will take a function over our space domain U. Then we will use the function's gradient to generate a set of points P that will be densest at regions with higher gradient. This way can have finer resolution in the places where the wave function is sharpest, hence adaptively increase decrease our spatial error while also decreasing computation time.

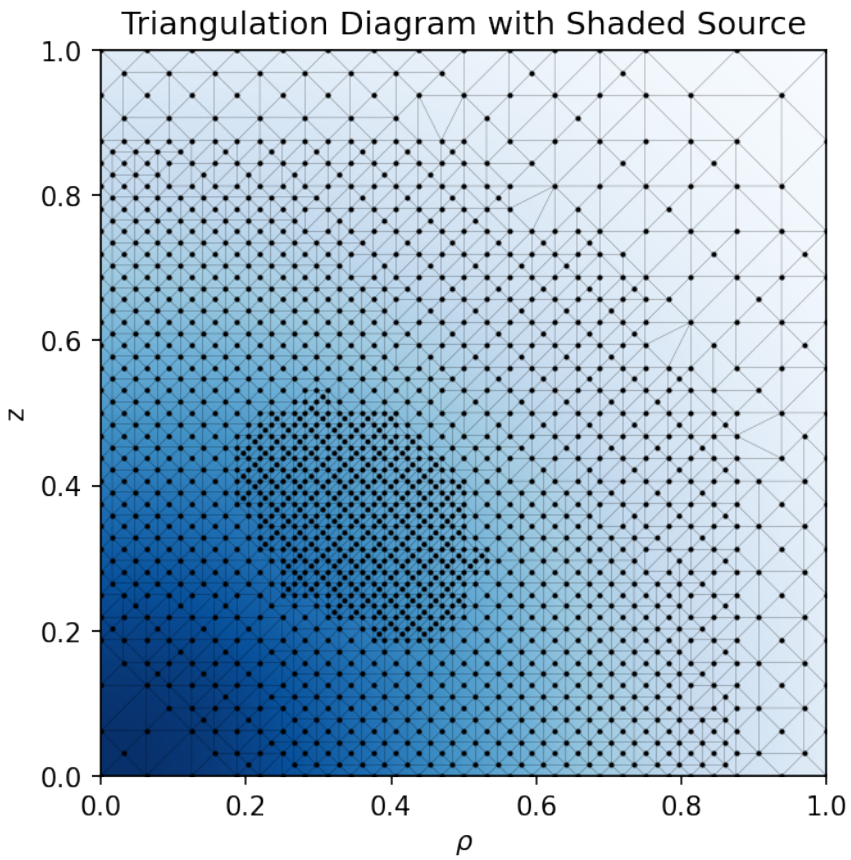
After we have the set of points P we will perform Delaunay's Algorithm to create the set of triangles T. Below is an implementation of such a scheme

[13]:
```
# Let's evaluate and create a mesh for our domain.

# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from tqdm import tqdm

# With this routine we will discretize a space using triangles according to a␣
 ↪size distribution.
# Set the bounds
r_bounds = (0,1)
z_bounds = (0,1)

# We will first create the first points
def merge(A,B):
    '''Merges two lists performing a set product A x B'''
    L = []
    for a in A:
        for b in B:
            L.append([a,b])
    return L

# Create the first 4 points.
points = merge(r_bounds,z_bounds)
ground_box = points.copy()

def center(box):
    '''Returns the central point of a box'''
    x = 0
    y = 0
    for p in box:
        x+=p[0]
        y+=p[1]

    return (x/4,y/4)
points.append(center(ground_box))


# For the subdivision parameters
h = 5e-2                        # How course parameter
# F = lambda r,z: abs(r)          # A function to do the subdivision
F = lambda r,z: np.exp(-(r**2+z**2)/0.5)

# Now we recursively generate the rest of the points.
boxes = []
box_q = [ground_box]            # We start with only one box on the queue
```

```
def create_box(point,box,points):
    '''Creates a new box on the grid'''
    new_box = [point]
    for p in box:
        if p == point: continue

        new_p = ((p[0] + point[0])/2,(p[1] + point[1])/2) # Create new point

        if new_p not in points:     # For if the point is new add it
            points.append(new_p)

        new_box.append(new_p)

    c = center(new_box)
    if c not in points: points.append(c)

    return new_box

def box_needed(point,box,F,h=h):
    '''Given a particular Function F, and a box, it returns weather or
    not there needs to be a finer asjustment in the mesh at said point'''
    needed = False
    for p in box:
        if p == point: continue

        if abs(F(*p)-F(*point)) > h: return True

    return needed

# While there are boxes left to check
while len(box_q) > 0:
    box = box_q[0]                  # Get the current box

    # For all the points in the box
    # If we need to add a new box add it
    for point in box:
        if box_needed(point,box,F):
            new_box = create_box(point,box,points)      # Create a box
            box_q.append(new_box)                       # Add the new box to␣
 ↪the box queue

    # Remove the current box from the boxes
    box_q.pop(0)

points = np.array(points)
```

```
# Let's reorder our points accordingly
def reorder(points):
    scores = np.array([len(points)*point[1]+point[0] for point in points])
    indx   = np.arange(0,len(points))

    sorted_pairs = sorted(zip(scores,indx))

    tuples = zip(*sorted_pairs)
    scores, indx = [ list(tuple) for tuple in  tuples]

    return points[indx]

# This step is important to get prettier sparse matrices
points = reorder(points)


# We can generate a mesh using Delaunay Triangulization
from scipy.spatial import Delaunay
mesh = Delaunay(points)

plt.figure(figsize=(5,5),dpi=150)
plt.scatter(*np.array(points).T,s=1,marker='o',c='k')

Npts = 50
r = np.linspace(*r_bounds,Npts)
z = np.linspace(*z_bounds,Npts)
extent = np.min(r), np.max(r), np.min(z), np.max(z)

R,Z = np.meshgrid(r,z)
FF = np.flip(F(R,Z),axis=0)
plt.imshow(FF,cmap=cm.Blues,interpolation='bilinear',extent=extent)

plt.triplot(points[:,0], points[:,1], mesh.simplices,lw=0.1,c='k');
plt.title('Triangulation Diagram with Shaded Source')
plt.xlabel(r'$\rho$')
plt.ylabel('z')
```

[13]: Text(0, 0.5, 'z')



1.3 Numerical Scheme

Now that we have a triangulation it is time to start deriving our basis functions. Each point $x_i \in P$ will have its own basis function $\phi_i \in \Phi$ associated with it. Then we will assume that the solution is a sum of those basis functions and derive an implicit linear system to calculate everything.

1.3.1 Basis functions

For each point x_k in a triangle T_i we will define a linear function $\phi_{i,k} : T_i \to \mathbb{R}$ such that it is linear inside the triangle, 1 at the point x_k and 0 at the edge across it. To be specific, consider the triangle T_0 to be the *"ground triangle"* (Orthogonal triangle with side length one centered at the origin). We can describe every triangle using a linear transformation $\tau_i : T_0 \to T_i$ defined using the three edge points of T_i (x_{i1}, x_{i2}, and x_{i3}) like so:

$$\tau_i(r, z) = (1 - r - z)\, x_{i1} + r\, x_{i2} + z\, x_{i3}$$

As a result, we can define the three functions $\phi_{0,k}$ from above for T_0 and have that: $\phi_{i,k}(r, z) = \phi_{0,k}(\tau^{-1}(r, z))$. Hence the three functions are defined like so:

$$\phi_{0,1}(r,z) = 1 - r - z$$
$$\phi_{0,2}(r,z) = r$$
$$\phi_{0,3}(r,z) = z$$

We can plot these three functions below. Note that these 3 functions are not basis functions for W each basis is a sum of these functions for each point. To be a bit more rigorous we will define the **basis function** $\phi_i(x)$ for some point $x_i \in U$ like so:

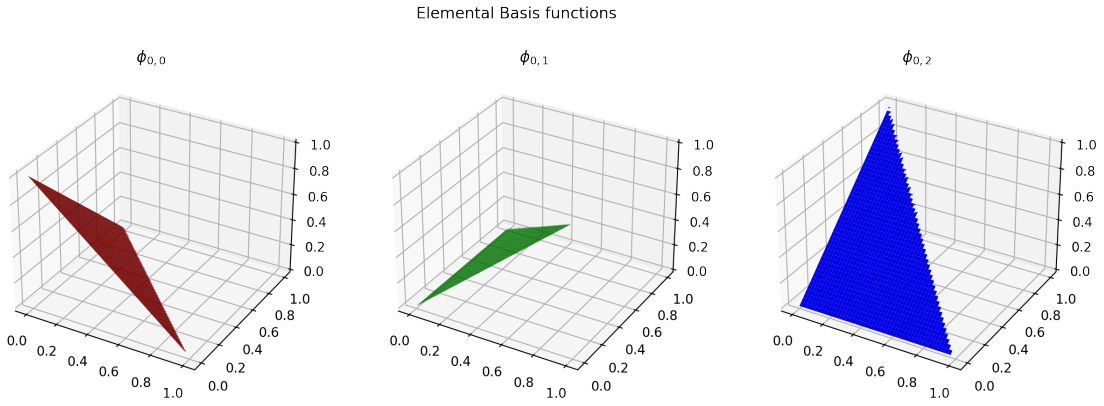
$$\phi_i(x) = \sum_{j \in I(N(x_i))} \phi_{j,k(x_i)}(x)$$

where $N(x_i)$ is the set of immidate neighbors of x_i, I(A) denotes an index set of A and $k(x_i)$ is a map from P to the index of the point in the ground triangle that x_i maps to under the inverse transformation $\tau(r,z)$

[37]:
```
# Plot the three elmental basis functions
fig = plt.figure(figsize=(15,5),dpi=200)
fig.suptitle('Elemental Basis functions')
ax1 = fig.add_subplot(131,projection='3d')
ax2 = fig.add_subplot(132,projection='3d')
ax3 = fig.add_subplot(133,projection='3d')

ax = [ax1,ax2,ax3]

# Define the three functions
f1 = lambda r,z: 1-r-z
f2 = lambda r,z: r
f3 = lambda r,z: z

f = [f1,f2,f3]
C = ['r','g','b']

# Get a triagnle domain
Npts = 100
r = np.linspace(0,1,Npts)
z = np.linspace(0,1,Npts)
R,Z = np.meshgrid(r,z)

for i in range(3):
    F = f[i](R,Z);
    out = np.where(F<0)
    R[out] = None
    Z[out] = None
    F[out] = None

    ax[i].plot_surface(R,Z,F,color=C[i]);
```

```
    ax[i].set_title(r'$\phi_{0,%d}$'%i)
```

```
<ipython-input-37-6ad17c8f1657>:31: UserWarning: Z contains NaN values. This may
result in rendering artifacts.
  ax[i].plot_surface(R,Z,F,color=C[i]);
<ipython-input-37-6ad17c8f1657>:26: RuntimeWarning: invalid value encountered in
less
  out = np.where(F<0)
```



1.3.2 The actual scheme

Now what we have our functions we can establish that the solutions u to our wave equation can be approximated by our basis functions. Remember by the definition of subspace W we have that the finite subset $\Phi = \{\phi_i\}$ is a basis. Hence we have that:

$$u(x,t) \approx \tilde{u} = \sum_n C_n(t)\phi_n(x)$$

Now we will use this to derive a numerical scheme for problem P_3. To do this we have to calculate the following integrals. But before we do, notice that the product of two basis functions is 0 unless they belong to the same Trinangle T_i. Specifically:

$$\phi_i(x)\phi_j(x) = 0 \text{ iff } \nexists\, T_\alpha \in T \text{ s.t. } x_i, x_j \in T_\alpha$$

for v in W the inner product can be calculated like so. Consider the set $X = \{T_\alpha \in T | \text{ st } x_i, x_j \in T_\alpha\}$

$$(\phi_i, \phi_j) = \int_\Omega \phi_i\, \phi_j\, dx = \sum_\alpha \int_\Omega (\phi_{a,k(x_i)} \cdot \phi_{a,k(x_j)})(\tau^{-1}(x))dx = \sum_\alpha J_\alpha \int_\Omega (\phi_{0,k(x_i)} \cdot \phi_{0,k(x_j)})(x)dx := \sum_\alpha J_\alpha T_{ij}$$

$$(\nabla\phi_i, \nabla\phi_j) = \int_\Omega \nabla\phi_i\, \nabla\phi_j\, dx = \sum_\alpha \int_\Omega (\nabla\phi_{a,k(x_i)} \cdot \nabla\phi_{a,k(x_j)})(\tau^{-1}(x))dx$$

$$= \sum_{\alpha} J_\alpha \int_\Omega (\nabla \phi_{0,k(x_i)} \cdot \nabla \phi_{0,k(x_j)})(x) dx := \sum_{\alpha} J_\alpha S_{ij}$$

where J_α is the Jacobian of $\tau(x)$. As a result we can reqrite the equation of P_3 like so:

$$S(\vec{u}_t + \vec{u}) + T(\vec{u}_{tt} + \vec{F}) = 0$$

Where $\vec{u}$ and $\vec{F}$ are the vectors like so $\vec{u} = (u(x_0) \cdots u(x_n))$ for $x_i \in P$

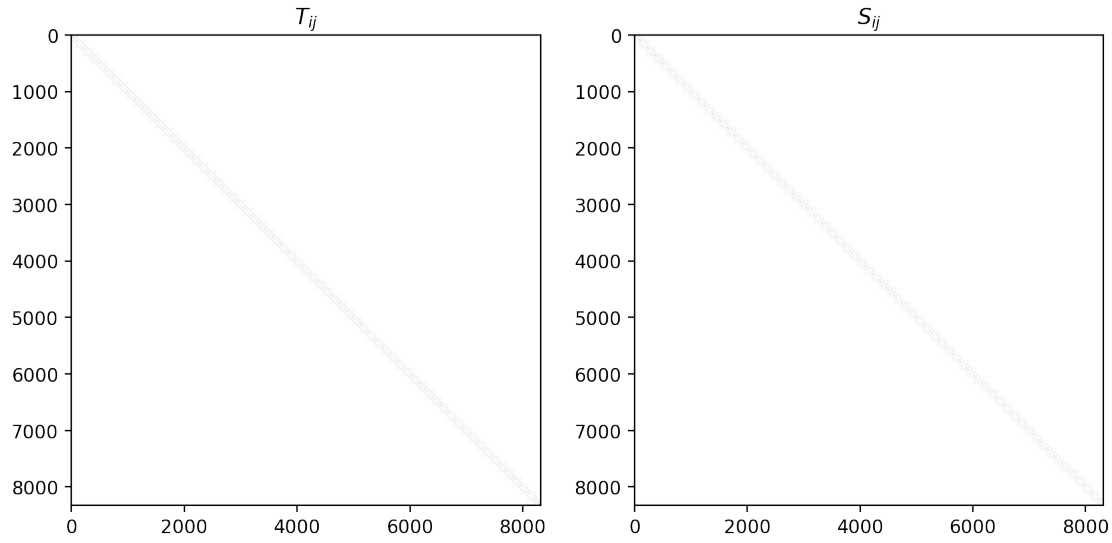
Now to do the time forwarding we will use euler for the second derivative, and backward euler for the first so that we can have an implicit, convergent scheme. As a result the numerical scheme becomes:

$$S\left[\frac{1}{\Delta t}((1+\Delta t)\vec{u}^{+1} - \vec{u}^n)\right] + T\left[\frac{1}{\Delta t^2}(\vec{u}^{n+1} - 2\vec{u}^n + \vec{u}^{n-1}) + \vec{F}^n\right] = 0$$

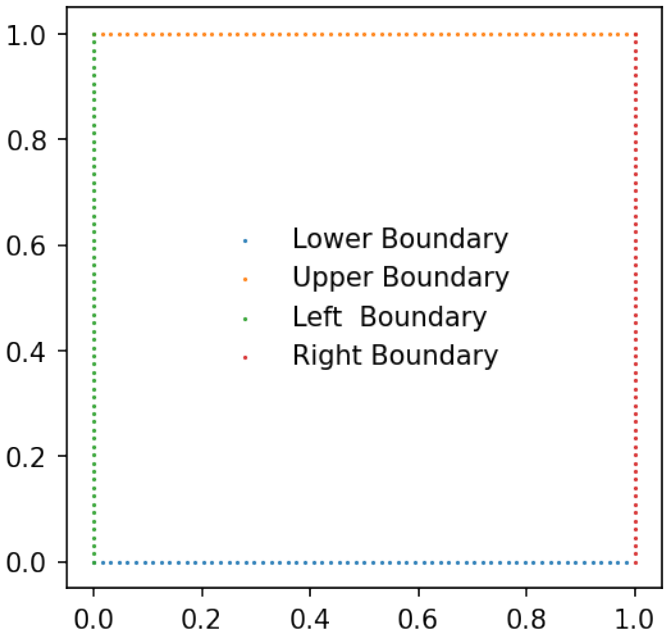
[2]:
```
# Having the triangles we need to get our S, and T matrices
# Let's start with T_ij which is the simpler one

N_points = len(points)

T = np.zeros((N_points,N_points))
S = np.zeros((N_points,N_points))

def get_simplices(tri, point):
    '''Find all simplices this point belongs to'''

    visited = set()
    queue = [tri.vertex_to_simplex[point]]
    while queue:
        simplex = queue.pop()
        for i, s in enumerate(tri.neighbors[simplex]):
            if tri.simplices[simplex][i] != point and s != -1 and s not in␣
 ↪visited:
                queue.append(s)
        visited.add(simplex)
    return np.array(list(visited))

def jacobian(p):
    '''Calculates the jacobian of the triangle transformation given the 3␣
 ↪points'''
    return␣
 ↪(p[1][0]-p[0][0])*(p[2][1]-p[1][1])-(p[2][0]-p[0][0])*(p[1][1]-p[0][1])

# The matrices with the integrals for T and S
A_T = np.pi/60 * np.array([[2,2,1],  [2,6,2], [1,2,2] ])
A_S = np.pi/3  * np.array([[2,-1,-1],[-1,1,0],[-1,0,1]])
```

```
# For all the points
for i in range(len(points)):
    # Find the neighboring triangles to x_i
    nei_tri = get_simplices(mesh,i)
    nei_pts = np.unique(mesh.simplices[nei_tri].reshape(-1))

    # For all the neighboring triangles
    for triangle in nei_tri:
        p_indx = mesh.simplices[triangle]    # Get the points for set triangle
        pts    = points[p_indx]             # Get the number of points
        J      = jacobian(pts)              # Calculate the jacobian for this␣
 ↪triangle

        # the index (1,2,3) of the ith point is:
        ki = p_indx.tolist().index(i)

        for kj in range(len(p_indx)):        # For each of those points
            # We need to unpack if this point is 1,2, or 3.
            # To do that we use the index of the triangle list
            j = p_indx[kj]
            T[i][j] += A_T[ki][kj]*J
            S[i][j] += A_S[ki][kj]*J
```

[3]:
```
# The two sparse matrices visualised

fig = plt.figure(figsize=(10,5),dpi=200)
ax_T = fig.add_subplot(121)
ax_S = fig.add_subplot(122)
ax_T.imshow(T!=0,cmap=cm.binary)
ax_S.imshow(S!=0,cmap=cm.binary)

ax_T.set_title(r'$T_{ij}$');
ax_S.set_title(r'$S_{ij}$');
```



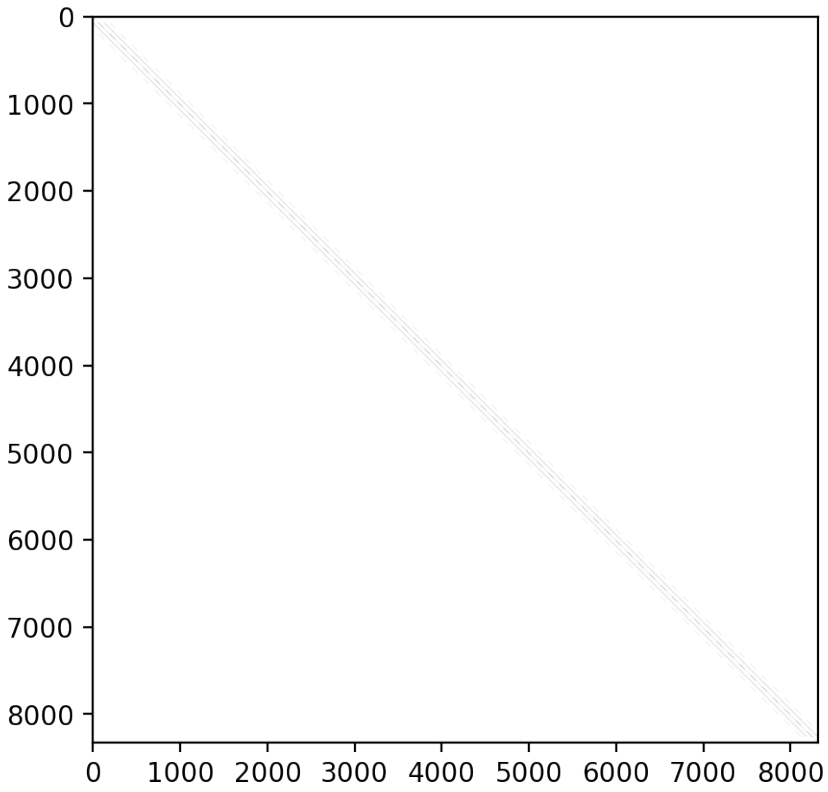
[4]:
```
# Boundary Conditions
# To implement boundary conditions we have to identify the edges of our points.


def get_boundary(points):
    '''Returns a list of boundary point indexes'''

    pts = points.tolist()
    bd_lower = np.arange(0,pts.index([1,0]))
    bd_upper = np.arange(pts.index([0,1]),len(pts))
    bd_left  = [pts.index(p) for p in pts if p[0]==0]
    bd_right = [pts.index(p) for p in pts if p[0]==1]

    return bd_lower,bd_upper,bd_left,bd_right

bd_lower,bd_upper,bd_left,bd_right = get_boundary(points)

# Print the boundary
plt.figure(figsize=(4,4),dpi=150)
plt.scatter(*points[bd_lower].T,label='Lower Boundary',marker='.',s=1)
plt.scatter(*points[bd_upper].T,label='Upper Boundary',marker='.',s=1)
plt.scatter(*points[bd_left].T, label='Left  Boundary',marker='.',s=1)
plt.scatter(*points[bd_right].T,label='Right Boundary',marker='.',s=1)
plt.legend(frameon=False);
```



[5]:
```
# Now that we have these we only need to write a program that would solve the␣
 ↪numerical scheme we have derived
# The numerical scheme is in the form A U_n+1 = B U_n + C U_n-1 + F

# So we will write something to solve A U = B and then proceed
from scipy.sparse.linalg import spsolve
from scipy.sparse import csc_matrix
from scipy.linalg import solve

# Define appropriate simulation constants
dt = 1e-3   # Time step
v  = 1      # Source wave speed
s  = 1e-2   # Source std
o  = 0.1    # Source offset

# Define a Source function
f = lambda r,z,t: np.exp(-((z-v*t-o)**2 + r**2)/s)

# Define the initial condition vectors
# Goal is to solve for U_next, using U_curr and U_prev
U_curr = np.zeros(len(points))
U_prev = np.zeros(len(points))
U_next = np.zeros(len(points))
F      = np.array([f(*point,0) for point in points])
```

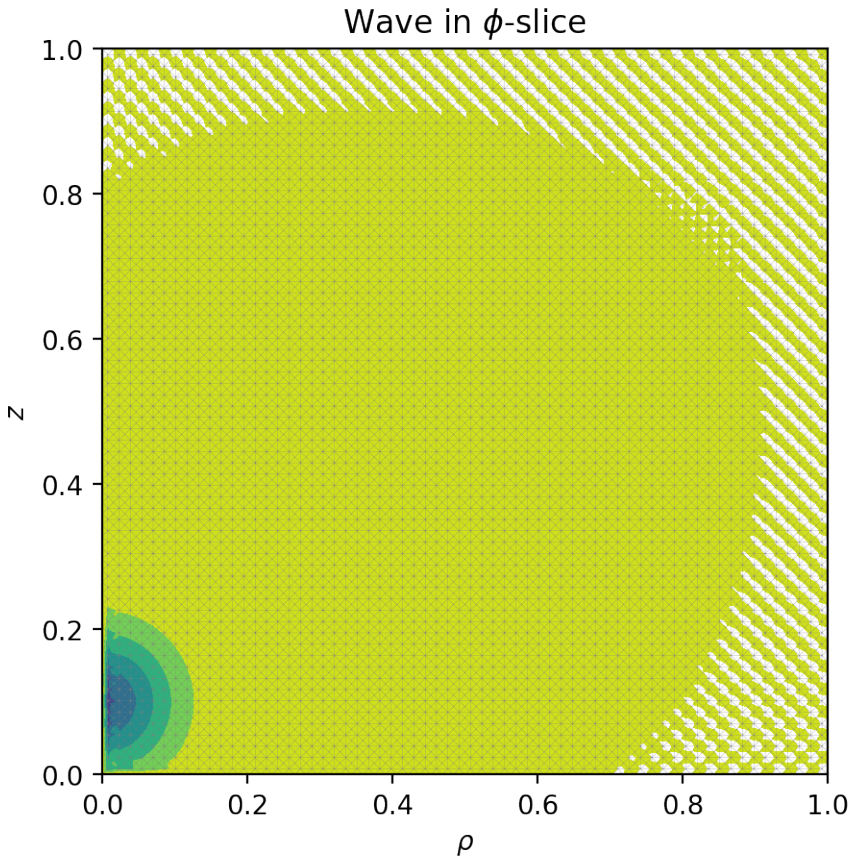
```
# Define the relevant numerical Scheme matrices
A = T/dt + (1+dt)*S
B = np.matmul(S + (2/dt) * T,U_curr) - np.matmul(T,U_prev/dt) - dt*np.
 ↪matmul(T,F)

# Now we need to apply boundary conditions and we are ready to go
# The conditions are as follows:
# Upper: Dirichlet  |   U = 0
# Lower: Dirichlet  |   U = 0
# Left : Neuman     |   dU/dr = 0
# Right: Dirichlet  |   U = 0

pts = points.tolist()

# Hence we will need to change the matrix elements for the points on the␣
 ↪boundaries appropriately.
# Dirichlet
# Lower
B[bd_lower] = 0
for p in bd_lower:
    for i in range(len(pts)): A[p][i] = 0
    A[p][p] = 1

# Upper
B[bd_upper] = 0
for p in bd_upper:
    for i in range(len(pts)): A[p][i] = 0
    A[p][p] = 1

# Right
B[bd_right] = 0
for p in bd_right:
    for i in range(len(pts)): A[p][i] = 0
    A[p][p] = 1

# Neumann
# Left
B[bd_left] = 0
for p in bd_left:
    for i in range(len(pts)): A[p][i] = 0
    A[p][p] = 1

    nei_pts = [ pt for pt in np.unique(mesh.simplices[get_simplices(mesh,p)].
 ↪reshape(-1)) if points[p][0]!= 0]
    for i in nei_pts:A[p][i] = -1/len(nei_pts)
```

[6]:
```
# Plot the matrix to solve
fig = plt.figure(figsize=(5,5),dpi=200)
plt.imshow(A!=0,cmap=cm.binary)

A = csc_matrix(A,dtype=float)
```



[7]:
```
# Finally use a sparse solving algorithm
U_next = spsolve(A,B)

fig = plt.figure(figsize=(5,5),dpi=200)
ax  = fig.add_subplot(111)
ax.set_aspect('equal')
ax.set_title(r'Wave in $\phi$-slice')
ax.set_xlabel(r'$\rho$')
ax.set_ylabel(r'$z$')

ax.triplot(points[:,0], points[:,1], mesh.simplices,lw=0.1,c='grey')
ax.tricontourf(points[:,0], points[:,1], mesh.simplices,U_next)
```

[7]:
```
<matplotlib.tri.tricontour.TriContourSet at 0x7fac838c75b0>
```


Wave in ϕ-slice
1.0
0.8
0.6
0.4
0.2
0.0
z
0.0
0.2
0.4
0.6
0.8
1.0
ρ