Homework ... the last one.

The goal of this worksheet is to put into practice what we learned about finite element matrices formed from linear elements on triangular meshes - and then have a bit of fun with that code

Background

The following is a quick summary of what we discussed in class about the FEM for linear elements in triangular meshes. Problem 1 will ask you to program this and test that code.

Local Mass and Stiffness Matrices

For the k^{th} triangular element in a mesh, whose vertices are

$$\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3,$$

we can express the edges of the triangle as

$$\begin{aligned}\mathbf{E}_1 &= \mathbf{r}_3 - \mathbf{r}_2 \\ \mathbf{E}_2 &= \mathbf{r}_1 - \mathbf{r}_3 \\ \mathbf{E}_3 &= \mathbf{r}_2 - \mathbf{r}_1\end{aligned}$$

(where we use a naming convention that $\mathbf{E}_i$ is the edge opposite $\mathbf{r}_i$, and oriented so the edges point clockwise around the triangle), and the area of the triangle as

$$A = \frac{1}{2}||\mathbf{E}_1 \times \mathbf{E}_2||_2.$$

Given these definitions, we can express the **local** stiffness and mass matricees (S^k and M^k) as

$$S^k = \frac{1}{4A}\begin{bmatrix}\mathbf{E}_1\cdot\mathbf{E}_1 & \mathbf{E}_1\cdot\mathbf{E}_2 & \mathbf{E}_1\cdot\mathbf{E}_3 \\ \mathbf{E}_2\cdot\mathbf{E}_1 & \mathbf{E}_2\cdot\mathbf{E}_2 & \mathbf{E}_2\cdot\mathbf{E}_3 \\ \mathbf{E}_3\cdot\mathbf{E}_1 & \mathbf{E}_3\cdot\mathbf{E}_2 & \mathbf{E}_3\cdot\mathbf{E}_3\end{bmatrix}, \quad M^k = \frac{A}{12}\begin{bmatrix}2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2\end{bmatrix} \tag{1}$$

Global Mass and Stiffness Matrices

Say that we have a list of N_v 3D vertices, V, and a list of N_t triangles, T. To get the vertices k_1, k_2, k_3 of the k^{th} triangle, and their 3D positions, in MATLAB we would do something like

```
% get vertex indices of k^th triangle as 1x3 vector of integers
tri = T(k,:);
% parse each index from 'tri'
k_1 = tri(1);
k_2 = tri(2);
k_3 = tri(3);
% get the 3D positions of the vertices
r_1 = V(k_1,:);
r_2 = V(k_2,:);
r_3 = V(k_3,:);
```

The previous section tells you how to compute local mass and stiffness matrices from $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3$ but we still need to assemble the **global** mass and stiffness matrices from the local pieces. For instance, the $(1,3)$ entry of the k^{th} *local* stiffness matrix contributes to the (k_1, k_3) entry of the *global* stiffness matrix - and the full *global* stiffness matrix is the **sum** of all of the local stiffness matrices. In math, this means that

$$\mathbf{S}^k_{global}(k_i, k_j) = \mathbf{S}^k(i,j)$$
$$\mathbf{M}^k_{global}(k_i, k_j) = \mathbf{M}^k(i,j)$$

for $i, j = 1, 2, 3$, and

$$\mathbf{S} = \sum_{k=1}^{N_t} \mathbf{S}^k_{global}$$
$$\mathbf{M} = \sum_{k=1}^{N_t} \mathbf{M}^k_{global}$$

[**Note: be sure to use sparse matrices here since you'll quickly run out of memory if you allocate big matrices with lots of zeros**]

Problem 1: Solving a Poisson equation on a sphere

Based on the previous section, implement a function to compute **S** and **M** from a $N_v \times 3$ array of vertices and an $N_t \times 3$ array of triangles.

To test this function, we'll try to solve the Poisson equation on the surface of a sphere

$$\Delta_{sphere} u = f \tag{2}$$
$$\frac{1}{\sin^2(\theta)} \frac{\partial^2 u}{\partial \phi^2} + \frac{1}{\sin(\theta)} \frac{\partial}{\partial \theta} \left(\sin(\theta) \frac{\partial u}{\partial \theta} \right) = f(\theta, \phi), \tag{3}$$

where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi]$. Specifically, we'll take

$$f = -2\cos(\theta)$$

so that the exact solution is

$$u = \cos(\theta)$$

Note that $u = \cos(\theta) + c$ also solves the Poisson problem for any constant c, but choosing $c = 0$ gives us the solution with the smallest 2-norm.

For our finite element problem we need a good triangulation of the sphere which is provided by icosphere on Matlab's file exchange (a similar version can be pip installed with python). Further, we need basis coefficients f_i such that

$$f \approx \sum_i^{N_v} f_i \psi_i(\mathbf{x})$$

where $\psi_i(\mathbf{x})$ are the linear hat functions defined on our mesh. This approximation of f is just linear Lagrange interpolation, so we can recognize f_i as f evaluated at the position of the i^{th} vertex. In MATLAB this can be accomplished as something like

```
% V is an N_v X 3 array of vertices
f = @(th,ph) -2*cos(th);
[phi, theta, ~] = cart2sph(V(:, 1), V(:, 2), V(:, 3));
theta = theta + pi/2; %put theta in [0,pi]
f_i = f(theta,phi);
```

Now we can find a weak solution to our Poisson problem by solving

$$\mathbf{Su} = \mathbf{Mf}$$

for $\mathbf{u} = [u_i]$, such that

$$u^{FEM} \approx \sum_i^{N_v} u_i \psi_i(\mathbf{x}).$$

But you will see that S is singular!. This is because the original PDE does not have a unique solution either (only determined up to a constant) so a vector of all 1s is in the nullspace of **S**. To get around this we seek a solution with minimal norm (like we did with the PDE) and this is accomplished by solving instead

$$(\mathbf{S} + \epsilon \mathbf{I})\,\mathbf{u} = \mathbf{Mf}$$

where ϵ is some small number (just take $\epsilon = 10^{-8}$). This approach is equivalent to simulating heat flow over a long time since the Poisson equation represents the steady state.

We can evaluate the accuracy of our solution using the exact coefficients u_i^{exact}, computed similar to f_i as

```
% V is an N_v X 3 array of vertices
u_ex = @(th,ph) cos(th);
[phi, theta, ~] = cart2sph(V(:, 1), V(:, 2), V(:, 3));
theta = theta + pi/2; %put theta in [0,pi]
u_ex_i = u_ex(theta,phi);
```

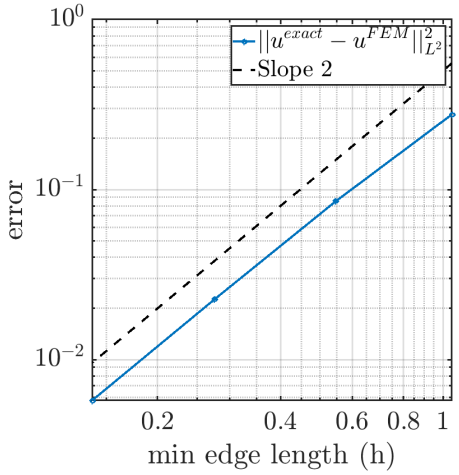
the L^2 error in the solution is computed as

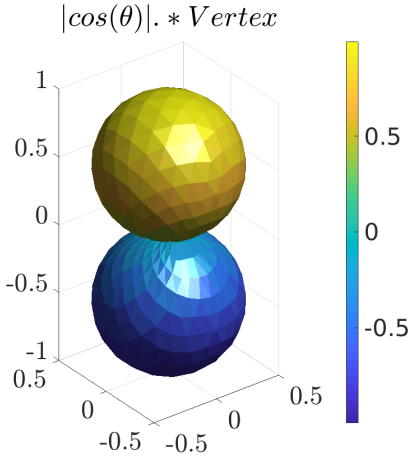
$$\begin{aligned}
||u^{\text{exact}} - u^{FEM}||_{L^2}^2 &= \int_{S^2} \left(\sum_i^{N_v} \left[u_i^{\text{exact}} - u_i^{FEM}\right] \psi_i(\mathbf{x}) \right)^2 \\
&= \int_{S^2} \sum_j^{N_v} \left[u_j^{\text{exact}} - u_j^{FEM}\right] \psi_j(\mathbf{x}) \left(\sum_i^{N_v} \left[u_i^{\text{exact}} - u_i^{FEM}\right] \psi_i(\mathbf{x}) \right) \\
&= \sum_j^{N_v} \left[u_j^{\text{exact}} - u_j^{FEM}\right] \left(\sum_i^{N_v} \left[u_i^{\text{exact}} - u_i^{FEM}\right] \left\{ \int_{S^2} \psi_j(\mathbf{x}) \psi_i(\mathbf{x}) \right\} \right) \\
&= \left[\mathbf{u}^{\text{exact}} - \mathbf{u}^{FEM}\right]^T \mathbf{M} \left[\mathbf{u}^{\text{exact}} - \mathbf{u}^{FEM}\right]
\end{aligned}$$

Show that the error decreases quadratically as you refine the mesh by factors of 2 using a log-log plot.

I know this was a lot of background, but the crux of the question is simple:

1. Define a mesh and compute the smallest edge length (smallest edge length is easy to find but in general it's a bad measure of the mesh size)
   (a)
```
% N defines the level of refinement for the sphere.
% Increasing N by 1 roughly halves the average edge length
N = 0;
[V,T] = icosphere(N);
h_mesh(N+1) = min(pdist(V));
```
2. Compute **S**,**M**
   (a)
```
[S,M] = get_FEM_Mats(V,T);
```
3. Compute the RHS and exact solution vectors u_i^{exact}, f_i
   (a)
```
u_ex = @(th,ph) cos(th);
f = @(th,ph) -2*cos(th);
[phi, theta, ~] = cart2sph(V(:, 1), V(:, 2), V(:, 3));
theta = theta + pi/2; %put theta in [0,pi]
u_ex_i = u_ex(theta,phi);
f_i = f(theta,phi);
```
4. Solve the linear system
$$(\mathbf{S} + \epsilon \mathbf{I})\, \mathbf{u}^{FEM} = \mathbf{M}\mathbf{f}$$
5. compute and save the L^2 error
$$Error(N) = \sqrt{[\mathbf{u}^{\text{exact}} - \mathbf{u}^{FEM}]^T\, \mathbf{M}\, [\mathbf{u}^{\text{exact}} - \mathbf{u}^{FEM}]}$$
6. Refine the mesh by increasing N by 1 and repeat from step 1





Problem 2

Do something fun with the new code that you have! The following are a few suggestions, but feel free to do whatever you want here. I'm exited to see what you come up with!

1. Compute the eigen vectors of the discrete Laplacian $Lap = M^{-1}S$ and plot some of them using e.g

   ```
   f = abs(repmat(e_vec,1,3)).*V;
   C = V;
   h = trisurf(tri, f(:,1), f(:,2), f(:,3),C);
   set(h,'edgecolor','none')
   daspect([1 1 1])
   ```

   these eigen vectors are discrete approximations to the *eigen functions* of the Laplacian on a sphere and they're called *spherical harmonics*

2. Solve the heat or wave equation on sphere. For example, the heat equation can be discretized using e.g backward euler

   $$\mathbf{M}\left(\mathbf{u}^{n+1} - \mathbf{u}^{n}\right) = \mathbf{S}\mathbf{u}^{n+1}$$

3. Solve the heat equation but the unknowns are the verticees themselves! One can show that this is equivalent to what's called mean curvature flow and it's discretized very simply as

   $$\mathbf{M}\left(\mathbf{V}^{n+1} - \mathbf{V}^{n}\right) = \mathbf{S}\mathbf{V}^{n+1}$$

   where $\mathbf{V}^n$ is the $N_v \times 3$ array of vertices. **This looks best on a mesh that isn't a sphere, try 'the stanford bunny'**